ZigBit™ OEM Module 1.1

Application Note

Using ZigBit Module with Analog Sensors

Doc. AN-481~06 v.1.2 April 2008

Summary

This application note provides a how-to tutorial for connecting an analog sensor to a ZigBit™ (ZDM-A1281-B0 and ZDM-A1281-A2). The note details both hardware and software aspects of using the module's ADC interface to read data off the sensor, process it and send it over the air. A wiring diagram and a sample application code are supplied. Although this document describes a specific industrial pressure sensor, the recommended procedure is easily adaptable to most of other types of analog sensors and applications that involve them.

Related documents:

[1] 8-bit AVR Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash ATmega 640/V, ATmega 1280/V, ATmega 1281/V, ATmega 2560/V, ATmega 2561/V. http://www.atmel.com/dyn/resources/prod_documents/doc2549.pdf

[2] Piezoresistive Transmitters Series 21 / R / Pro for Industrial Applications. http://www.keller-druck.com/picts/pdf/engl/21e.pdf

[3] ZigBit™ Development Kit 1.3. User's Guide. MeshNetics Doc. S-ZDK-451

[4] MeshBean2 (WDB-A1281-P1). Schematics. MeshNetics Doc. P-MB2P-461~01

[5] ZigBeeNet™ Stack Documentation. MeshNetics Doc. P-ZBN-452~02

Hardware Implementation

Hardware implementation covers physical interface between the sensor and a ZigBit™ module. The particular model referenced in this application note is a Keller Piezoresistive Pressure Transmitter (PPT) Series 21 [2]. This sensor is a typical analog sensor with a `4 – 20 mA` current loop and input power of `8` to `28V`; hence the exercise of attaching it to a ZigBit™ module is easily adapted to any similar analog sensor.

For the purposes of this application note, it is assumed that a module is mounted on MeshBean2 development board (see [4]). Because the board cannot supply power above `3V`, a separate, external power supply to power the sensor is required. In a real application, developers may choose to replace an external power supply with a DC to DC converter and use the board's own power source to power on the sensor. This note assumes that an external power supply is used.

When connected to ADC converter, the analog current loop must be converted to a voltage output with a precision resistor. An additional concern is to wire the sensor such that its voltage output is in the acceptable range, specifically below the reference voltage of the module's MCU. ATmega 1281 MCU uses a reference voltage of `1.25V` (see [1]). Precision resistor was selected using the familiar Ohm's law formula: $V=IR$. Given that the maximum current, $I_{max} = 20\ mA$, and the reference voltage, $V_{ref}=1.25V$:

$$R \approx V/I = 1.25V/20mA = 62.5\ Ohm$$

Figure 1 is the wiring diagram for the sensor and the module. MeshBean2 components irrelevant to the wiring of the sensor have been omitted.

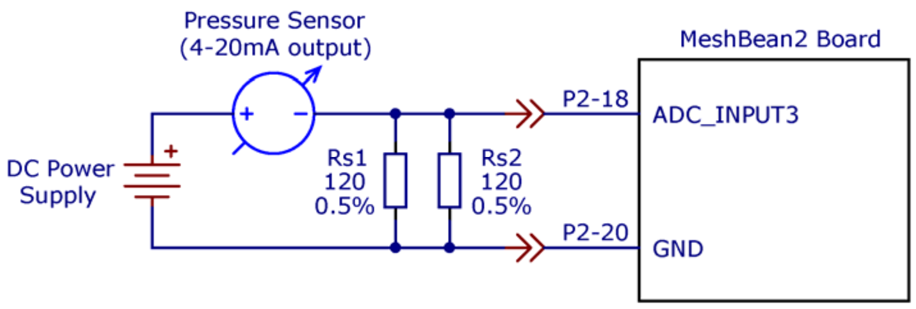

Figure 1. Module to Sensor Wiring

The final assembly is shown below in Figure 2. Power connector and the resistor have been glued together and to a connector and plugged into the extension connector installed on the MeshBean2 board. MeshBean2 schematics [4] shows the default connector pinout. Pins 18 and 19 are connected to `ADC_INPUT3` and `GND`, respectively. Note that by default there are 3 unoccupied ADC channels available on a MeshBean2, thus up to 3 independent analog sensors may be connected without any hardware modification.

