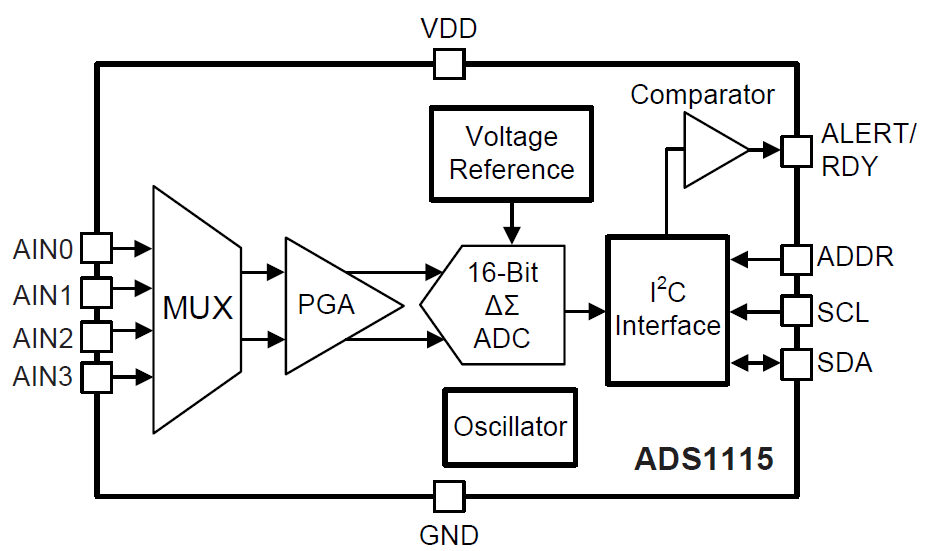
ADS1115 16Bit



1. **ADS1115()**

Basic constructor.

1. **ADS1115(**DEVICE\_ADDRESS newValue**)**

Constructor, with I2C bus address enum input.

enum DEVICE\_ADDRESS

{

ADDR\_GND = 0x48,

ADDR\_VCC = 0x49,

ADDR\_SDA = 0x4A,

ADDR\_SCL = 0x4B

};

1. **ADS1115(**DEVICE\_ADDRESS newValue, uint8\_t newAlertPin**)**

Constructor with I2C bus address enum input and Alert pin input.

enum DEVICE\_ADDRESS

{

ADDR\_GND = 0x48,

ADDR\_VCC = 0x49,

ADDR\_SDA = 0x4A,

ADDR\_SCL = 0x4B

};

1. void **SetDeviceAddress(**DEVICE\_ADDRESS newValue**)**

Member, method.

Re-Sets I2C bus address.

enum DEVICE\_ADDRESS

{

ADDR\_GND = 0x48,

ADDR\_VCC = 0x49,

ADDR\_SDA = 0x4A,

ADDR\_SCL = 0x4B

};

1. DEVICE\_ADDRESS **GetDeviceAddress()**

Member, function.

Reads current I2C bus address

enum DEVICE\_ADDRESS

{

ADDR\_GND = 0x48,

ADDR\_VCC = 0x49,

ADDR\_SDA = 0x4A,

ADDR\_SCL = 0x4B

};

1. void **SetConfig\_OS(**uint8\_t newValue, bool updateNow**)**

Sets or Clears the OS bit field in the ADS1115 configuration register.

**newValue:** Valid values = [0, 1]

Bit 15 OS Operational status or single-shot conversion start

WR = H: Start a single conversion (when in power-down state)

WR = L: No effect

RD = L: Device is currently performing a conversion

RD = H: Device is not currently performing a conversion

**updateNow:**

**False = used to set/clr bit value without updating the ADS1115's internal register.**

**True = used to set/clr bit value and then writes value to the ADS1115's internal register.**

Note-1:

The **updateNow** flag is used to limit the amount of traffic on the I2C communications bus.

If changing more than one field in the Configuration register, you set all the bits fields with the **updateNow** set to false except for the last field to be set, you pass a True so that the new configuration register value will be updated just once.

