AN-895
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

ADuC702x MicroConverter® I²C®-Compatible Interface

by Michael Looney

INTRODUCTION

This application note describes the hardware master and slave implementation of an I²C-compatible (inter-integrated circuit) interface using the ADuC702x family of precision Analog Devices, Inc. microcontrollers. This application note also contains example code showing how a master and a slave can communicate with each other using the I²C interface (see the Implementation of the Serial EEPROM Protocol section).

The main features of the I²C bus are:

- Only two bus lines are required, a serial data line (SDA) and a serial clock line (SCL). Both of these lines are bidirectional, meaning that both the master and the slave can operate as transmitters or as receivers.
- An I²C master can communicate with multiple slave devices. Because each slave device has a unique 7-bit address, single master/slave relationships can exist at all times even in a multislave environment.
- The master and slave can transmit and receive at up to 400 kbps.
- On-chip filtering rejects <50 ns spikes on the SDA and the SCL lines to preserve data integrity.

A typical block diagram of an I²C interface is shown in Figure 1.

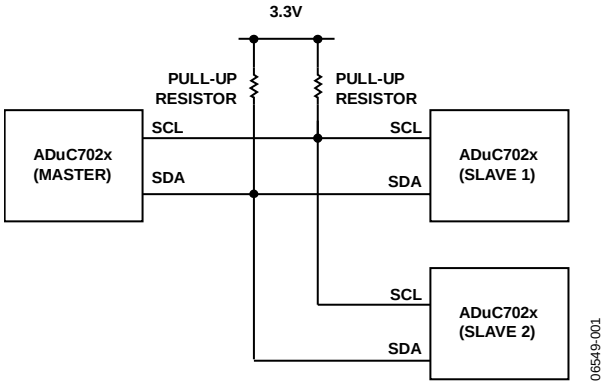

Figure 1. Single Master Multislave I²C Block Diagram

TABLE OF CONTENTS

I²C INTERFACE OVERVIEW

I²C is a 2-wire serial communication system developed by Philips that allows multiple masters and multiple slaves to be connected via two wires (SCL and SDA). In an I²C interface, there must be at least a single master and a single slave.

The SCL signal controls the data transfer between master and slave. The SCL signal is always transmitted from the master to the slave. The slave, however, does have the ability to pull this line low if it is not ready for the next transmission to begin. This is called clock stretching. One clock pulse must be generated for each data bit transferred.

The SDA signal is used to transmit or receive data. The SDA input must be stable during the high period of SCL. A transition of the SDA line while SCL is high is seen as a start or stop condition (see Figure 2 and Figure 3).

I²C FUNDAMENTALS

Start Condition

A typical data transfer sequence for an I²C interface starts with the start condition. The start condition is simply a high to low transition in the SDA line while the SCL line is pulled high (see Figure 2). The master is always responsible for generating the start condition. The start (and stop) conditions are the only times that the SDA line should change during a high period of the SCL line. During normal data transfer (including slave addressing), the data on the SDA line must be stable during the high period of the SCL line.

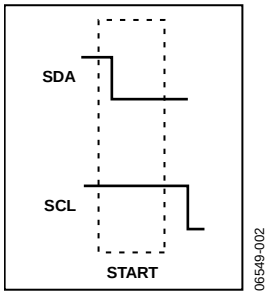

Figure 2. Start Condition for I²C

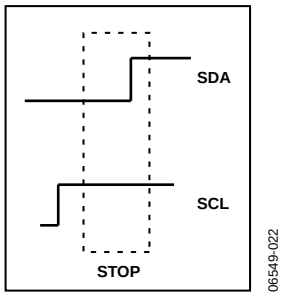

Figure 3. Stop Condition for I²C

Slave Address

After the start condition, the master sends a byte, most significant bit (MSB) first, on the SDA line, along with eight SCL pulses. The first seven bits of this byte is the 7-bit slave address. The slave only responds to the master if this 7-bit address matches the address of the slave device (or one of the four slave addresses). The eighth bit, the least significant bit (LSB), is the R/$\overline{W}$ status bit. The R/$\overline{W}$ status bit determines the direction of the message. If this bit is cleared, the master writes data to a selected slave. If this bit is set, the master expects to receive data from the slave. The master generates the clock in both cases.

If the slave receives the correct address, that is, the seven MSBs from the master match the seven MSBs of the I2C0ADR memory mapped register (MMR), the slave returns a valid ACK, pulls the SCL line low, and sets flags in the I2C0STA.

While the slave does all the manipulation of the I²C slave addressing automatically in hardware, as described previously, it is up to the master to output the slave address appropriately.

Acknowledge (ACK)/No Acknowledge (NACK)

If the slave address matches the address sent by the master, the slave automatically sends an acknowledge (ACK). Otherwise, it sends a no acknowledge (NACK). An ACK is seen as a low level on the SDA line on the ninth clock pulse. A NACK is seen as a high level on the SDA line on the ninth clock pulse (see Figure 4).

During data transfer, the ACK or the NACK is always generated by the receiver. However, the clock pulse required for the ACK is always generated by the master. The transmitter must release the SDA line (high) during the ACK clock pulse. For a valid ACK, the receiver must pull the SDA line low.

On the ADuC702x MicroConverter, both the ACK and the NACK are automatically generated in hardware, at the end of each byte in the reception.

If a master receives a NACK from a slave-receiver (either the slave did not respond to the slave address or the data transmitted), the master should generate the stop condition to abort the transfer (see the Data Transfer section).

