CONVENIENT AND ISOLATED POWER MEASUREMENTS VIA SPI OR I²C USING ACS71020/ACS37800 POWER MONITORING ICS

By Wade Bussing and Robert Bate
Allegro MicroSystems

ABSTRACT

This application note describes the I²C and SPI interfaces on the ACS71020/ACS37800 Power Monitoring ICs from Allegro MicroSystems. Power Monitoring IC is used throughout this application note to indicate both the ACS71020 and ACS37800, unless otherwise noted.

Detailed examples include reading and writing registers on the Power Monitoring IC via the I²C and SPI interfaces. Other sections describe converting the device's fixed-point register contents to real-world values. Application schematics, scope plots, and the associated Arduino example code are also provided. See Appendix A for full application schematics. See Appendix B for full source code including an Arduino-compatible ".ino" sketch file. The sketch file and schematics are also available for download on Allegro's Software Portal [1].

INTRODUCTION

The Power Monitoring IC provides users with an accurate and isolated solution for sensing current, voltage, and power in a single IC. With its I²C and SPI interfaces, the Power Monitoring IC provides convenient access to sixteen different power measurements. For more information, refer to the ACS71020 and ACS37800 device datasheets [2].

Examples listed in this application note make use of the "Teensy" 3.2 microcontroller [3] and Arduino software environment [4]. While this document focuses on implementation using the Teensy 3.2, the practices and example code translate directly to other Arduino boards.

I²C OVERVIEW

The I²C bus is a synchronous, two-wire serial communication protocol that provides a full-duplex interface between two or more devices. The bus specifies two logic signals:

1. Serial Clock Line (SCL) output by the Master.
2. Serial Data Line (SDA) output by either the Master or the Slave.

Any device has the potential to become the bus Master and assume control over the SCL and SDA logic signal lines.

The block diagram in Figure 1 illustrates the I²C bus topology.

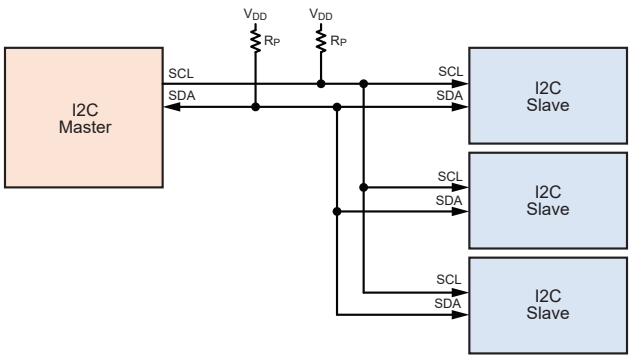

Figure 1: I²C bus diagram showing Master and Slave devices

[1] Allegro's Software Portal: https://registration.allegromicro.com.
[2] ACS71020 datasheet: https://www.allegromicro.com//en/products/sense/current-sensor-ics/zero-to-fifty-amp-integrated-conductor-sensor-ics/acs71020.
ACS37800 datasheet: https://www.allegromicro.com//en/products/sense/current-sensor-ics/zero-to-fifty-amp-integrated-conductor-sensor-ics/acs37800.
[3] Teensy homepage: https://www.pjrc.com/teensy/teensy31.html.
[4] Arduino homepage: https://www.arduino.cc.

DATA TRANSMISSION

The transmission of data over I²C is composed of several steps outlined in the sequence below.

1. Start Condition: Defined by a negative edge of the SDA line, initiated by the Master while SCL is high.
2. Address Cycle: 7-bit Slave address, plus 1 bit to indicate write (0) or read (1), followed by an Acknowledge bit.
3. Data Cycles: Reading or writing 8 bits of data, followed by an Acknowledge bit. This cycle can be repeated for multiple bytes of data transfer. The first data byte on a write could be the register address. See the following sections for further information.
4. Stop Condition: Defined by a positive edge on the SDA line while SCL is high.

Except to indicate Start or Stop conditions, SDA must remain stable while the clock signal is high. SDA may only change states while SCL is low. It is acceptable for a Start or Stop condition to occur at any time during the data transfer. The Power Monitoring IC will always respond to a Read or Write request by resetting the data transfer sequence.

The clock signal SCL is generated by the Master, while the SDA line functions as either an input or open drain output, depending on the direction of data transfer. Timing of the I²C bus is summarized in the timing diagram in Figure 2. Signal references and definitions of these names can be found the ACS71020 and ACS37800 device datasheets.

I²C BUS SPEEDS

Common I²C bus speeds are 100 kbps standard mode and 10 kbps low-speed mode, but arbitrarily low clock frequencies are also allowed. Recent revisions of the I²C protocol can host more nodes and run at faster speeds including 400 kbps Fast mode and 1 Mbps Fast mode plus (Fm+), which are all supported by the Power Monitoring IC. Note the I²C specification outlines an additional 3.4 Mbps High Speed mode that is not supported by the Power Monitoring IC.

IMPLEMENTATION OF I²C WITH THE POWER MONITORING IC

The Power Monitoring IC may only operate as a Slave I²C device, therefore it cannot initiate any transactions on the I²C bus.

The Power Monitoring IC will always respond to a Read or Write request by resetting the data transfer sequence. The state of the Read/Write bit is set low (0) to indicate a Write cycle and set high (1) to indicate a Read cycle. The Master monitors for an Acknowledge bit to confirm the Slave device (the Power Monitoring IC) is responding to the address byte sent by the Master. When the Power Monitoring IC decodes the 7-bit Slave address as valid, it responds by pulling SDA low during the ninth clock cycle.

When a data write is requested by the Master, the Power Monitoring IC pulls SDA low during the clock cycle following the data byte to indicate that the data has been successfully received. After sending either an address byte or a data byte, the Master must release the SDA line before the ninth clock cycle, allowing the handshake process to occur.

The I²C slave address used for the Power Monitoring IC throughout this application note is 127. For information on selecting other I²C Slave addresses, refer to the ACS71020 and ACS37800 device datasheets.