1. uint8\_t **GetConfig\_OS(**bool updateNow**)**

Reads OS bit field (bit 15) in the Configuration register.

Return value range is [0,1]

Note-2:

The **updateNow** flag is used to limit the amount of traffic on the I2C communications bus.

If reading more than one field from the Configuration register, reading the first field with

**updateNow** = **True,** and the fallowing fields with **updateNow** = false, this will update the internal variable that holds the configuration register value once from the ADS1115 then the fallowing field reads just reports the field values based on the internal variable's value.

1. void **SetConfig\_Mux(**MUX newValue, bool updateNow**)**

Type description here!

1. MUX **GetConfig\_Mux(**bool updateNow**)**

Type description here!

1. void **SetConfig\_SingleEndedMux(**uint8\_t newValue, bool updateNow**)**

Type description here!

1. uint8\_t **GetConfig\_SingleEndedMux(**bool updateNow**)**

Type description here!

1. void **SetConfig\_DifferentialMux(**uint8\_t newValue, bool updateNow**)**

Type description here!

1. uint8\_t **GetConfig\_DifferentialMux(**bool updateNow**)**

Type description here!

1. void **SetConfig\_PGA(**uint8\_t newValue, bool updateNow**)**

Type description here!

1. uint8\_t **GetConfig\_PGA(**bool updateNow**)**

Type description here!

1. void **SetConfig\_Mode(**uint8\_t newValue, bool updateNow**)**

Type description here!

1. uint8\_t **GetConfig\_Mode(**bool updateNow**)**

Type description here!

1. void **SetConfig\_Rate(**uint8\_t newValue, bool updateNow**)**

Type description here!

1. uint8\_t **GetConfig\_Rate(**bool updateNow**)**

Type description here!

1. void **SetConfig\_COMP\_MODE**(uint8\_t newValue, bool updateNow)

Type description here!

1. uint8\_t **GetConfig\_COMP\_MODE**(bool updateNow)

Type description here!

1. void **SetConfig\_COMP\_POL**(uint8\_t newValue, bool updateNow)

Type description here!

1. uint8\_t **GetConfig\_COMP\_POL**(bool updateNow)

Type description here!

24) void **SetConfig\_COMP\_LAT**(uint8\_t newValue, bool updateNow)

Type description here!

1. uint8\_t **GetConfig\_COMP\_LAT**(bool updateNow)

Type description here!

1. void **SetConfig\_COMP\_QUE**(uint8\_t newValue, bool updateNow)

Type description here!

1. uint8\_t **GetConfig\_COMP\_QUE**(bool updateNow)

Type description here!

1. uint16\_t **ReadConversionReg**()

Type description here!

1. float **ReadConversionRegFloat**()

Type description here!

1. float **ReadConversionScalled**()

What is cool about the class object is that it is easy to use all the functionality the ADS1115 has to offer.

For example: The object can easily be told to use a deferent gain settings for each ADC channel. The ADS1115 has 6 gain settings via

a bit field within the configuration register. Also, the object has scaling factors for each channel. So it's easy to rescale a channel

to return a 4-20mA reading, instead of a 1-5 volt reading. This is also used to calibrate each channel.

Note: The ADS1115 does not have registers to hold the scaling factors (slope & offset).

These scaling factors are held in arrays within the class object and are applied to the returned ADC count register value.

1. void **WriteConfigReg**(uint16\_t newRegValue)

Type description here!

1. uint16\_t **ReadConfigReg**()

Type description here!

1. void **WriteCompLoThreshReg**(uint16\_t newRegValue)

Type description here!

1. uint16\_t **ReadCompLoThreshReg**()

Type description here!

34) void **WritCompHiThreshReg**(uint16\_t newRegValue)

Type description here!

1. uint16\_t **ReadCompHiThreshReg**()

Type description here!

1. void **SetRegister**(uint8\_t reg, uint16\_t newVal)

Type description here!

1. uint16\_t **GetRegister**(uint8\_t reg)

Type description here!