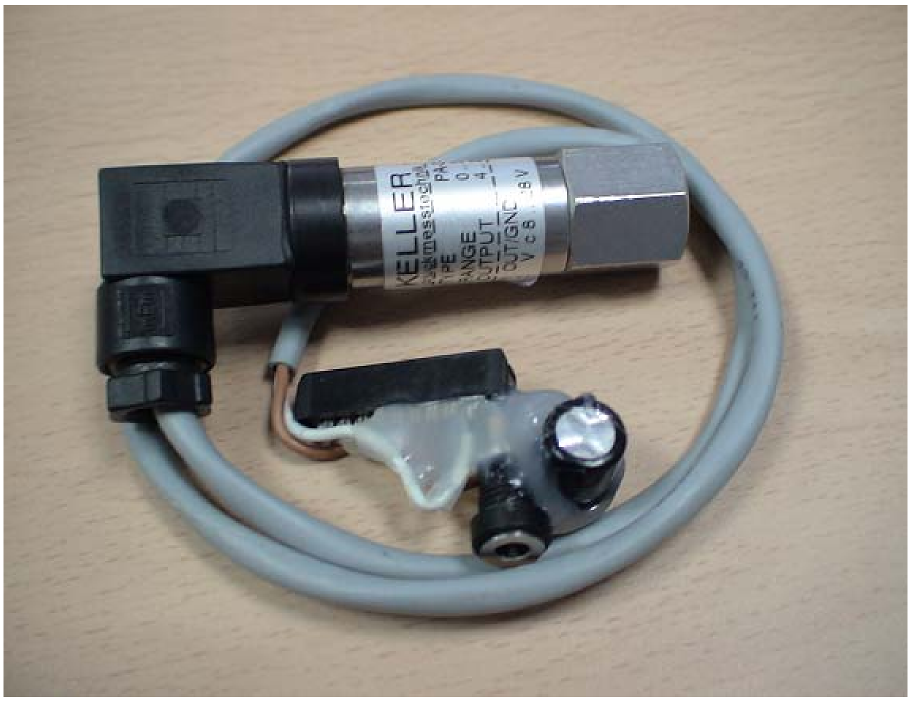

Figure 2. Sensor Assembly

Components Used

The suggested wiring requires following hardware components (in addition to the sensor itself):

- Resistor
- Power connector
- Plug
- Power supply.

Software Implementation

The end-user application accompanying the hardware assembly provides a simple template for reading data off an analog sensor, converting it to a meaningful value representing some environmental condition, and transferring said value to some higher functionality device. In ZigBee terminology, the setup corresponds roughly to an end-device (RFD) reading a physical sensor and forwarding its readings wirelessly to a coordinator device (FFD). The coordinator is simply connected to a PC, which outputs data transferred from the end device(s).

A real-world application will incorporate a more intelligent central controller performing some further analysis and/or visualization. For example, it could implement tasks like actuation of additional wireless devices, data aggregation or logging. This extended functionality is outside of the scope of this application note.

The rest of this document assumes high-level familiarity with ZigBit Development Kit. The reader is encouraged to familiarize himself with the sample applications available with ZDK [3].

ADC interface

Once the sensor is physically connected to one of ADC inputs, a driver to communicate with the sensor needs to be written. In our case, the only functionality required of the driver is to read a digital representation of the voltage supplied to the appropriate ADC pin.

ZigBit Development Kit comes with reference drivers for every interface available on the module. API provided by the reference ADC driver can be found in `adc.h`. API includes the following functions:

- `result_t adc_open(uint8_t adcNumber, void (*f)(uint16_t data));`
- result_t adc_get(uint8_t adcNumber);
- `result_t adc_close(uint8_t adcNumber);`.

An application needs only to include the header file and link in the precompiled `libZigBit.a` library to make use of the standard ADC API. Because ADC-related functionality is part of hardware abstraction layer (HAL), which is supplied in source code, developer may also inspect and/or change the driver to suit the needs. The driver itself can be found in `adc.c`.

The application source code listing is provided in Appendix A. The standard UART setup and logical address selection has been borrowed from `peer2peer` sample application (see [5] for the description of `peer2peer`). Note at the end of `fw_userEntry` function a call to `adc_open`, which opens the 3rd ADC channel for reading (but only if the device is not a coordinator, i.e. its logical address is different from `0`). Since ADC conversion is not an instantaneous operation, a call to `adc_open` also includes a callback that gets invoked whenever an ADC conversion is complete. In turn, `process_data` converts the raw value returned from the ADC converter to a floating point representation.