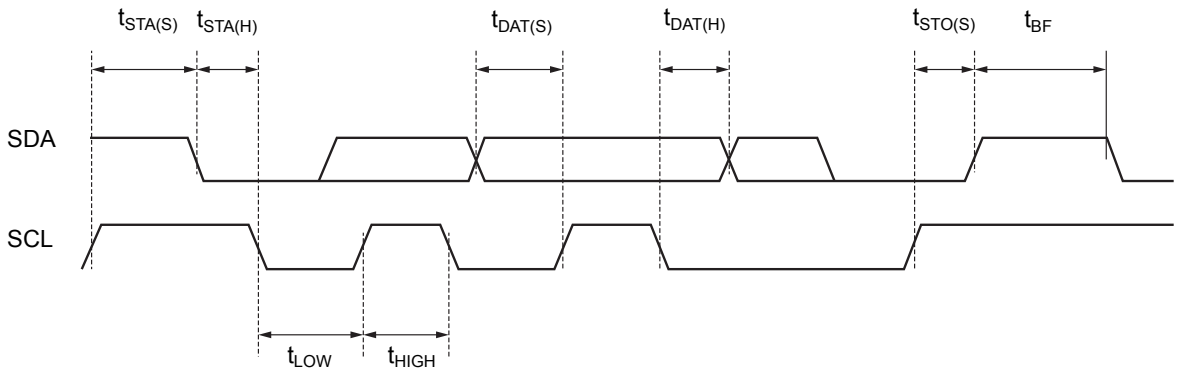

Figure 2: I²C Input and Output Timing Diagram

WRITE CYCLE OVERVIEW

The write cycle to access registers on the Power Monitoring IC are outlined in the sequence below.

1. Master initiates Start Condition.
2. Master sends 7-bit slave address and the write bit (0).
3. Master waits for ACK from Power Monitoring IC.
4. Master sends 8-bit register address (limited to 0-127 on Power Monitoring IC).
5. Master waits for ACK from Power Monitoring IC.
6. Master sends 0:7 bits of data.
7. Master waits for ACK from Power Monitoring IC.
8. Master sends 8:15 bits of data.
9. Master waits for ACK from Power Monitoring IC.
10. Master sends 16:23 bits of data.
11. Master waits for ACK from Power Monitoring IC.
12. Master sends 24:31 bits of data.
13. Master waits for ACK from Power Monitoring IC.
14. Master initiates Stop Condition.

The I²C write sequence is further illustrated in the timing diagrams below in Figure 3.

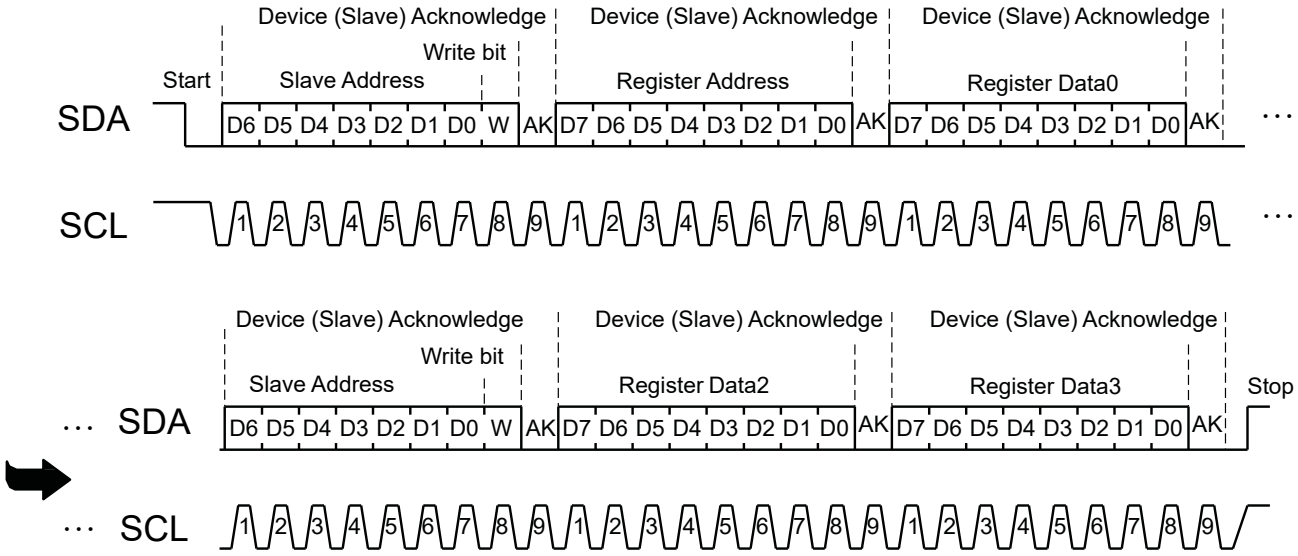

Figure 3: I²C Write Timing Diagram

READ CYCLE OVERVIEW

The I²C read cycle to access registers on Power Monitoring IC is outlined in the sequence below.

1. Master initiates Start Condition.
2. Master sends 7-bit slave address and the write bit (0).
3. Master waits for ACK from Power Monitoring IC.
4. Master sends 8-bit register address.
5. Master waits for ACK from Power Monitoring IC.
6. Initiate a Start Condition. This time it is referred to as a Restart Condition.
7. Master sends 7-bit slave address and the read bit (1).
8. Master waits for ACK from Power Monitoring IC.
9. Master receives 0:7 bits of data.
10. Master sends ACK to Power Monitoring IC.
11. Master receives 8:15 bits of data.
12. Master sends ACK to Power Monitoring IC.
13. Master receives 16:23 bits of data.
14. Master sends ACK to Power Monitoring IC.
15. Master receives 24:31 bits of data.
16. Master sends NACK to Power Monitoring IC.
17. Master initiates Stop Condition.

The I²C read sequence is further illustrated in the timing diagrams in Figure 4.