1. void **SetCalSlope**(uint8\_t chnl, float newSlope)

Type description here!

1. float **GetCalSlope**(uint8\_t chnl)

Type description here!

1. void **SetCalOffset**(uint8\_t chnl, float newOffset)

Type description here!

1. float **GetCalOffset**(uint8\_t chnl)

Type description here!

1. void **SetChannelPGA**(uint8\_t chnl, uint8\_t newPGA)

Type description here!

1. uint8\_t **GetChannelPGA**(uint8\_t chnl)

Type description here!

1. void **EndWire**()

Type description here!

1. float **Convert16bit2sComplToFloat**(uint16\_t **regValue**, float **ABS\_FSR**)

Converts the 16bit input value to a float value.

The returned float value is scaled to **ABS\_FSR** input value.

1. uint16\_t **ConvertFloatTo2sCompl16Bit**(float **inpVal**, float **ABS\_FSR**)

Converts the float input value to a 2's complement 16bit value.

The returned 16bit value is scaled to **ABS\_FSR** input value.

/\*

    DATE: 01/04/2022

    TIME: 14:22

   TITLE: ADS1115 Driver (ADS1115 HAL)

  AUTHOR: Jerome B. Ford

    NOTE: Still under development. as of 1/7/2022

 \* REF:

 \* https://www.ti.com/lit/ds/symlink/ads1113.pdf?HQS=dis-dk-null-digikeymode-dsf-pf-null-wwe&ts=1641161543112&ref\_url=https%253A%252F%252Fwww.ti.com%252Fgeneral%252Fdocs%252Fsuppproductinfo.tsp%253FdistId%253D10%2526gotoUrl%253Dhttps%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Fads1113

 \*

  ADS115 DATA SHEET INFO:

  DEVICE ADDRESSING

    ADDR\_GND =  0x48;   // 1001000  Default

    ADDR\_VCC =  0x49;   // 1001001

    ADDR\_SDA =  0x4A;   // 1001010

    ADDR\_SCL =  0x4B;   // 1001011

  Address Pointer Register

    BIT   NAME        DESCRIPTION

      7   ---         Reserved 0h Always write 0h

      6   ---         Reserved 0h Always write 0h

      5   ---         Reserved 0h Always write 0h

      4   ---         Reserved 0h Always write 0h

      3   ---         Reserved 0h Always write 0h

      2   ---         Reserved 0h Always write 0h

      1               00 : Conversion register  (16bits)

      0               01 : Config register      (16bits)

                      10 : Lo\_thresh register   (16bits)

                      11 : Hi\_thresh register   (16bits)

  //---------------------------------------------------------------------------------------------//

  Config Register (P[1:0] = 1h) [reset = 8583h]

    BIT   NAME        DESCRIPTION

     15   OS          Operational status or single-shot conversion start

                      WR=H: Start a single conversion (when in power-down state)

                      WR=L: No effect

                      RD=L: Device is currently performing a conversion

                      RD=H: Device is not currently performing a conversion

     14   MUX[2]      Input multiplexer[0-2] configuration (ADS1115 only)

     13   MUX[1]      000 : AINP = AIN0 and AINN = AIN1 (default)

     12   MUX[0]      001 : AINP = AIN0 and AINN = AIN3

                      010 : AINP = AIN1 and AINN = AIN3

                      011 : AINP = AIN2 and AINN = AIN3

                      100 : AINP = AIN0 and AINN = GND

                      101 : AINP = AIN1 and AINN = GND

                      110 : AINP = AIN2 and AINN = GND

                      111 : AINP = AIN3 and AINN = GND

     11   PGA[2]      Programmable gain amplifier configuration

     10   PGA[1]      000 : FSR = ±6.144 V (1)

      9   PGA[0]      001 : FSR = ±4.096 V (1)

                      010 : FSR = ±2.048 V (default)

                      011 : FSR = ±1.024 V

                      100 : FSR = ±0.512 V

                      101 : FSR = ±0.256 V

                      110 : FSR = ±0.256 V

                      111 : FSR = ±0.256 V

      8   MODE        Device operating mode

                      WR=L : Continuous-conversion mode

                      WR=H : Single-shot mode or power-down state (default)