Atmega MCU has a 10 bit ADC converter, so the raw decimal value returned must be scaled to the maximum number expressible in 10 bit value. Hence, the raw value is divided by $2^{10}=1024$. The resulting value must then be scaled by the reference voltage, which in our case is `1.25V`. Having done this conversion, one arrives at a digital representation of the voltage input to the particular ADC line. This value needs further correlation with properties of the specific sensor.

For the particular case of Keller PPT (see [2]), which measures atmospheric pressure on a linear scale from `-1 atm` to `4 atm` with a linear scaling factor. The rest of the code is fairly straightforward. After collecting 5 samples of ADC input, the last one is taken, converted to a string representation and sent over the network. The receiving node (always the coordinator) simply outputs the string message to UART.

Developer Tips and Tricks

It is not advisable to process the first several outputs of ADC conversion as it may contain errors relating to ADC logic initialization. This is especially true if the device has just woken up. Thus first 5 - 10 samples of ADC inputs must be discarded.

MeshBean2 boards serving as end devices cannot be powered by USB connector because USB voltage levels are too unstable, even if the PC is properly grounded. A noisy power supply in turn contaminates the reference voltage used by the ADC converter rendering all values read from the sensor input unreliable. Developers are strongly encouraged to test their power sources for noise. Our experiments show that battery-powered MeshBean2 boards can produce ADC values within the tolerance declared for the Atmega 1281 MCU (+/- 2 LSB bits).