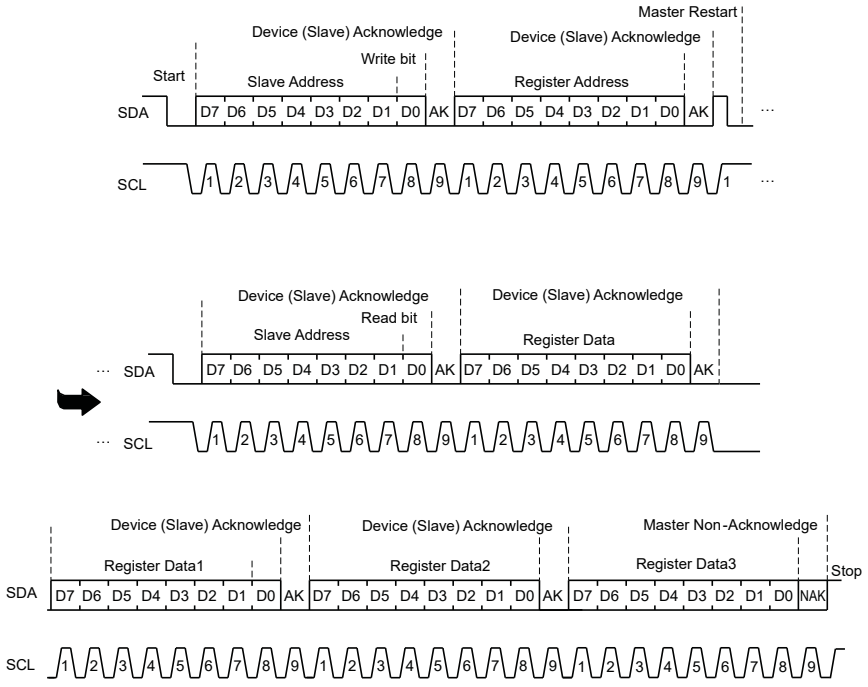

Figure 4: I²C Read Timing Diagram

APPLICATION EXAMPLE: WRITING TO A REGISTER VIA I²C

To write any device over I²C, the device's slave address must be known, and the read/write bit must be considered.

In this example, the ACS71020 device's slave address is 127 (0x7F), which is coupled to a 1-bit write command (0). A value of 10 (0x0000000A) is written to register 0x0C. These bits correspond to a relatively benign "vmrs_avg_1" register, so it is acceptable to change in this example. A successful I²C write is shown in Figure 5.

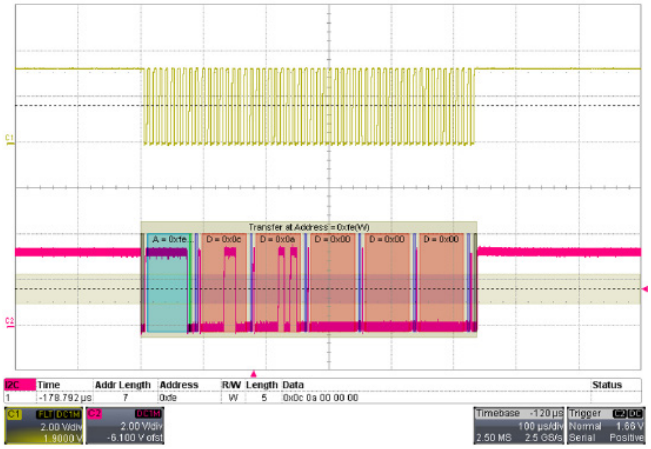

Figure 5: I²C Write Sequence

The components of a real I²C write message in Figure 5 are analyzed below. The scope used in this demonstration is equipped with an I²C decoding package, which makes for simple analysis.

1. The device's slave address and write bit are coupled to form an 8-bit message: [0x7F,0x0] = 0xFE.
2. The register address 0x0C is written to the device to indicate which register will be changed.
3. The master provides data to the slave device beginning with the least significant byte [0:7].
4. The second byte is sent [8:15].
5. The third byte is sent [16:23].
6. The fourth byte is sent [24:31].

APPLICATION EXAMPLE: READING FROM A REGISTER VIA I²C

This example will read the 0x0C register to confirm the write sequence from the previous example was successful. All the conditions from the previous write example apply.

The read sequence is a two-step process in which the master writes to the slave to indicate the desired register to read from. This is followed by a read command to begin the data transfer. A successful I²C read is shown in Figure 6.

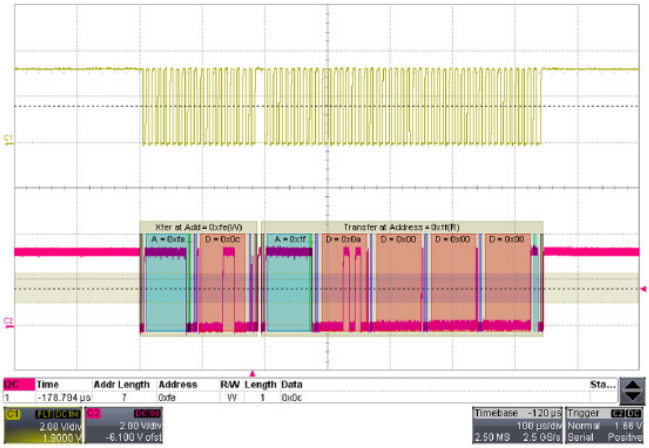

Figure 6: I²C Read Sequence

The components of a real I²C read message in Figure 6 are analyzed below.

1. The device's slave address and write bit are coupled to form an 8-bit message: [0x7F,0x0] = 0xFE.
2. The register address 0x0C is written to the device to indicate which register will be read.
3. The device's slave address and a read bit are coupled to form an 8-bit message: [0x7F,0x1] = 0xFF.
4. The slave returns data starting with the least significant byte [0:7].
5. The second byte is sent [8:15].
6. The third byte is sent [16:23].
7. The fourth byte is sent [24:31].

Note that the data returned in step 4 is expected based on the write sequence example in Figure 5.

SPI OVERVIEW

The SPI bus is a 4-wire synchronous serial communication protocol that provides a full-duplex interface between a single master and one or more slaves. The bus specifies four logic signals:

1. SCLK (Serial Clock) output by the master.
2. MOSI (Master-Out Slave-In) output by the master.
3. MISO (Master-In Slave-Out) output by the slave.
4. CS (Chip Select) output by the master, active low.