      7   DR[2]       Data rate

      6   DR[1]       000 : 8 SPS

      5   DR[0]       001 : 16 SPS

                      010 : 32 SPS

                      011 : 64 SPS

                      100 : 128 SPS (default)

                      101 : 250 SPS

                      110 : 475 SPS

                      111 : 860 SPS

      4   COMP\_MODE   Comparator mode (ADS1114 and ADS1115 only)

                      0 : Traditional comparator (default)

                      1 : Window comparator

      3   COMP\_POL    Comparator polarity (ADS1114 and ADS1115 only)

                      0 : Active low (default)

                      1 : Active high

      2   COMP\_LAT    Latching comparator (ADS1114 and ADS1115 only)

                      0 : Nonlatching comparator . The ALERT/RDY pin does not latch when asserted (default).

                      1 : Latching comparator. The asserted ALERT/RDY pin remains latched until

                          conversion data are read by the master or an appropriate SMBus alert response

                          is sent by the master. The device responds with its address, and it is the lowest

                          address currently asserting the ALERT/RDY bus line.

      1   COMP\_QUE[1] Comparator queue and disable (ADS1114 and ADS1115 only)

      0   COMP\_QUE[0] 00 : Assert after one conversion

                      01 : Assert after two conversions

                      10 : Assert after four conversions

                      11 : Disable comparator and set ALERT/RDY pin to high-impedance (default)

\*/

#ifndef ADS1115\_h

#define ADS1115\_h

#include "Wire.h"

#include "Arduino.h"

class ADS1115

{

  public:

    enum DEVICE\_ADDRESS

    {

        ADDR\_GND = 0x48,

        ADDR\_VCC = 0x49,

        ADDR\_SDA = 0x4A,

        ADDR\_SCL = 0x4B

    };

    enum PGA

    {

       FSR0\_6144mV = 0,

       FSR1\_4096mV = 1,

       FSR2\_2048mV = 2,

       FSR3\_1024mV = 3,

       FSR4\_0512mV = 4,

       FSR5\_0256mV = 5,

       FSR6\_0256mV = 6,

       FSR7\_0256mV = 7,

    };

    enum MUX

    {

       CHNL0\_P0\_N1 = 0,

       CHNL1\_P0\_N3 = 1,

       CHNL2\_P1\_N3 = 2,

       CHNL3\_P2\_N3 = 3,

       CHNL4\_SE0 = 4,

       CHNL5\_SE1 = 5,

       CHNL6\_SE2 = 6,

       CHNL7\_SE3 = 7,

    };

    enum RATE

    {

        SPS0\_8 = 0,

        SPS1\_16 = 1,

        SPS2\_32 = 2,

        SPS3\_64 = 3,

        SPS4\_128 = 4,

        SPS5\_250 = 5,

        SPS6\_475 = 6,

        SPS7\_860 = 7,

    };

    const uint8\_t RegAddress\_Conversion   = 0;

    const uint8\_t RegAddress\_Config       = 1;

    const uint8\_t RegAddress\_CompLoThresh = 2;

    const uint8\_t RegAddress\_CompHiThresh = 3;

    //-------------------------------------------------------------------------------//

    ADS1115()

    {

      Wire.begin(); // Join i2c bus as master.

    }

    //-------------------------------------------------------------------------------//

    ADS1115(DEVICE\_ADDRESS newValue)

    {

      \_deviceAddress = newValue;

      Wire.begin(); // Join i2c bus as master.

    }

    //-------------------------------------------------------------------------------//

    ADS1115(DEVICE\_ADDRESS newValue, uint8\_t newAlertPin)

    {

      \_deviceAddress = newValue;

      \_AlertPin = newAlertPin;

      pinMode(\_AlertPin, INPUT);

      Wire.begin(); // Join i2c bus as master.

