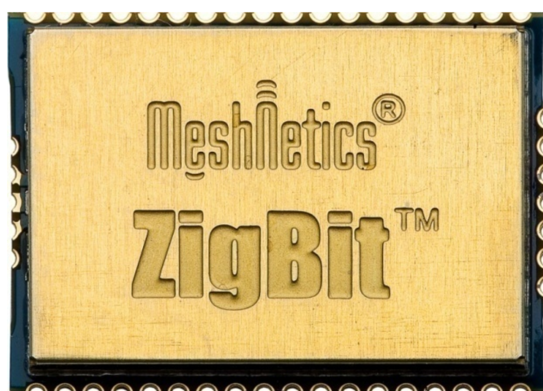
MeshNetics

Next For

ZigBit



Módulo OEM ZigBit™ 1.1
Nota de Aplicación

Utilización de módulos ZigBit™ con sensores analógicos

c/ Doce de Octubre 38 - 28009 Madrid • Telf.: 91 504 02 01 • Fax: 91 504 00 69
e-mail: info@nextfor.com • http://www.nextfor.com

Resumen

Esta nota de aplicación ofrece un tutorial *how-to* para conectar un sensor analógico a ZigBit™ (ZDM-A1281-B0 y ZDM-A1281-A2). La nota detalla los aspectos hardware y software de la utilización del interfaz ADC del módulo para leer datos del sensor, procesarlos y enviarlos a través del aire. Se proporciona un diagrama de cableado y código de aplicaciones. Aunque este documento describe un sensor industrial de presión específico, el proceso recomendado es fácilmente adaptable a la mayoría de las clases de sensores analógicos y las aplicaciones que los envuelven.

Documentos relacionados

[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] eZeeNet™ Software 1.7. eZeeNet™ API. Reference Manual. MeshNetics Doc. P-EZN-452~02

Implementación del hardware

La implementación del hardware cubre el interfaz físico entre el sensor y un módulo ZigBit™. El modelo al que se hace referencia en esta nota de aplicación es un transmisor piezoresistivo Keller (PPT) Serie 21 [2]. Es un típico sensor analógico en lazo de corriente 4-20mA y entrada de 8 a 28V, por lo que el modo de integrarlo con un módulo ZigBit™ puede ser adaptado a cualquier sensor analógico similar.

Para los propósitos de esta nota de aplicación, se supone un módulo montado sobre la placa de desarrollos MeshBean (ver [4]). Como la tarjeta no puede suministrar un voltaje de 3V, se requiere una fuente de alimentación externa y separada para el sensor. En una aplicación real, los desarrolladores pueden elegir reemplazar la alimentación externa por un convertidor DC-DC y usar la fuente de la propia placa para alimentar el sensor. Esta nota asume la utilización de una fuente externa de alimentación.

Cuando se conecta al convertidor ADC, el lazo de corriente analógico debe ser convertido a voltaje de salida mediante una resistencia de precisión. Una preocupación adicional es conectar el sensor tal que su voltaje de salida esté en el rango aceptable, específicamente debajo del voltaje de referencia del microcontrolador del módulo. ATmega 1281 MCE utiliza un voltaje de referencia de 1,25v (ver [1]). La resistencia de precisión fue seleccionada utilizando la familiar ley de Ohm: $V = IR$. Así, si la corriente máxima es $I = 20\ mA$, y el voltaje de referencia $V_{ref} = 1{,}25v$:

$$R \approx \frac{V}{I} = \frac{1{,}25v}{20mA} = 62{,}5\Omega$$

La Figura 1 muestra el diagrama de cableado para el sensor y el módulo. Han sido omitidos los componentes irrelevantes para el cableado del sensor.