The block diagram shown in Figure 7 illustrates the I2C bus topology.

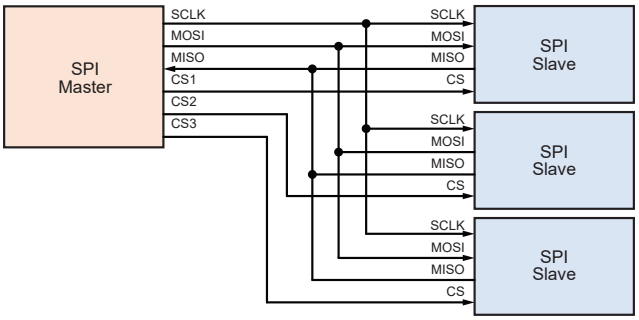

Figure 7: SPI bus diagram showing Master and Slave devices

DATA TRANSMISSION

To begin communication, the master selects the slave device with a logic level 0 on the chip select line. If a waiting period is required, such as for analog-to-digital conversion, the master must wait for at least that period before issuing clock cycles.

During each SPI clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is repeated until all bits are transferred even when only one-directional data transfer is intended. When complete, the master stops toggling the clock signal, and deselects the slave with a logic level 1 on the chip select line. The information transfer is illustrated in Figure 8.

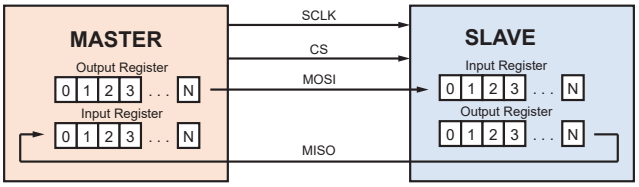

Figure 8: SPI Data Transfer

CLOCK FREQUENCY, POLARITY, PHASE, AND MODE

SPI is a flexible protocol—the bus master configures the clock frequency, polarity, and phase supported by the slave device. By convention, polarity and phase are named CPOL and CPHA respectively. CPOL controls the polarity of the clock and when it is 0, the idle state of the clock is 0, and the active state is 1. When CPOL is 1, the idle state of the clock is 1 and the active state is 0. CPHA controls the phase and is dependent on the value of CPOL. The combinations of polarity and phases are often referred to as modes, which are commonly numbered according to the convention in Table 1.

Table 1: SPI Modes

Mode	CPOL	CPHA	Description
0	0	0	Clock active high. Data captured on rising edge of the clock and output on the falling edge.
1	0	1	Clock active high. Data captured on falling edge of the clock and output on the rising edge.
2	1	0	Clock active low. Data captured on falling edge of the clock and output on the rising edge.
3	1	1	Clock active low. Data captured on rising edge of the clock and output on the falling edge.

MODE 0

CPOL and CPHA are 0. The idle state of the clock is 0 and the active state is 1. Data is captured on the rising edge of the clock and output on the falling edge. It is important to note that both the master and slave must output the first bit of data as soon as the chip select goes low. The blue line shows when the data is captured, and the red line shows when the data is output.

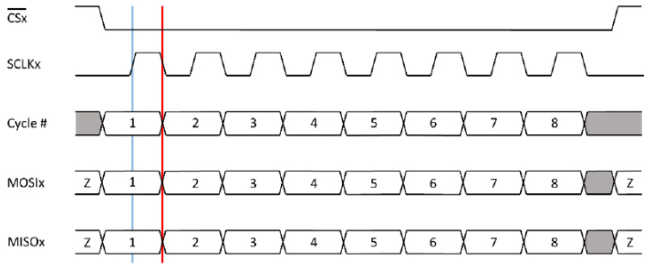

Figure 9: SPI Mode 0, CPOL = 0, CPHA = 0

MODE 1

CPOL is 0 and CPHA is 1. The idle state of the clock is 0 and the active state is 1. Data is captured on the falling edge of the clock and output on the rising edge. The blue line shows when the data is captured, and the red line shows when the data is output.

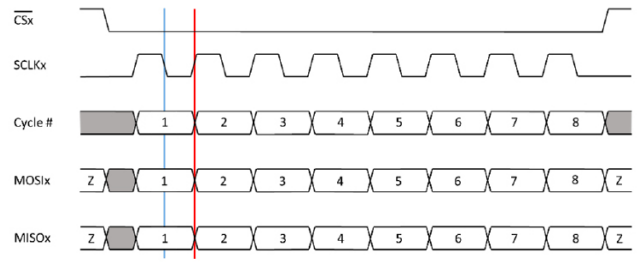

Figure 10: SPI Mode 1, CPOL = 0, CPHA = 1

MODE 2

CPOL is 1 and CPHA is 0. The idle state of the clock is 1 and the active state is 0. Data is captured on the falling edge of the clock and output on the rising edge. It is important to note that both the master and slave must output the first bit of data as soon as the chip select goes low. The blue line shows when the data is captured, and the red line shows when the data is output.

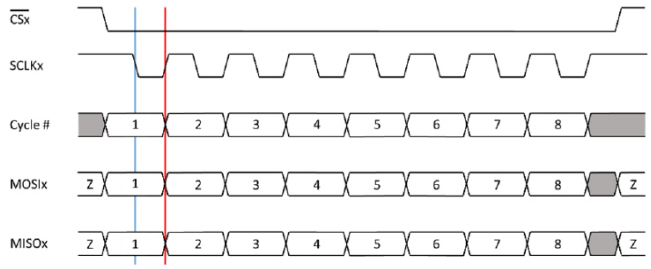

Figure 11: SPI Mode 2, CPOL = 1, CPHA = 0

MODE 3

CPOL is 1 and CPHA is 1. The idle state of the clock is 1 and the active state is 0. Data is captured on the rising edge of the clock and output on the falling edge. This is the mode that the Power Monitoring IC uses. The blue line shows when the data is captured, and the red line shows when the data is output.