A master receiver must signal the end of a data sequence to the slave-transmitter by generating a no acknowledge (NACK) after the last byte that was sent by the slave. Once the slave receives the NACK, it releases the SDA line to allow the master to generate the stop condition.

Data Transfer

In the I²C interrupt service routine (ISR), or in a polled implementation, the slave decides whether or not to transmit or receive depending on the status of the R/$\overline{W}$ bit sent by the master. The slave then either transmits or receives a bit on each clock sent by the master. It is up to the master to provide the nine clocks (eight for the data and one for the ACK) for the slave to transmit/receive data to/from the master. The I²C interrupt bit is set every time a valid data byte is transmitted or received by the slave.

Note again that in a slave-transmitter, master-receiver system the master must signal the end of a data sequence to the slave by sending a NACK after the last byte transmitted by the slave. Once the slave receives the NACK, it releases the SDA line to allow the master to generate the stop condition.

If a master wants to abort a data transfer or to interrupt the data transfer of another master on the bus, it can do this by sending a start condition followed by a stop condition.

Stop Condition

The data transfer sequence is terminated by the stop condition. A stop condition is defined by a low to high transition on the SDA line while SCL is high (see Figure 3).

The stop condition is always generated by the master. The master sends the stop condition once the master is satisfied that the data sequence is over or if it receives a NACK from the slave device. The reception of the stop condition resets the slave device into waiting for the slave address again.

The I²C interface on the ADuC702x parts can be configured to generate an interrupt on the stop condition. This is enabled by Bit 14 in the I2CxCFG MMR.

A typical transfer sequence is shown in Figure 5.

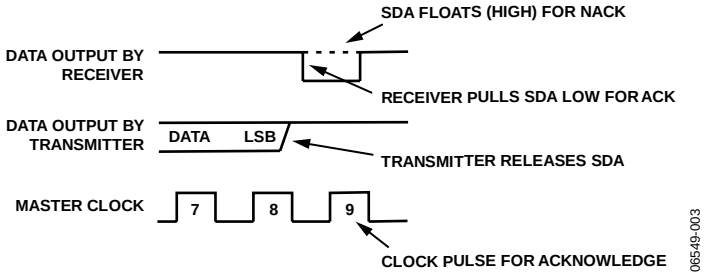

Figure 4. Acknowledge (ACK) and No Acknowledge (NACK) on the I²C Bus

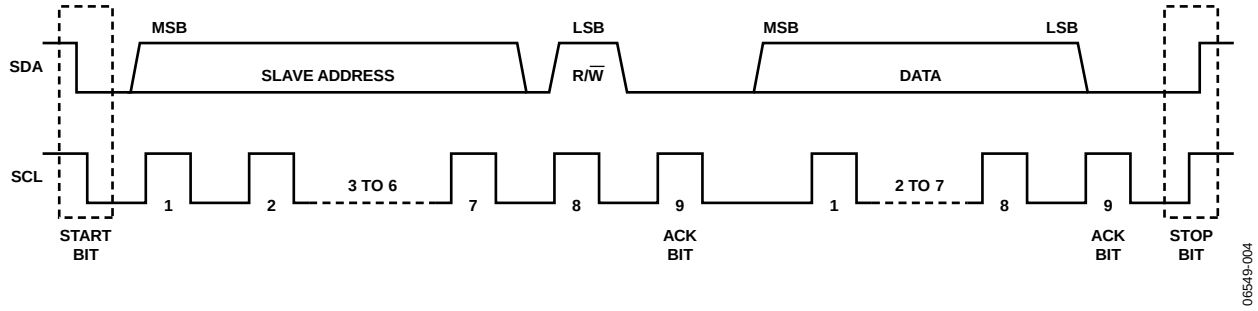

Figure 5. Typical I²C Transfer Sequence

Repeated Start Condition

A repeated start condition occurs when a second start condition is sent to a slave without a stop condition being sent in between. This allows the master to reverse the direction of the transfer, by changing the R/$\overline{W}$ bit without having to give up control of the bus.

An example of a transfer sequence is shown in Figure 6. This is generally used where the first data sent to the part sets up the register address to be read from.

An interrupt is generated when a repeated start + slave address is received. This can be differentiated from a start + slave address by using the status bits in the I2CxSSTA MMR.

Note that the ADuC702x device cannot directly generate an I^2C repeated start sequence when in master mode.

If a user has full control of the entire I^2C bus, a workaround is possible by disabling the generation of the stop condition before the desired second start condition. This is possible using the following steps:

1. Put a lower value pull-up resistor on the SDA line. For example, connect SDA to V_{CC} via a 4.7 kΩ resistor; connect SCL to V_{CC} via a 20 kΩ resistor.
2. Add the following code sequence:

```
I2C0CCNT = 0x0;    // Sets Start/Stop condition counter value to 0 - minimum value.
I2C0ADR = 0xA0;    // Write sequence
I2C0MTX = 0x7;     // Load the Tx FIFO
while ((I2C0FSTA & 0x30) != 0x00) {}  // Wait for the Tx FIFO to empty
I2C0CNT = 0x1;     // Read 2 bytes from the slave
I2C0ADR = 0xA1;    // Send out the Read condition
I2C0CCNT = 0x80;   // Set the Start/Stop counter to a nonzero value to re-enable the Stop
                   Condition
```