    }

    //-------------------------------------------------------------------------------//

    ~ADS1115()

    {

      EndWire();

    }

    //-------------------------------------------------------------------------------//

    void SetDeviceAddress(DEVICE\_ADDRESS newValue)

    {

      \_deviceAddress = newValue;

    }

    //-------------------------------------------------------------------------------//

    DEVICE\_ADDRESS GetDeviceAddress()

    {

      return \_deviceAddress;

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_OS(uint8\_t newValue, bool updateNow)

    {

      /\*

        15   OS       Operational status or single-shot conversion start

                      WR=H: Start a single conversion (when in power-down state)

                      WR=L: No effect

                      RD=L: Device is currently performing a conversion

                      RD=H: Device is not currently performing a conversion

      \*/

      newValue &= 1;    // force valid value = [0 - 1]

      \_RegValue\_Config &= ~(1 << 15);

      \_RegValue\_Config |= (newValue << 15);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_OS(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 15) & 1);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_Mux(uint8\_t newValue, bool updateNow)

    {

      /\*

        Input multiplexer[0-2] configuration (ADS1115 only)

        14   MUX[2]

        13   MUX[1]

        12   MUX[0]

            000 : AINP = AIN0 and AINN = AIN1, Differential, (default)

            001 : AINP = AIN0 and AINN = AIN3, Differential

            010 : AINP = AIN1 and AINN = AIN3, Differential

            011 : AINP = AIN2 and AINN = AIN3, Differential

            100 : AINP = AIN0 and AINN =  GND, Single Ended

            101 : AINP = AIN1 and AINN =  GND, Single Ended

            110 : AINP = AIN2 and AINN =  GND, Single Ended

            111 : AINP = AIN3 and AINN =  GND, Single Ended

      \*/

        //1111110000000000

        //5432109876543210

        //0000000000000111 =     7dec, 0x0007

        //0111000000000000 = 28672dec, 0x7000

        //1000111111111111 = 36863dec, 0x8FFF

      newValue &= 7;// force valid value = [0 - 7]

      \_RegValue\_Config &= ~(7 << 12);

      \_RegValue\_Config |= (newValue << 12);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_Mux(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 12) & 7);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_SingleEndedMux(uint8\_t newValue, bool updateNow)

    {

      newValue &= 3;

      SetConfig\_PGA( \_gain[4 + newValue], false);

      SetConfig\_Mux(4 + newValue, updateNow);

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_SingleEndedMux(bool updateNow)

    {

      return GetConfig\_Mux(updateNow) & 3;

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_DifferentialMux(uint8\_t newValue, bool updateNow)

    {

      newValue &= 3;

      SetConfig\_Mux(newValue, updateNow);

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_DifferentialMux(bool updateNow)

    {

      return GetConfig\_Mux(updateNow) & 3;

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_PGA(uint8\_t newValue, bool updateNow)

    {

        /\*

            11  PGA[2]  Programmable gain amplifier configuration

            10  PGA[1]  000 : FSR = ±6.144 V (1)

            9   PGA[0]  001 : FSR = ±4.096 V (1)

                        010 : FSR = ±2.048 V (default)

                        011 : FSR = ±1.024 V

                        100 : FSR = ±0.512 V

                        101 : FSR = ±0.256 V

                        110 : FSR = ±0.256 V

                        111 : FSR = ±0.256 V

            1111110000000000

            5432109876543210

            0000000000000111 =     7dec, 0x0007

            0000111000000000 =  3584dec, 0x0E00  = (7<<9)

            1111000111111111 = 61951dec, 0xF1FF

        \*/

      newValue &= 7;  // force valid value = [0 - 7]

      \_RegValue\_Config &= ~(7 << 9);

      \_RegValue\_Config |= (newValue << 9);

      //Serial.println("1) PGA = " + String(newValue, DEC));

      if(updateNow == true)

      {

        //Serial.println("2) PGA = " + String(newValue, DEC));

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_PGA(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 9) & 7);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_Mode(uint8\_t newValue, bool updateNow)