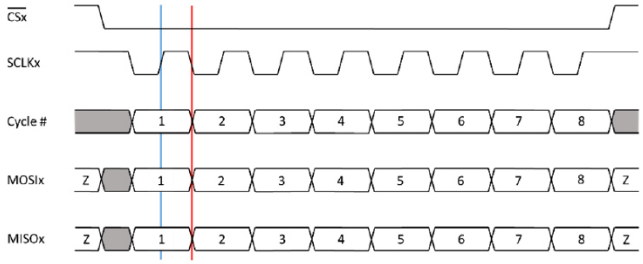

Figure 12: SPI Mode 3, CPOL = 1, CPHA = 0

SPI IMPLEMENTATION ON THE POWER MONITORING IC

The Power Monitoring IC provides a full-duplex 4-pin SPI interface using SPI mode 3, as shown in Figure 12. An SPI transaction is a minimum of 40 bits in length and the data is transmitted with least significant bit first.

Note: The Power Monitoring IC MISO pin continues to drive the MISO line when CS goes high. This may prevent other devices from communicating properly. It is recommended that the Power Monitoring IC be the only device on the SPI bus if using SPI communication.

INTERFACE TIMING

The Power Monitoring IC SPI interface operates in pure Slave mode, meaning the Master has full control over the SCLK, CS, and MOSI data lines. The Master may maximize data throughput up to $f_{SCLK(MAX)}$ of 10 MHz.

Figure 13 and Figure 14 show the timing diagrams for write and read cycles.

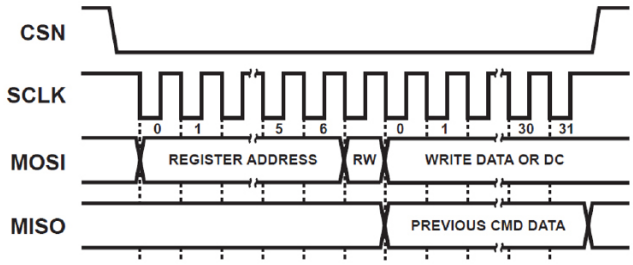

Figure 13: SPI Write Timing

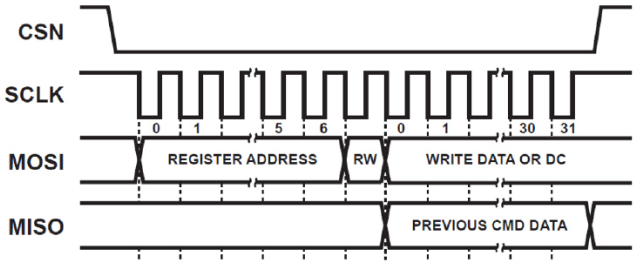

Figure 14: SPI Read Timing

WRITE CYCLE OVERVIEW

Write cycles consist of 7 address bits corresponding to the serial register, a 1-bit R/W asserted high, and 32 data bits. The Power Monitoring IC SPI expects the least significant byte is sent first.

MOSI bits are clocked in on the rising edge of the SCLK signal generated by the Master. The complete SPI packet is latched on the rising edge of the CS signal generated by the Master.

The simultaneous MISO signal output represents the contents of the corresponding SPI read packet, which includes 32 bits of data. The data bits correspond to the register contents selected during the previous read command. In the case where no previous read command was issued, the device will transmit all zeros. The write sequence is outlined below:

1. Master sets Chip Select Low.
2. Master sends 7-bit register address + write bit in the 8th bit.
3. Master sends 0:7 bits of data.
4. Master sends 8:15 bits of data.

5. Master sends 16:23 bits of data.
6. Master sends 24:31 bits of data.
7. Master sets Chip Select High.

READ CYCLE OVERVIEW

Read cycles have two stages: a Read command which selects the serial register address, followed by another Read command to receive the data from the selected register. Both commands consist of 1-bit sync (low), 7 address bits identifying the target register, a 1-bit R/W asserted low, and 32 data bits (all zeros because no data is being written). The Power Monitoring IC expects the least significant byte is sent first.

In the first stage, as with the Write command, the Read command MOSI bits are clocked-in on the rising edge of the Master generated signal, such that the Master can sample them on the SCLK rising edges. Because an SPI Read command can transmit 32 data bits at one time, and the device registers are four bytes, the entire 32-bit contents of one serial register may be transmitted with one SPI frame.

The SCLK signal and data are latched on the rising edge of the chip select (CS) signal. During the first Read stage, the simultaneous MISO signal output is the contents of the SPI read data from the previous Read command cycle. In the second stage, the Read command continues on the next falling edge of the Master-generated (CS) signal. The MISO bits are the contents of the register selected during the first stage, read 16 bits at a time. The MISO bits transmit on the falling edges of the SCLK, such that the Master can sample them on the SCLK rising edges.

The read cycle is summarized below.

1. Master sets Chip Select Low.
2. Master Sends 7-bit register address + read bit in the 8th bit.
3. Master sends 0:7 bits of data (0x0).
4. Master sends 8:15 bits of data (0x0).
5. Master sends 16:23 bits of data (0x0).
6. Master sends 24:31 bits of data (0x0).
7. Master sets Chip Select High.
8. Master sets Chip Select Low.
9. Master Sends 7-bit register address + read bit in the 8th bit.
10. Master sends 0:7 bits of data (0x0) and simultaneously receives 0:7 bits of data from the device.
11. Master sends 8:15 bits of data (0x0) and simultaneously receives 8:15 bits of data from the device.
12. Master sends 16:23 bits of data (0x0) and simultaneously receives 16:23 bits of data from the device.
13. Master sends 24:31 bits of data (0x0) and simultaneously receives 24:31 bits of data from the device.
14. Master sets Chip Select High.

APPLICATION EXAMPLE: WRITING TO A REGISTER VIA SPI

In this example, a value of 10 (0x0000000A) will be written to the register 0x0C. These bits correspond to a relatively benign "vmrs_avg_1" register, so it is acceptable to change in this example. A successful SPI write is shown in Figure 15.