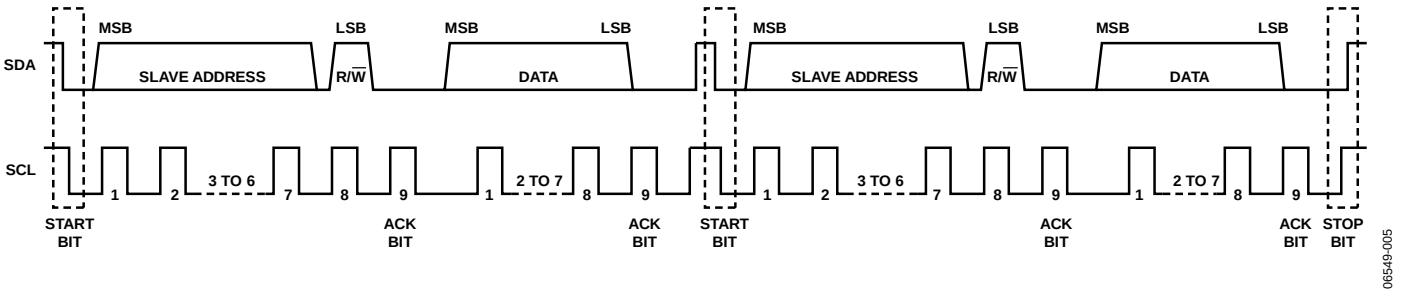

Figure 6. I^2C Repeated Start Sequence

Clock Stretching

In an I²C communication, the master device determines the clock speed. Unlike RS232, the I²C bus provides an explicit clock signal that relieves master and slave from synchronizing exactly to a predefined baud rate.

However, there are situations where an I²C slave is not able to cooperate with the clock speed given by the master and needs to slow down a little. This is done by a mechanism referred to as clock stretching.

An I²C slave is allowed to hold down the clock if it needs to reduce the bus speed. The master, on the other hand, is required to read back the clock signal after releasing it to a high state and wait until the line has actually gone high.

Bit 11 of the I2CxCFG MMR allows clock stretching.

SERIAL EEPROM PROTOCOLS

The ATMEL AT24C series serial EEPROM supports five commands:

- Random write
- Sequential write
- Current address read
- Random read
- Sequential read

These five commands are described in Figure 7 through Figure 11.

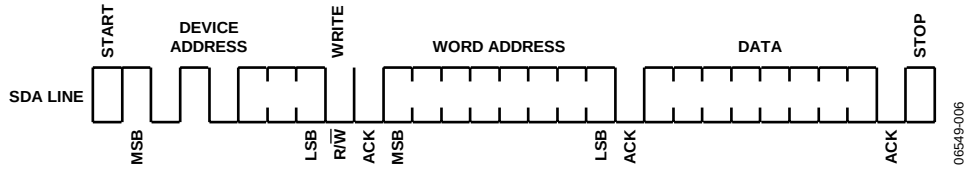

Figure 7. Random Write

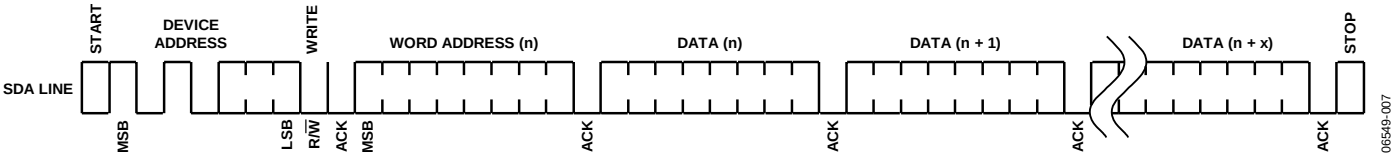

Figure 8. Sequential Write

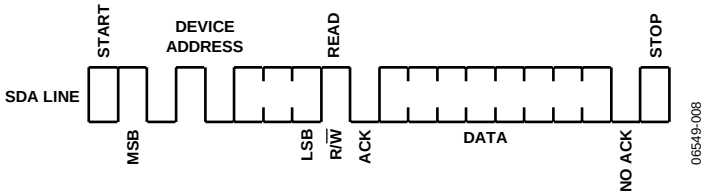

Figure 9. Current Read

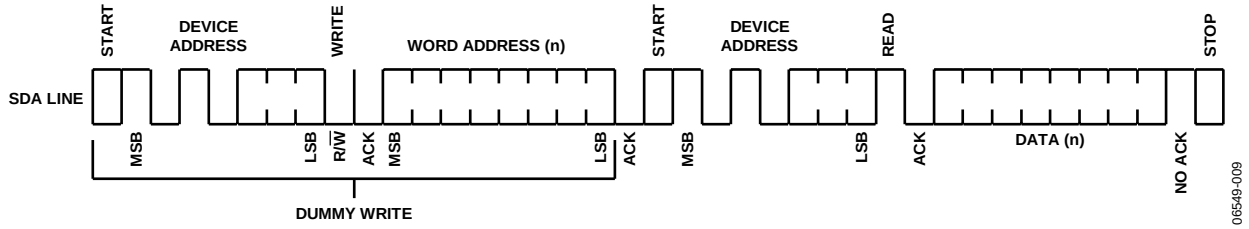

Figure 10. Random Read

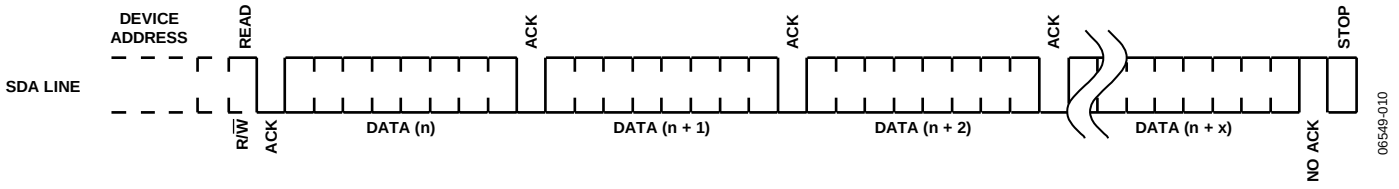

Figure 11. Sequential Read

I²C IMPLEMENTATION ON THE ADUC702x SERIES MICROCONVERTER

The ADuC702x series of parts contain two full hardware master and slave I²C ports. These ports support up to four addresses each and have configurable interrupts that allow the serial EEPROM commands to be implemented.