    {

        /\*

            bit 8   MODE    Device operating mode

                            WR=L : Continuous-conversion mode

                            WR=H : Single-shot mode or power-down state (default)

        \*/

      newValue &= 1;    // force valid value = [0 - 1]

      \_RegValue\_Config &= ~(1 << 8);

      \_RegValue\_Config |= (newValue << 8);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_Mode(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 8) & 1);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_Rate(uint8\_t newValue, bool updateNow)

    {

     /\*

        Data rate

        Bit 7   DR[2]

        Bit 6   DR[1]

        Bit 5   DR[0]

            000 :   8 SPS

            001 :  16 SPS

            010 :  32 SPS

            011 :  64 SPS

            100 : 128 SPS (default)

            101 : 250 SPS

            110 : 475 SPS

            111 : 860 SPS

        1111110000000000

        5432109876543210

        0000000000000111 =     7dec, 0x0007

        0000000011100000 =   224dec, 0x00e0

        1111111100011111 = 65311dec, 0xF1FF

     \*/

      newValue &= 7;    // Force valid value = [0 - 7]

      \_RegValue\_Config &= ~(7 << 5);

      \_RegValue\_Config |= (newValue << 5);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_Rate(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 5) & 7);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_COMP\_MODE(uint8\_t newValue, bool updateNow)

    {

      /\*

        Bit 4   COMP\_MODE   Comparator mode (ADS1114 and ADS1115 only)

                      0 : Traditional comparator (default)

                      1 : Window comparator

      \*/

      newValue &= 1;// force valid value = [0 - 1]

      \_RegValue\_Config &= ~(1 << 4);

      \_RegValue\_Config |= (newValue << 4);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_COMP\_MODE(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 4) & 1);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_COMP\_POL(uint8\_t newValue, bool updateNow)

    {

        /\*

            Bit 3   COMP\_POL    Comparator polarity (ADS1114 and ADS1115 only)

                                0 : Active low (default)

                                1 : Active high

        \*/

      newValue &= 1;  // force valid value = [0 - 1]

      \_RegValue\_Config &= ~(1 << 3);

      \_RegValue\_Config |= (newValue << 3);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_COMP\_POL(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 3) & 1);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_COMP\_LAT(uint8\_t newValue, bool updateNow)

    {

      /\*

        Bit 2   COMP\_LAT    Latching comparator (ADS1114 and ADS1115 only)

                            0 : Nonlatching comparator . The ALERT/RDY pin does not latch when asserted (default).

                            1 : Latching comparator. The asserted ALERT/RDY pin remains latched until

                                conversion data are read by the master or an appropriate SMBus alert response

                                is sent by the master. The device responds with its address, and it is the lowest

                                address currently asserting the ALERT/RDY bus line.

      \*/

      newValue &= 1;// force valid value = [0 - 1]

      \_RegValue\_Config &= ~(1 << 2);

      \_RegValue\_Config |= (newValue << 2);

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_COMP\_LAT(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return ((\_RegValue\_Config >> 2) & 1);

    }

    //-------------------------------------------------------------------------------//

    void SetConfig\_COMP\_QUE(uint8\_t newValue, bool updateNow)

    {

      /\*

        bit

          1

          0

          COMP\_QUE[1] Comparator queue and disable (ADS1114 and ADS1115 only)

          COMP\_QUE[0] 00 : Assert after one conversion

                      01 : Assert after two conversions

                      10 : Assert after four conversions

                      11 : Disable comparator and set ALERT/RDY pin to high-impedance (default)

      \*/

      newValue &= 3;                //Force valid values [0 - 3]

      \_RegValue\_Config &= ~(3);     //CLR bits 0 & 1.