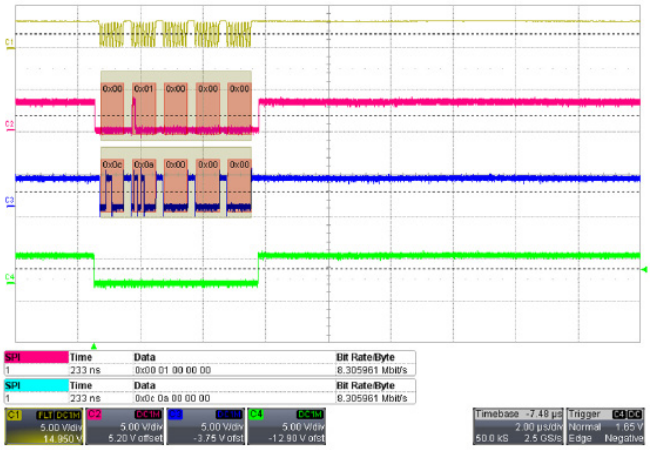

Figure 15: SPI Write Sequence

Below, the components of a real SPI write message in Figure 15 are analyzed.

1. The write bit and 7-bit register address are coupled to form an 8-bit message: [0x,0x0C] = 0x0C.
2. The data is sent over the MOSI line starting with the least significant byte [0:7].
3. The second byte is sent [8:15].
4. The third byte is sent [16:23].
5. The fourth byte is sent [24:31].

APPLICATION EXAMPLE: READING FROM A REGISTER VIA SPI

The following read example will confirm the data written during the write example above. A successful SPI read sequence is shown in Figure 16.

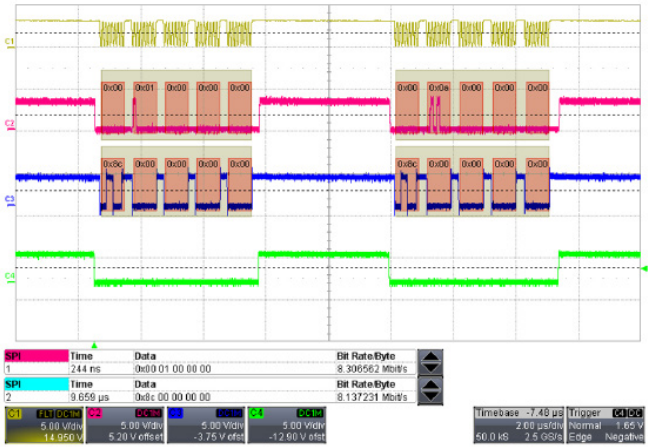

Figure 16: SPI Read Sequence

The components of a real SPI read message in Figure 16 are analyzed below.

1. The read bit and 7-bit register address are coupled to form an 8-bit message: [0x1,0x0C] = 0x8C.
2. The read instruction is sent over the MOSI line. The remaining 32 data bits are left zero because nothing is being written to the device.
3. The master sends a second read command. Data from the first read request is returned on the MISO line starting with the least significant byte [0:7].
4. The second byte is sent [8:15] and received.
5. The third byte is sent [16:23] and received.
6. The fourth byte is sent [24:31] and received.

Note that the data returned in step 3 is expected based on the write sequence example in Figure 15.

CUSTOMER WRITE ACCESS

An access code must be sent to the device prior to writing most of the volatile registers or EEPROM on the Power Monitoring IC, but some free space EEPROM registers may be written to regardless of the access code. Any register or EEPROM location may be read at any time regardless of the access mode.

To enter customer access mode, an access command must be sent via the I²C, or SPI interfaces. The command consists of a serial write operation with the address and data values shown in Table 2. There is no time limit for when the code may be entered. Once the customer access mode is entered, it is not possible to change access modes without power-cycling the device.

Table 2: Customer Access Code

Register Name	Address	Data
Access_code	0x2F	0x4F70656E
Customer_access	0x30	–

APPLICATION EXAMPLE: irms REGISTER

Like most volatile registers on the Power Monitoring IC, the irms and vrms measurements are fixed-point numbers. vrms and irms registers contain the most recently calculated values for rms voltage and rms current over the previous input cycle. For converting a fixed-point number to a real-world value, examine the irms measurement. The irms and vrms values are saved in the 0x20 address and are further described in Table 3.

Table 3: vrms and irms registers

Address	Bits	Name	Description
0x20	15:0	vrms	Most recently calculated vrms value
	30:16	irms	Most recently calculated irms value

The field "irms" is an unsigned 15-bit fixed point number with 14 fractional bits and a step size of 2^{-14}. This means that the value has a range from 0 to almost 2 (1.99939). Knowing this math, and the device's nominal current, it is easy to convert these 15 bits to a real-world value.

The ACS71020KMABTR-30B3-I2C will be reviewed in this application example. The -30B3 device has a nominal current range of 30 A, a full-scale range of 60 A, and a typical sensitivity of 1092 LSB/A.

With approximately 10 A_{RMS} (14.1421 A_{PEAK}) applied, the irms register reads 5926 codes. Note that the actual values may vary due to small differences in device sensitivity.

The "irms" number can be multiplied by the full-scale current range (30 A in this case) to convert to amps. Given that irms is an unsigned 15-bit fixed point number with 14 fractional bits and a step size of 2^{-14}, the code value can be converted to amps using the following equation.

Equation 1:

$$\text{code} \times \text{step size} \times \text{nominal current range}$$

Inputting real values into Equation 1 confirms the 10 A_{RMS} measurement.

$$5926 \times 2^{-14} \times 30\ A_{RMS} = 10.8508\ A_{RMS}$$

CONCLUSION

The ACS71020 and ACS37800 are highly versatile power monitoring ICs from Allegro MicroSystems with sixteen different power monitoring registers that are easily accessible over I^2C and SPI interfaces.

The application schematics and Arduino ".ino" sketch file used with this application note are listed in Appendix A and Appendix B. This material is available for download on Allegro's Software Portal.

APPENDIX A: APPLICATION SCHEMATICS

Application schematics and Teensy 3.2 connections are shown in Figure 17. R2 through R5 are installed for I²C assemblies as that protocol specifies an open drain output. R2 through R5 are removed for SPI assemblies as that protocol specifies push/pull structures on the pins. R6 is either installed or "DNI" on this schematic to tell the Teensy code to operate in I²C or SPI mode.

For more information regarding the Teensy microcontroller, visit PJRC at the following link: https://www.pjrc.com/teensy/card7a_rev1.pdf.