At a basic level, the I²C hardware interface behaves like a standard UART. There are receive and transmit buffers each of which consists of 2-byte FIFOs. This application note describes in detail the four types of communication (master/slave receive/transmit) and the use of the FIFO.

Use of the FIFO

There are four 2-byte FIFOs per I²C block:

- Master receive
- Master transmit
- Slave receive
- Slave transmit

The following sections describe the transmit FIFO and the receive FIFO.

Table 1. I2CxFSTA MMR Bit Descriptions

Bit No.	Description
31 to 10	Reserved.
9	Master Transmit FIFO Flush. *Set* by the user to flush the master Tx FIFO. This bit also flushes the slave receive FIFO. *Cleared* automatically once the master Tx FIFO is flushed.
8	Slave Transmit FIFO Flush. *Set* by the user to flush the slave Tx FIFO. *Cleared* automatically once the slave Tx FIFO is flushed.
7 to 6	Master Rx FIFO Status Bits. 00 FIFO empty 01 Byte written to FIFO 10 1 byte in FIFO 11 FIFO full
5 to 4	Master Tx FIFO Status Bits. 00 FIFO empty 01 Byte written to FIFO 10 1 byte in FIFO 11 FIFO full
3 to 2	Slave Rx FIFO Status Bits. 00 FIFO empty 01 Byte written to FIFO 10 1 byte in FIFO 11 FIFO full
1 to 0	Slave Rx FIFO Status Bits. 00 FIFO empty 01 Byte written to FIFO 10 1 byte in FIFO 11 FIFO full

Transmit FIFO

To transmit data, the I2C0STX/I2C0MTX registers must be loaded. Writing a byte to the Tx register is equivalent to writing to Byte 1 of the FIFO (see Figure 12).

- If Byte 0 is empty, the byte in Byte 1 gets pushed to Byte 0 automatically. This is described in the state machine (see Figure 13). Note that the states are visible to the user in the I2C0FSTA register.
- If Byte 0 is already full, the byte stays in Byte 1. Writing in Tx again overwrites Byte 1.

Setting the transmit FIFO flush bit in the I2CFSTA register empties the FIFO.

When a transmission occurs, Byte 0 is transmitted, Byte 1 is shifted to Byte 0, and the FIFO is in State 2.

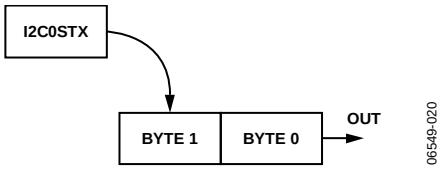

Figure 12. Transmit FIFO

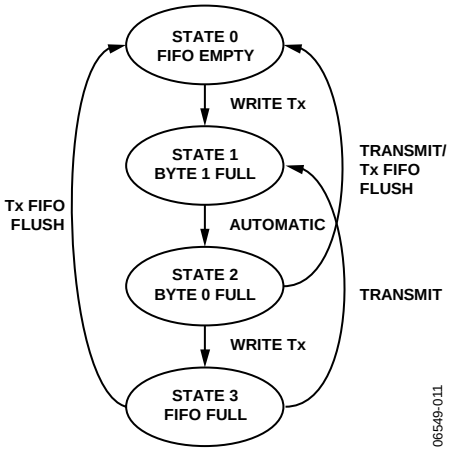

Figure 13. Transmit FIFO State Machine

Receive FIFO

When receiving data, the data arrives in Byte 0.

- If Byte 1 is empty, Byte 0 is shifted automatically to Byte 1.
- If Byte 1 is already full, Byte 0 stays until I2C0SRX is read (equivalent to reading Byte 1).
- If other data arrive while the FIFO is full, the slave delivers a NACK for the data and Bit 4 of I2C0SSTA is set.

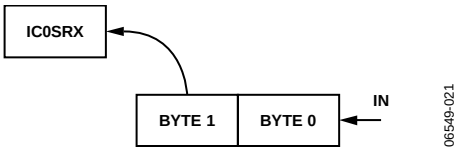

Figure 14. Receive FIFO

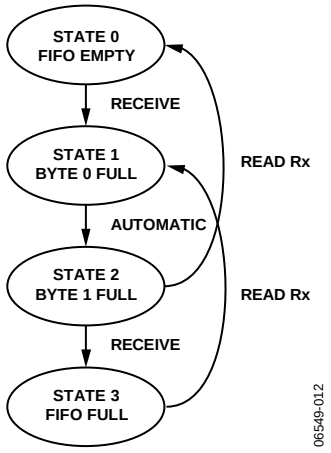

Figure 15. Receive FIFO State Machine

Master Transmit

In order to transmit a byte, the data must first be loaded into the transmit FIFO. The address of the slave must be specified in the I2C0ADR register. For a write of data, the write ($\overline{W}$) bit in the address register must be set to zero. Writing to the I2C0ADR register automatically generates a start condition.