      \_RegValue\_Config |= newValue; //SET bits 0 & 1.

      if(updateNow == true)

      {

        WriteConfigReg(\_RegValue\_Config);

      }

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetConfig\_COMP\_QUE(bool updateNow)

    {

      if(updateNow == true)

      {

        ReadConfigReg();

      }

      return (\_RegValue\_Config & 3);  // mask return bits 0 & 1

    }

    //-------------------------------------------------------------------------------//

    uint16\_t ReadConversionReg()

    {

      \_RegValue\_Conversion = GetRegister(RegAddress\_Conversion);

      return \_RegValue\_Conversion;

    }

    //-------------------------------------------------------------------------------//

    float ReadConversionRegFloat()

    {

        uint8\_t pga\_val = GetConfig\_PGA(false); // <= returns [0 - 7]

        \_RegValue\_Conversion = GetRegister(RegAddress\_Conversion);

        return Convert16bit2sComplToFloat(\_RegValue\_Conversion, \_fsr[pga\_val]);

    }

    //-------------------------------------------------------------------------------//

    float ReadConversionScalled()

    {

        uint8\_t mux\_val = GetConfig\_Mux(false); // <= returns [0 - 7]

        float Vchnl = ReadConversionRegFloat();

        Vchnl \*= \_calSlope[mux\_val];

        Vchnl += \_calOffset[mux\_val];

        return Vchnl;

    }

    //-------------------------------------------------------------------------------//

    void WriteConfigReg(uint16\_t newRegValue)

    {

      \_RegValue\_Config = newRegValue;

      SetRegister(RegAddress\_Config, \_RegValue\_Config);

    }

    //-------------------------------------------------------------------------------//

    uint16\_t ReadConfigReg()

    {

      \_RegValue\_Config = GetRegister(RegAddress\_Config);

      return \_RegValue\_Config;

    }

    //-------------------------------------------------------------------------------//

    void WriteCompLoThreshReg(uint16\_t newRegValue)

    {

      \_RegValue\_CompLoThresh = newRegValue;

      SetRegister(RegAddress\_CompLoThresh, \_RegValue\_CompLoThresh);

    }

    //-------------------------------------------------------------------------------//

    uint16\_t ReadCompLoThreshReg()

    {

      \_RegValue\_CompLoThresh = GetRegister(RegAddress\_CompLoThresh);

      return \_RegValue\_CompLoThresh;

    }

    //-------------------------------------------------------------------------------//

    void WritCompHiThreshReg(uint16\_t newRegValue)

    {

      \_RegValue\_CompHiThresh = newRegValue;

      SetRegister(RegAddress\_CompHiThresh, \_RegValue\_CompHiThresh);

    }

    //-------------------------------------------------------------------------------//

    uint16\_t ReadCompHiThreshReg()

    {

      \_RegValue\_CompHiThresh = GetRegister(RegAddress\_CompHiThresh);

      return \_RegValue\_CompHiThresh;

    }

    //-------------------------------------------------------------------------------//

    void SetRegister(uint8\_t reg, uint16\_t newVal)

    {

        ByteField16 thevalues;

        thevalues.valUint16 = newVal;

        Wire.beginTransmission(\_deviceAddress);

        Wire.write(reg);

        Wire.write(thevalues.bytes[1]);

        Wire.write(thevalues.bytes[0]);

        Wire.endTransmission();

    }

    //-------------------------------------------------------------------------------//

    uint16\_t GetRegister(uint8\_t reg)

    {

      Wire.beginTransmission(\_deviceAddress);

      Wire.write(reg);

      Wire.endTransmission();

      ByteField16 thevalues;

      Wire.requestFrom(\_deviceAddress, 2);

      if(Wire.available() <= 2)

      {

        thevalues.bytes[1] = Wire.read();

        thevalues.bytes[0] = Wire.read();

      }

      return thevalues.valUint16;

    }

    //-------------------------------------------------------------------------------//

    void SetCalSlope(uint8\_t chnl, float newSlope)

    {

      chnl &= 7;

      \_calSlope[chnl] = newSlope;

    }

    //-------------------------------------------------------------------------------//

    float GetCalSlope(uint8\_t chnl)

    {

      chnl &= 7;

      return \_calSlope[chnl];

    }

    //-------------------------------------------------------------------------------//

    void SetCalOffset(uint8\_t chnl, float newOffset)

    {

      chnl &= 7;