Supplementary Materials

`EndDevice.c` – source code listing for the end user application

Appendix A: Source Code Listing

```
/**
 * Copyright (c) 2005-2007 LuxLabs Ltd. dba MeshNetics, All Rights
Reserved
 **/
#include "framework.h"
#include "adc.h"
#include "leds.h"

#define END_POINT        1           // End-point for transmission.
#define DATA_HANDLE      1           // Data frame handle.

// Pins connected to leds
#define NETWORK_LED                 0   // Network indication red LED.
#define NETWORK_TRANSMISSION_LED 1  // Data transmission yellow LED.

static IEEEAddr_t macAddr = 2;       // MAC address of the node

#define LED_FLASH_DELAY 300u        //defines network led blink period
                                    //when device is searching network

static enum                         // Possible network states.
{
  NETWORK_IDLE_STATE,               // Waiting for network button press.
  NETWORK_JOINED_STATE,             // Join procedure finished.
  NETWORK_JOIN_REQUEST_STATE,        // Waiting for the first join event
} networkState = NETWORK_IDLE_STATE; // Network default state.

static enum
{
  SENSOR_OFF,
  SENSOR_READY,
  SENSOR_READ
} sensorState = SENSOR_OFF;

// Application-specific stuff
short sensorUp = FALSE;
short sample = 0;
float sensorValue = 0.0;
void processData(uint16_t);
static uint8_t temp_buffer[32];

// Functions' prototypes.
void RegisterNetworkEvents(void);
void SetNetworkParameters(void);
void SetPowerParameters(void);
void mainLoop();                     // Main loop.
void networkJoin(void);
void networkLost(void);
void networkTransmit(void);
void dataConfirm(uint8_t, FW_DataStatus_t);
void dataIndication(const FW_DataIndication_t *params);

/*********************************************************************
  User first entry point.
*********************************************************************/
void fw_userEntry(FW_ResetReason_t resetReason)
```

```
{
      // Enabling interrupts.
      TOSH_interrupt_enable();
      leds_open();

      SetNetworkParameters();
      RegisterNetworkEvents();

      // Register end-point for transmission and receive.
      fw_registerEndPoint(END_POINT, dataIndication);

      adc_init();
      if (adc_open(ADC_INPUT_3, processData) == SUCCESS)
            sensorState = SENSOR_READY;

      // Start main loop.
      fw_setUserLoop(200, mainLoop);
}

/**********************************************************************
  Main loop.
**********************************************************************/
void mainLoop()
{
      switch (networkState)
      {
      // In network.
      case NETWORK_JOINED_STATE:
            {
                  if (sensorState == SENSOR_READY) {
                        adc_get(ADC_INPUT_3);
                  } else if (sensorState == SENSOR_READ) {
                        networkTransmit();
                        sensorState = SENSOR_READY;
                  }
                  break;
            }
      case NETWORK_IDLE_STATE:      // Network hasn't been started once
            {
              // Start network.
                  networkState = NETWORK_JOIN_REQUEST_STATE;
                  fw_joinNetwork();
                  break;
          }
      case NETWORK_JOIN_REQUEST_STATE:
            {
            static uint32_t ledTime = 0;
            if ((fw_getSystemTime() - ledTime) > LED_FLASH_DELAY)
                  {
                        leds_toggle(NETWORK_LED);
                        ledTime = fw_getSystemTime();
                  }
            }
      default:
            break;
      }
}
```

```
/************************************
      ADC sensor callback
************************************/
void
processData (uint16_t raw)
{
      int length;

      // Vadc = 1.25V
      // Pressure scale is 0..5 atm
      sensorValue = ((float) raw / 1024) * 1.25 * 5;

      if (sample >= 6) {
            length = sprintf(temp_buffer, "%d.%d atm\n\r",
                                    (int) sensorValue,
                              ((int) (sensorValue * 10000)) % 10000);
            sensorState = SENSOR_READ;
      } else {
      sample++;
      adc_get (ADC_INPUT_3);
      }
}

/***
 ONLY GENERIC CODE BELOW
***/

/*************************************************************************
  Network joint indication.
*************************************************************************/
void networkJoin(void)
{
      // Node was joined - indicating.
      leds_on(NETWORK_LED);
      networkState = NETWORK_JOINED_STATE;
}

/*************************************************************************
  Network loss indication.
*************************************************************************/
void networkLost(void)
{
      // Indicate network loss.
      if (networkState != NETWORK_JOIN_REQUEST_STATE)
            leds_off(NETWORK_LED);
      networkState = NETWORK_JOIN_REQUEST_STATE;
}

void RegisterNetworkEvents()
{
      // Register network events.
      FW_NetworkEvents_t handlers;
      handlers.joined = networkJoin;
      handlers.lost = networkLost;
      handlers.addNode = NULL;
      handlers.deleteNode = NULL;
      fw_registerNetworkEvents(&handlers);
}
```

```
void SetNetworkParameters()
{
        FW_Param_t param;

        // Set node role.
        param.id = FW_NODE_ROLE_PARAM_ID;
        param.value.role = ZIGBEE_END_DEVICE_TYPE;
        fw_setParam(&param);

        // Set MAC addr
        param.id = FW_MAC_ADDR_PARAM_ID;
        macAddr = 2;
        param.value.macAddr = &macAddr;
        fw_setParam(&param);

        // Set PANID.
        param.id = FW_PANID_PARAM_ID;
        param.value.panID = PANID;
        fw_setParam(&param);

        // Set channel mask.
        param.id = FW_CHANNEL_MASK_PARAM_ID;
        param.value.channelMask = CHANNEL_MASK;
        fw_setParam(&param);

        // Set logical address.
        param.id = FW_NODE_LOGICAL_ADDR_PARAM_ID;
        param.value.logicalAddr = 1;
        fw_setParam(&param);
}

/***********************************************************************
  Data transmission over network.
***********************************************************************/
void networkTransmit(void)
{
        // Init data structure
        FW_DataRequest_t params;
        params.dstNWKAddr = 0;  // coordinator
        params.addrMode = NODE_NWK_ADDR_MODE;
        params.srcEndPoint = END_POINT;
        params.dstEndPoint = END_POINT;
        params.arq = TRUE;
        params.broadcast = FALSE;
        params.handle = DATA_HANDLE;
        params.data = temp_buffer;
        params.length = 32;

        if(fw_dataRequest(&params, dataConfirm) == FAIL)
                leds_off(NETWORK_TRANSMISSION_LED);
        else
                leds_on(NETWORK_TRANSMISSION_LED);
}

/***********************************************************************
  Data transmission confirmation handler.
***********************************************************************/
```

```
void dataConfirm(uint8_t handle, FW_DataStatus_t status)
{
      // Clear data transmission led
      leds_off(NETWORK_TRANSMISSION_LED);
}

/************************************************************************
  Data receive indication handler.
************************************************************************/
void dataIndication(const FW_DataIndication_t *params)
{

}
// eof EndDevice.c
```