On the first clock of each byte transmitted, an I^2C interrupt is generated. Bit 2 and Bit 1 in I2C0MSTA are set, indicating that the master has just transmitted a byte and that the FIFO is underflow. This allows the user to add a byte to the FIFO.

If only one byte is in the FIFO when initiating the transfer, the first I^2C interrupt occurs on the first clock of the address transmitted. If two bytes are in the FIFO, then the interrupt is generated on the first clock of the first byte transmitted.

The transmission ends when the FIFO is empty. A stop condition is automatically generated. This occurs 5.1 µs after the last byte is transmitted.

A simple example of how this operates is shown in the flowchart in Figure 16.

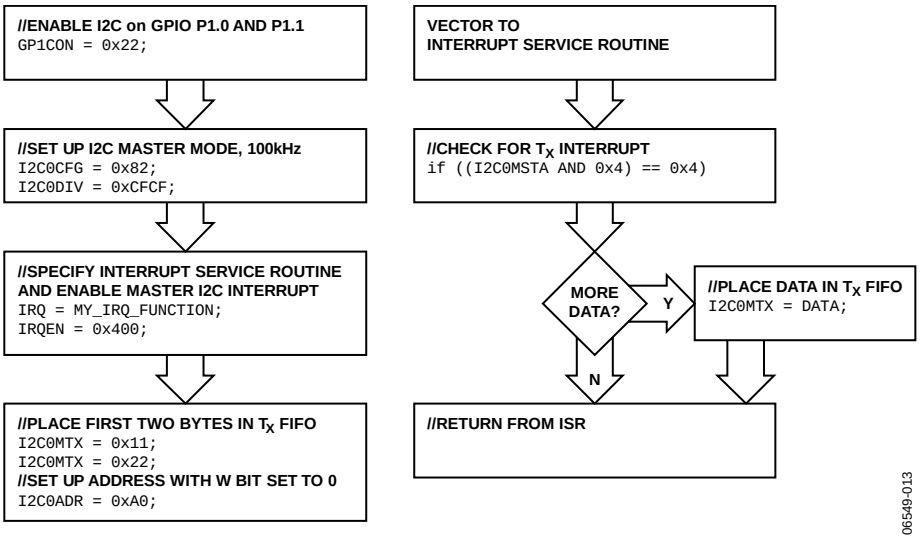

Figure 16. Master Transmit Flowchart

Slave Receive

As the data is received by an I²C slave, an interrupt is generated as each byte of data is placed in the receive FIFO, that is, after the ninth clock of each byte is received. If the FIFO is not read before a third byte is received, the interface automatically delivers a NACK for the last data transmitted, and Bit 4 of the I2CSSTA register is set, indicating a receive FIFO overflow.

To read data from the FIFO, the I2C0RX register is used. Bit 3 of the I2C0SSTA register indicates that the slave has received data. Only reading I2C0SRX clears this bit. Flushing the FIFO does not clear Bit 3.

The master automatically sends a stop condition after sending the last data.

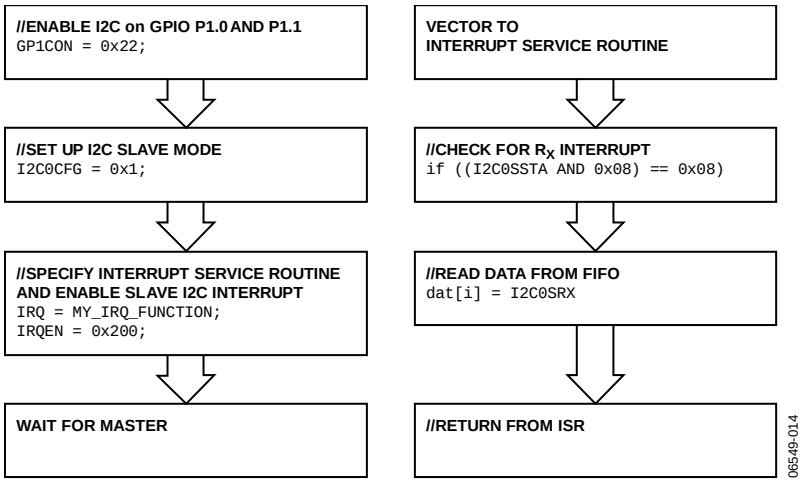

Figure 17. Slave Receive Flowchart

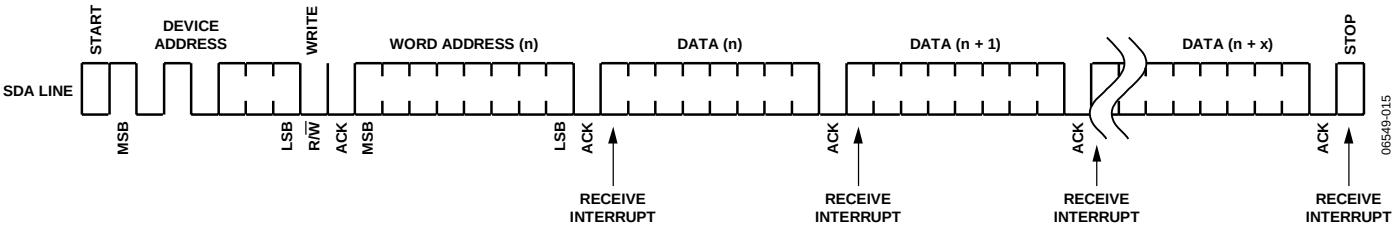

Figure 18. Example of a Slave Receive

Master Receive

In master mode, to read data from a slave, a similar approach is used. First, the number of bytes to be read is configured by the I2C0CNT register. This denotes the number of bytes to be read from the slave, plus one. It can have a value between 0 and 7 but can be reset during code execution in order to read larger amounts of data.