      \_calOffset[chnl] = newOffset;

    }

    //-------------------------------------------------------------------------------//

    float GetCalOffset(uint8\_t chnl)

    {

      chnl &= 7;

      return \_calOffset[chnl];

    }

    //-------------------------------------------------------------------------------//

    void SetChannelPGA(uint8\_t chnl, uint8\_t newPGA)

    {

      /\*

        Programmable gain amplifier configuration

        11  Config.PGA[2]

        10  Config.PGA[1]

         9  Config.PGA[0]

        000 : FSR = ±6.144 V (1)

        001 : FSR = ±4.096 V (1)

        010 : FSR = ±2.048 V (default)

        011 : FSR = ±1.024 V

        100 : FSR = ±0.512 V

        101 : FSR = ±0.256 V

        110 : FSR = ±0.256 V

        111 : FSR = ±0.256 V

      \*/

      chnl &= 7;

      \_gain[chnl] = (newPGA & 7);

    }

    //-------------------------------------------------------------------------------//

    uint8\_t GetChannelPGA(uint8\_t chnl)

    {

      chnl &= 7;

      return \_gain[chnl];

    }

    //-------------------------------------------------------------------------------//

    void EndWire()

    {

        Wire.endTransmission(); // Not sure if this is needed!

        Wire.end();

        TWCR &= ~0x04;          // <= NOT SURE ABOUT THIS!

    }

    //-------------------------------------------------------------------------------//

    float Convert16bit2sComplToFloat(uint16\_t regValue, float ABS\_FSR)

    {

        float volts = 0.0f;

        uint16\_t adcVolts = 0;

        if(regValue >= 32768)

        {

            adcVolts = (~regValue);

            adcVolts++;

            volts = -ABS\_FSR;

            volts \*= (float)adcVolts / 32768.0f;

        }

        else

        {

            adcVolts = regValue;

            volts = ABS\_FSR;

            volts \*= (float)adcVolts / 32768.0f;

        }

        return volts;

    }

    //-------------------------------------------------------------------------------//

    uint16\_t ConvertFloatTo2sCompl16Bit(float inpVal, float ABS\_FSR)

    {

        uint16\_t regValue = 0;

        float tempFloat = 0;

        if(inpVal < 0.0f)

        {

            tempFloat = 32768.0f;

            tempFloat \*=  (-inpVal/ABS\_FSR);

            regValue = (uint16\_t)tempFloat;

            regValue = ~regValue;

            regValue++;

        }

        else

        {

            tempFloat = 32768.0f;

            tempFloat \*= (inpVal/ABS\_FSR);

            regValue = (uint16\_t)round(tempFloat);

        }

        return regValue;

    }

    //-------------------------------------------------------------------------------//

  private:

    DEVICE\_ADDRESS \_deviceAddress = ADDR\_GND;

    uint16\_t \_RegValue\_Conversion = 0;

    uint16\_t \_RegValue\_Config = 0;

    uint16\_t \_RegValue\_CompLoThresh = 0;

    uint16\_t \_RegValue\_CompHiThresh = 0;

    uint8\_t \_AlertPin = 0;

    float       \_fsr[8] = {6.144f, 4.096f, 2.048f, 1.024f, 0.512f, 0.256f, 0.256f, 0.256f}; // Full scale range. Selected by PGA.

    float  \_calSlope[8] = {1.000f, 1.000f, 1.000f, 1.000f, 1.000f, 1.000f, 1.000f, 1.000f}; // Slopes for channel. Selected by MUX.

    float \_calOffset[8] = {0.000f, 0.000f, 0.000f, 0.000f, 0.000f, 0.000f, 0.000f, 0.000f}; // Offset for channel. Selected by MUX.

    uint8\_t    \_gain[8] = {     2,      2,      2,      2,      2,      2,      2,      2}; // Channel Gain.

    union ByteField16

    {

      float valFloat;

      uint16\_t valUint16;

      int16\_t valSint16;

      uint8\_t bytes[4];

    };

};

#endif