For more information regarding the Arduino software environment, visit the Arduino homepage at the following link: https://www.arduino.cc.

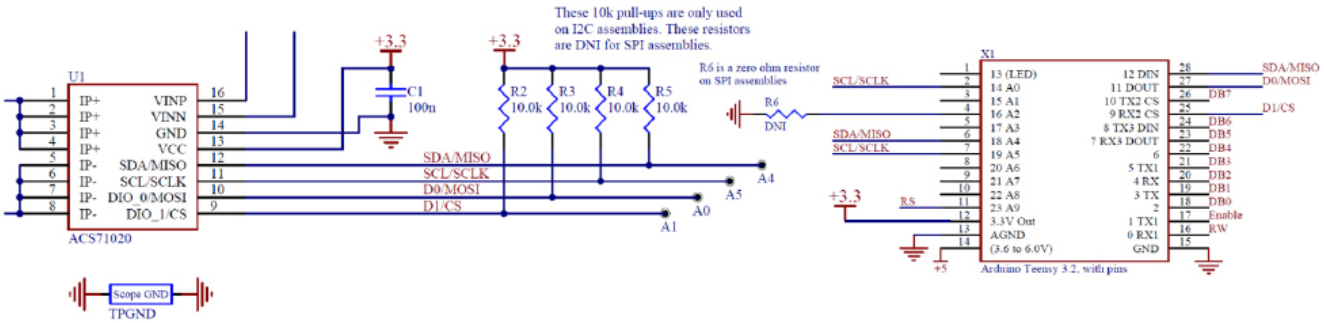

Figure 17: Application Schematic for Power Monitoring IC and the Teensy 3.2 microcontroller

APPENDIX B: FULL ARDUINO SOURCE CODE FOR ACS71020 AND TEENSY 3.2

The snippet below shows full Arduino source code used alongside this application note.