In order to start receiving data, the read (R) bit is set in the I2C0ADR register. This initiates a transfer with a start condition generated with the address and a R/$\overline{W}$ bit set by the I2C0ADR register. After each byte is received (after the ninth clock, ACK or NACK), an interrupt is generated. Bit 3 of I2C0MSTA is set, indicating that a byte has just been received. Only reading I2C0MRX clears this bit.

When the master does not need to receive more data, it automatically generates a NACK to the last byte received. This tells the slave to cease transmitting bytes and allows the master to then generate a stop condition.

If the data received is not read on time and the FIFO is full, the master delivers a NACK for the extra data received.

A flowchart of receiving four bytes from the slave is shown in Figure 19.

The I2C0CNT is a load register to an internal counter that is not user accessible. It is only a 3-bit counter, which means that the master can only be configured to transmit eight bytes at a time. However, it is possible to reload the internal counter by writing in I2C0ADR during a transmit sequence.

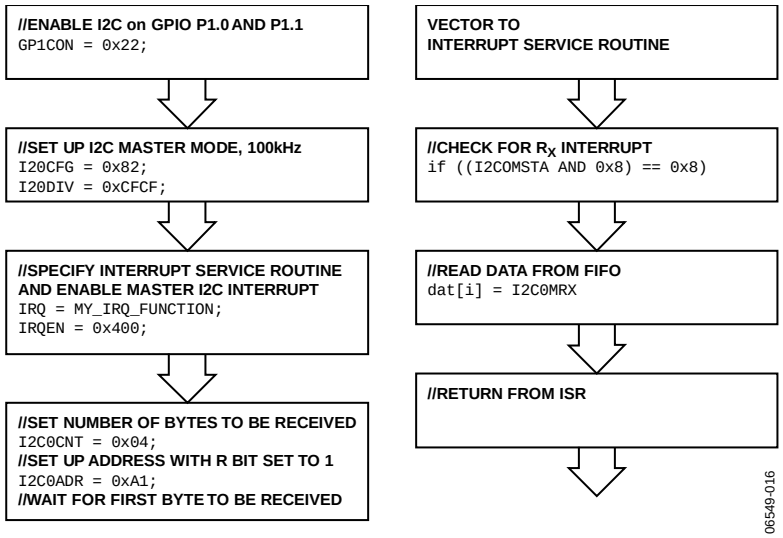

Figure 19. Master Receive Flowchart

Slave Transmit

The slave generates an interrupt on each request for data to be transmitted, the first occurring after the ACK of the address, that is, while Byte 0 of the FIFO is sent. Data needs to be preloaded into the slave Tx FIFO, otherwise the first read request from the master results in a NACK being generated. If the FIFO is preloaded with two sets of data, one interrupt occurs after the ACK of the address and then after the ACK of each byte sent. If the FIFO is preloaded with one set of data only, two interrupts occur after the ACK of the address, the FIFO emptying after sending the first data.

Once a byte has been transmitted, an interrupt is generated as long as the master continues to request data. Bit 2 of I2C0SSTA is set each time a byte is transmitted to the master.

An example of a slave responding to a request for data from a master is shown in Figure 21.

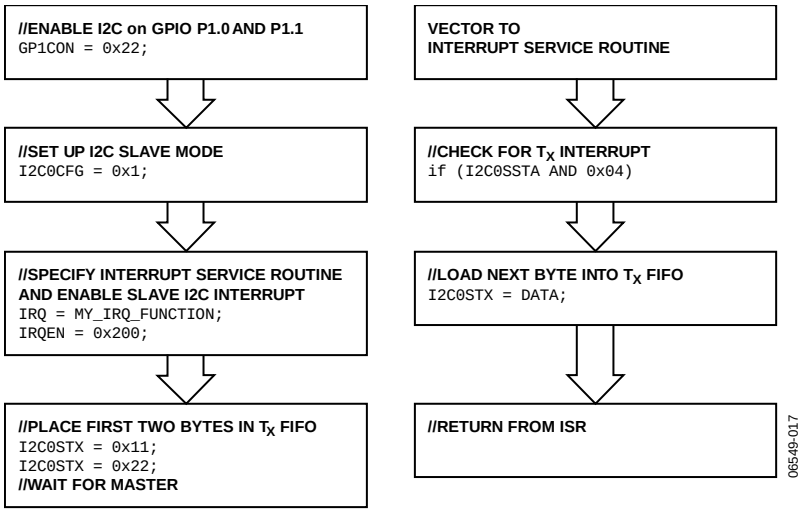

Figure 20. Slave Transmit Flowchart

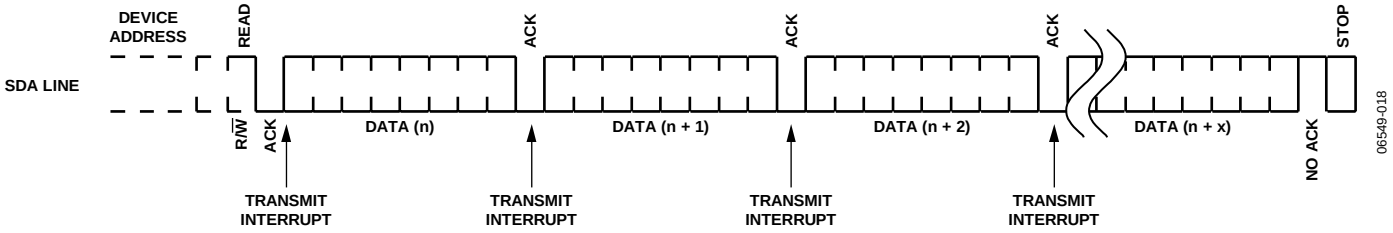

Figure 21. Example of a Slave Transmit

I²C REGISTER DEFINITIONS

The I²C peripheral interface consists of 15 registers in total:

- Four registers that include the transmit/receive master/slave MMR (I2CxSRX, I2CxSTX, I2CxMRX, I2CxMTX).
- Three status registers that include the master/slave/FIFO (I2CxMSTA, I2CxSSTA, I2CxFSTA).
- Eight configuration registers that include four slave addresses, one master address byte, one master clock divider, one master receive data count, and one I²C configuration register (I2CxID0, I2CxID1, I2CxID2, I2CxID3, I2CxADR, I2CxDIV, I2CxCNT, I2CxCFG).
- Other registers for general call are not discussed in this application note.

Four of these registers are described in detail in the following sections:

- I2CxDIV, the clock divider register.
- I2CxMSTA, the master status register, see Table 2.
- I2CxCFG, the I²C configuration register, see Table 3.
- I2CxSSTA, the slave status register, see Table 4.

I2CxMSTA: Master Status Register

Table 2. I2CxMSTA MMR Bit Descriptions

Bit No.	Description
7	Master Transmit FIFO Flush. *Set* by the user to flush the master Tx FIFO. This bit also flushes the slave Rx FIFO. *Cleared* automatically once the master Tx FIFO is flushed.
6	Master Busy. *Set* automatically if the master is busy. *Cleared* automatically.
5	Arbitration Loss. *Set* in multimaster mode if another master has the bus. *Cleared* when the bus becomes available.
4	No ACK. *Set* automatically if there is no acknowledge of the address by the slave device. *Cleared* automatically by reading the I2C0MSTA register.
3	Master Receive IRQ. *Set* after receiving data. *Cleared* automatically by reading the I2C0MRX register.
2	Master Transmit IRQ. *Set* at the end of a transmission. *Cleared* automatically by writing to the I2C0MTX register.
1	Master Transmit FIFO Underflow. *Set* automatically if the master transmit FIFO is underflowing. *Cleared* automatically by writing to the I2C0MTX register.
0	Master Tx FIFO Empty. *Set* automatically if the master transmit FIFO is empty. *Cleared* automatically by writing to the I2C0MTX register.

I2CxDIV, Clock Divider Register

This is a 16-bit register containing two 8-bit values, DIVH and DIVL. The value in this register sets up the speed of the I²C bus. This is set up according to the formula

$$f_{serialclock} = \frac{f_{UCLK}}{(2 + DIVH) + (2 + DIVL)}$$

where:

f_{UCLK} is the clock before the clock divider.
DIVH is the high period of the clock.
DIVL is the low period of the clock.

Thus, for 100 kHz operation,

DIVH = *DIVL* =0xCF (I2C0DIV = 0xCFCF)

and for 400 kHz,

DIVH = 0x28 *DIVL* = 0x3C (I2C0DIV = 0x283C)

I2CxCFG: I²C Configuration Register

Table 3. I2CxCFG MMR Bit Descriptions

Bit No.	Description
31 to 15	Reserved. These bits should be written by the user as 0.
14	Enable Stop Interrupt. *Set* by the user to enable the generation of an interrupt on receiving a stop condition after receive a valid start + matching address. *Cleared* by the user to disable the generation of an interrupt on receiving a stop condition.
13 to 12	Reserved. These bits should be written by the user as 0.
11	Enable Stretch SCL (Holds SCL Low). *Set* by the user to enable stretching of the SCL line. This bit instructs the I²C interface to hold SCL low if it is already low, or when it next goes low. *Cleared* by the user to disable stretching of the SCL line.
10	Reserved. This bit should be written by the user as 0.
9	Slave Tx FIFO Request Interrupt Enable. *Set* by the user to disable the slave Tx FIFO request interrupt. *Cleared* by the user to generate an interrupt request just after the negative edge of the clock for the R/$\overline{W}$ bit. This allows the user to input data into the slave Tx FIFO if it is empty. At 400 kbps and the core clock running at 41.78 MHz, the user has 45 clock cycles to take appropriate action, taking interrupt latency into account.
8	General Call Status Bit Clear. *Set* by the user to clear the general call status bits. *Cleared* automatically by hardware after the general call status bits have been cleared.
7	Master Serial Clock Enable Bit. *Set* by the user to enable generation of the serial clock in master mode. *Cleared* by the user to disable serial clock in master mode.
6	Loop Back Enable Bit. *Set* by the user to internally connect the transition to the reception, and to test user software. *Cleared* by the user to operate in normal mode.
5	Start Back-Off Disable Bit. *Set* by the user in multimaster mode. If losing arbitration the master tries to transmit again immediately. *Cleared* by the user to enable start back-off. The master after losing arbitration waits before trying to transmit again.
4	Hardware General Call Enable. When this bit and the general call enable bit are set, and have received a general call (Address 0x00) and a data byte, the devices checks the contents of the I2C0ALT against the receive register. If they match, the device has received a hardware general call. This is used if a device needs urgent attention from a master device without knowing which master it needs to turn to. The ADuC702x watch for these addresses. The device that requires attention embeds its own address into the message. All masters listen and the master that knows how to handle the device contacts its slave and acts appropriately. The LSB of the I2C0ALT register should always be written as a 1, as per the *I²C-Bus Specification*, version 2.1, January 2000.
3	General Call Enable Bit. *Set* this bit to enable the slave device to deliver an ACK for an I²C general call, Address 0x00 (write). The device then recognizes a data bit. If it receives a 0x06 as the data byte, that is, "reset and write programmable part of slave address by hardware," the I²C interface resets as per the *I²C-Bus Specification*. This command can be used to reset an entire I²C system. The general call interrupt status bit sets on any general call. It is up to the user to take correct action by setting up the I²C interface after a reset. If it receives a 0x04 as the data byte, that is, "write programmable part of slave address by hardware," the general call interrupt status bit sets on any general call. It is up to the user to take correct action by reprogramming the device address.
2	Reserved.
1	Master Enable Bit. *Set* by the user to enable the master I²C channel. *Cleared* by the user to disable the master I²C channel.
0	Slave Enable Bit. *Set* by the user to enable the slave I²C channel. A slave transfer sequence is monitored for the device address in I2C0ID0, I2C0ID1, I2C0ID2, and I2C0ID3. If the device address is recognized, the part participates in the slave transfer sequence. *Cleared* by the user to disable the slave I²C channel.

I2CxSSTA: Slave Status Register

Note: reading the status register modifies its contents. Only read it once and save its value in a variable during an ISR.

Table 4. I2CxSSTA MMR Bit Descriptions

Bit No.	Description
31 to 15	Reserved. These bits should be written by the user as 0.
14	Start Decode Bit. *Set* by the hardware if the device receives a valid start + matching address. *Cleared* either by an I^2C stop condition, or by an I^2C general call reset.
13	Repeated Start Decode Bit. *Set* by the hardware if the device receives a valid repeated start + matching address. *Cleared* by either an I^2C stop condition, by a read of the I2CxSSTA register, or by an I^2C general call reset.
12 to 11	ID Decode Bit. 00 Received address matched ID Register 0 01 Received address matched ID Register 1 10 Received address matched ID Register 2 11 Received address matched ID Register 3
10	Stop After Start and Matching Address Interrupt. *Set* by hardware if the slave device receives an I^2C stop condition after a previous I^2C start condition and matching address. *Cleared* by a read of the I2CxSSTA register.
9 to 8	General Call ID. 00 No general call 01 General call reset and program address 10 General call program address 11 General call matching alternative ID
7	General Call Interrupt.
6	Slave Busy. *Set* automatically if the slave is busy. *Cleared* automatically.
5	No ACK. *Set* if the master asks for data and no data is available. *Cleared* automatically.
4	Slave Receive FIFO Overflow. *Set* automatically if the slave receive FIFO is overflowing. *Cleared* automatically by reading I2C0SRX.
3	Slave Receive IRQ. *Set* after receiving data. *Cleared* automatically by reading the I2C0SRX register.
2	Slave Transmit IRQ. *Set* at the end of a transmission. *Cleared* automatically by writing to the I2C0STX register.
1	Slave Transmit FIFO Underflow. *Set* automatically if the slave transmit FIFO is underflowing. *Cleared* automatically by writing to the I2C0STX register.
0	Slave Transmit FIFO Empty. *Set* automatically if the slave transmit FIFO is empty. *Cleared* automatically by writing to the I2C0STX register.

IMPLEMENTATION OF THE SERIAL EEPROM PROTOCOL

This section covers the implementation of the five commands supported by the serial EEPROM specifications in the case of a single I²C address:

- Current address read
- Random read
- Random write
- Sequential read
- Sequential write

In the interrupt service routine, the status register must be read once and its value saved. The flowchart of the interrupt service routine is shown in Figure 22.

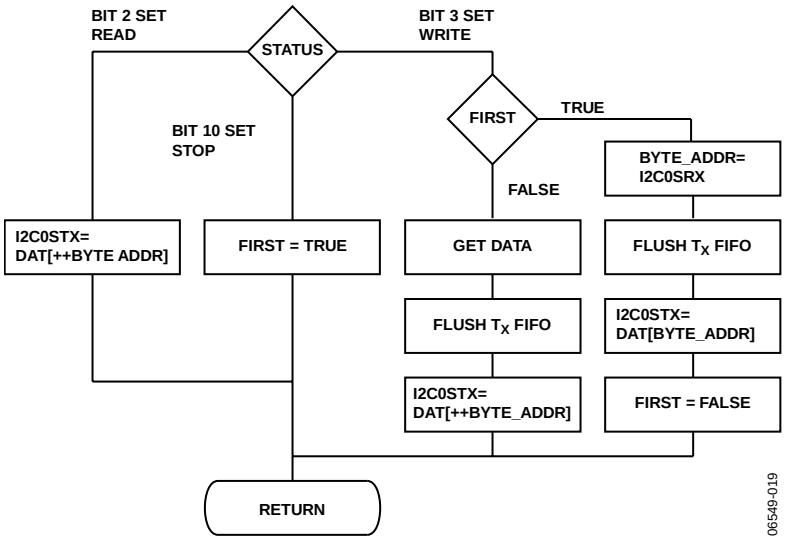

Figure 22. Slave Interrupt Service Routine

See AN-895 Companion Code.zip for companion codes.

Command Codes

I²C Configuration

```
I2CCFG = 0x4001;   // Enable slave, enable STOP detect
I2C0ID0 = 0xA0;    // Slave ID
I2C0STX = dat[0];  // Set initial data
```

Variable Initialization

```
Byte_addr = 0
First = 1
```

Enable I²C Slave Interrupt

```
IRQEN = 0x200;     // I2C0 Slave Interrupt
while (1)    // Wait for interrupt
```