The full ".ino" Arduino sketch is available on Allegro Microsystem's Software Portal under the ACS71020 device page. To register with Allegro's software portal and view the ACS71020 source code, visit https://registration.allegromicro.com/login.

```
/*
 *    Example source code for an Arduino to show how
 *    to use SPI and I2C to communicate with an Allegro ACS71020
 *
 *    Written by K. Robert Bate, Allegro MicroSystems, LLC.
 *
 *    ACS71020_SPI_Example is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 */
#include <SPI.h>
#include <Wire.h>

#define kNOERROR 0
#define kREADERROR 1
#define kWRITEERROR 2

const uint16_t SPIChipSelectPin = 23;
const uint16_t I2CAddress = 127;
const uint16_t ProtocolSelectPin = 16;

const uint32_t WRITE = 0x00;
const uint32_t READ = 0x80;
const uint32_t COMMAND_MASK = 0x80;
const uint32_t ADDRESS_MASK = 0x7F;

unsigned long nextTime;
bool ledOn = false;

bool UseI2C = false;

// Setup the demo board.
void setup()
{
        // Initialize serial
    Serial.begin(115200);

        // Turn on the pullup so the determination of communication protocol can be made.
    pinMode(ProtocolSelectPin, INPUT_PULLUP);

    delay(50); // Wait for the pullup to take affect

    UseI2C = (digitalRead(ProtocolSelectPin) == HIGH);

    if (UseI2C)
    {
        // Initialize I2C
        Wire.begin();
        Wire.setClock(400000);
    }
    else
    {
        // Initialize SPI
        SPI.begin();
```

```
        // Setup chip select
        pinMode(SPIChipSelectPin, OUTPUT);
        digitalWrite(SPIChipSelectPin, HIGH);

        SPI.setSCK(14);

        // Make sure all of the SPI pins are
        // ready by doing a read
        uint32_t unused;
        Read(0x0, unused);
    }

    Write(0x2F, 0x4F70656E);   // Unlock device

    // If the Arduino has built in USB, keep the next line
    // in to wait for the Serial to initialize
    while (!Serial);

    if (UseI2C)
    {
        Serial.println("Using I2C version of ACS71020");
    }
    else
    {
        Serial.println("Using SPI version of ACS71020");
    }

        pinMode(LEDPin, OUTPUT);
    digitalWrite(LEDPin, LOW);
    nextTime = millis();
}

/*
 * Every 500 milliseconds, read the ACS71020 and print out the values
 */
void loop()
{
    uint32_t vrms_irms;
    uint32_t vrms;
    uint32_t irms;
    uint32_t pactive;
    uint32_t paparent;
    uint32_t pimag;
    uint32_t pfactor;
    uint32_t numptsout;
    uint32_t vrmsavgonesec_irmsavgonesec;
    uint32_t vrmsavgonesec;
    uint32_t irmsavgonesec;
    uint32_t vrmsavgonemin_irmsavgonemin;
    uint32_t vrmsavgonemin;
    uint32_t irmsavgonemin;
    uint32_t pactavgonesec;
    uint32_t pactavgonemin;
    uint32_t vcodes;
    uint32_t icodes;
    uint32_t pinstant;
    uint32_t flags;

    // Every 1/2 second, toggle the state of the LED and read the ACS71020
    if (nextTime < millis())
    {
        Read(0x20, vrms_irms);
        Read(0x21, pactive);
```

```
Read(0x22, paparent);
Read(0x23, pimag);
Read(0x24, pfactor);
Read(0x25, numptsout);
Read(0x26, vrmsavgonesec_irmsavgonesec);
Read(0x27, vrmsavgonemin_irmsavgonemin);
Read(0x28, pactavgonesec);
Read(0x29, pactavgonemin);
Read(0x2A, vcodes);
Read(0x2B, icodes);
Read(0x2C, pinstant);
Read(0x2D, flags);

vrms = vrms_irms & 0x7FFF;
Serial.printf("vrms = %ul\n", vrms);
irms = (vrms_irms >> 16) & 0x7FFF;
Serial.printf("irms = %ul\n", irms);
pactive = pactive & 0x1FFFF;
Serial.printf("pactive = %dl\n", pactive);
paparent = paparent & 0xFFFF;
Serial.printf("paparent = %ul\n", paparent);
pimag = pimag & 0x1FFFF;
Serial.printf("pimag = %ul\n", pimag);
pfactor = pfactor & 0x7FF;
Serial.printf("pfactor = %dl\n", pfactor);
numptsout = numptsout & 0x1FF;
Serial.printf("numptsout = %ul\n", numptsout);
vrmsavgonesec = vrmsavgonesec_irmsavgonesec & 0x7FFF;
Serial.printf("vrmsavgonesec = %ul\n", vrmsavgonesec);
irmsavgonesec = (vrmsavgonesec_irmsavgonesec >> 16) & 0x7FFF;
Serial.printf("irmsavgonesec = %ul\n", irmsavgonesec);
vrmsavgonemin = vrmsavgonemin_irmsavgonemin & 0x7FFF;
Serial.printf("vrmsavgonemin = %ul\n", vrmsavgonemin);
irmsavgonemin = (vrmsavgonemin_irmsavgonemin >> 16) & 0x7FFF;
Serial.printf("irmsavgonemin = %ul\n", irmsavgonemin);
pactavgonesec = pactavgonesec & 0x1FFFF;
Serial.printf("pactavgonesec = %ul\n", pactavgonesec);
pactavgonemin = pactavgonemin & 0x1FFFF;
Serial.printf("pactavgonemin = %ul\n", pactavgonemin);
vcodes = vcodes & 0x1FFFF;
Serial.printf("vcodes = %ul\n", vcodes);
icodes = icodes & 0x1FFFF;
Serial.printf("icodes = %ul\n", icodes);
Serial.printf("pinstant = %ul\n", pinstant);
Serial.print("pospf = ");
Serial.println((flags >> 6) & 0x1);
Serial.print("posangle = ");
Serial.println((flags >> 5) & 0x1);
Serial.print("undervoltage = ");
Serial.println((flags >> 4) & 0x1);
Serial.print("overvoltage = ");
Serial.println((flags >> 3) & 0x1);
Serial.print("faultlatched = ");
Serial.println((flags >> 2) & 0x1);
Serial.print("faultout = ");
Serial.println((flags >> 1) & 0x1);
Serial.print("vzerocrossout = ");
Serial.println((flags >> 0) & 0x1);
Serial.println();

        if (ledOn)
{
    digitalWrite(LEDPin, LOW);
```

```
            ledOn = false;
        }
        else
        {
            digitalWrite(LEDPin, HIGH);
            ledOn = true;
        }

        nextTime = millis() + 500L;
    }
}

/*
 * Read a register
 *
 *    address   - the address to be written
 *    value     - the value that was read
 *    returns   - the error (0 otherwise)
 */
uint16_t Read(uint8_t address, uint32_t& value)
{
    uint16_t results = kNOERROR;

    if (UseI2C)
    {
        Wire.beginTransmission(I2CAddress);
        Wire.write(address);
        results = Wire.endTransmission();

        if (results == kNOERROR)
        {
            Wire.requestFrom(I2CAddress, 4u);

            value = Wire.read();    // receive a byte as character
            value |= Wire.read() << 8;    // receive a byte as character
            value |= Wire.read() << 16;    // receive a byte as character
            value |= Wire.read() << 24;    // receive a byte as character
        }
    }
    else
    {
        SPI.beginTransaction(SPISettings(1000000, LSBFIRST, SPI_MODE3));

        // Combine the register address and the command into one byte
        uint8_t command = (address & ADDRESS_MASK) | READ;

        // take the chip select low to select the device
        digitalWrite(SPIChipSelectPin, LOW);

        // send the device the register you want to read
        SPI.transfer(command);
        SPI.transfer(0);
        SPI.transfer(0);
        SPI.transfer(0);
        SPI.transfer(0);

        digitalWrite(SPIChipSelectPin, HIGH);
        delayMicroseconds(4);
        digitalWrite(SPIChipSelectPin, LOW);

        // send the command again to read the contents
        SPI.transfer(command);
        value = (uint32_t)SPI.transfer(0);
```

```
        value |= (uint32_t)SPI.transfer(0) << 8;
        value |= (uint32_t)SPI.transfer(0) << 16;
        value |= (uint32_t)SPI.transfer(0) << 24; // high byte

        // take the chip select high to de-select
        digitalWrite(SPIChipSelectPin, HIGH);

        SPI.endTransaction();
    }

    return results;
}

/*
 * Write a register
 *
 *    address   - the address to be written
 *    value     - the value to be written
 *    returns   - the error (0 otherwise)
 */
uint16_t Write(uint8_t address, uint32_t value)
{
    uint16_t results = kNOERROR;

    if (UseI2C)
    {
        Wire.beginTransmission(I2CAddress);
                // Send the address then the value (least significant byte first)
        Wire.write(address);
        Wire.write(value);
        Wire.write(value >> 8);
        Wire.write(value >> 16);
        Wire.write(value >> 24);
        results = Wire.endTransmission();
    }
    else
    {
        SPI.beginTransaction(SPISettings(1000000, LSBFIRST, SPI_MODE3));

        // Combine the register address and the command into one byte
        uint8_t command = ((address & ADDRESS_MASK) | WRITE);

        // take the chip select low to select the device:
        digitalWrite(SPIChipSelectPin, LOW);

                // Send the command then the value (least significant byte first)
        SPI.transfer(command);
        SPI.transfer((uint8_t)value);
        SPI.transfer((uint8_t)(value >> 8));
        SPI.transfer((uint8_t)(value >> 16));
        SPI.transfer((uint8_t)(value >> 24));

        // take the chip select high to de-select:
        digitalWrite(SPIChipSelectPin, HIGH);

        SPI.endTransaction();
    }
    if (address < 0x10)
    {
        delay(30);  // If writing to EEPROM delay 30 ms
    }

    return results;
}
```

Revision History

Number	Date	Description	Responsibility
–	April 8, 2020	Initial release	W. Bussing
1	January 6, 2021	Added ACS37800 part number	K. Hampton

955 PERIMETER ROAD • MANCHESTER, NH 03103 • USA
+1-603-626-2300 • FAX: +1-603-641-5336 • ALLEGROMICRO.COM