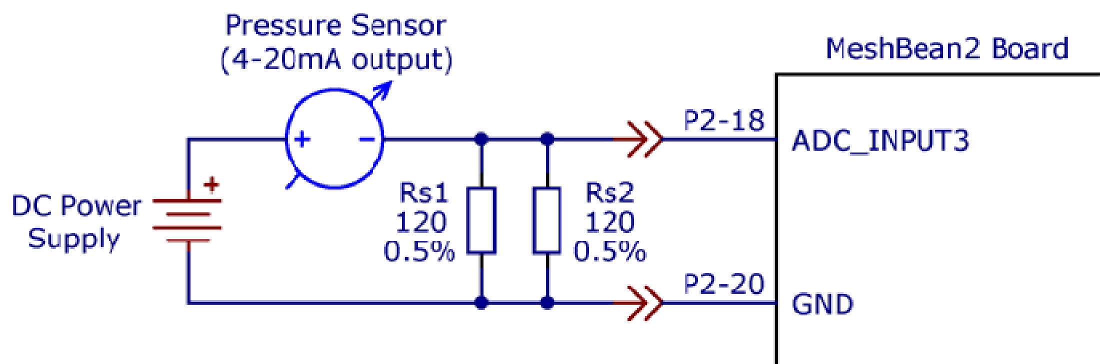


Figura 1: Cableado del módulo al sensor

El montaje final se muestra en la Figura 2. El conector de alimentación y la resistencia se han pegado junto con el conector de extensión instalado en la tarjeta MeshBean. En el esquemático de MeshBean [4] se puede encontrar el conexionado de los pines. Los pines 18 y 19 se conectan a `ADC_INPUT3` y `GND` respectivamente. Existen 3 canales ADC desocupados por defecto en las placas MeshBean, por los que pueden conectarse hasta 3 sensores analógicos independientes sin ninguna modificación del hardware.

Figura 2: Montaje del sensor

Componentes utilizados

El cableado sugerido requiere de los siguientes componentes hardware (además del propio sensor):

- Resistencia
- Conector de alimentación
- Enchufe
- Alimentador

Implementación del software

La aplicación de usuario final que acompaña al montaje hardware proporciona una simple plantilla para la lectura de los datos de un sensor analógico. Estos datos se convierten en un valor significativo de alguna condición del entorno, que se transfiere a un dispositivo de alta funcionalidad. En terminología ZigBit, el montaje corresponde de forma aproximada a un dispositivo final (RFD) que lee un sensor físico y transfiere los datos leídos, vía *wireless,* a un dispositivo coordinador (FFD). El coordinador está simplemente conectado a un PC, que recibe los datos transferidos desde el dispositivo final.

Una aplicación real suele incorporar un controlador central inteligente capaz de desempeñar funciones adicionales de análisis y visualización. Por ejemplo, es posible implementar tareas de control sobre otros dispositivos inalámbricos funcionando como actuadores, agrupación de datos o registros.

El resto de esta nota de aplicación asume que el lector presenta un alto nivel de familiaridad con el Kit de Desarrollo ZigBit. Asimismo, se anima al lector a que se familiarice con las aplicaciones de ejemplo disponibles en ZDK [3].

Interfaz ADC

Una vez que el sensor es conectado físicamente a una de las entradas ADC, se necesita escribir un *driver* para la comunicación con el sensor. En este caso, la única funcionalidad exigida al *driver* es leer una representación digital del voltaje aplicado al pin ADC correspondiente.

El Kit de Desarrollo ZigBit incluye *drivers* de referencia para cada interfaz disponible en el módulo. El API suministrado por el *driver* ADC puede ser encontrado en `adc.h`. El API incluye las siguientes funciones:

- `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);`

Cada aplicación necesita, tan sólo, incluir la cabecera y el enlace en la librería precompilada `libZigBit.a` para utilizar el API ADC estándar. Como la funcionalidad relativa al ADC es parte de la capa de abstracción hardware (HAL), proporcionada en código fuente, los desarrolladores podrán inspeccionar y cambiar el *driver* en función de sus necesidades. El *driver* puede ser encontrado en `adc.c`.

El Apéndice A contiene el código fuente de la aplicación. La configuración estándar UART y la selección de direcciones lógicas ha sido tomada de la aplicación de ejemplo `peer2peer` (ver [5] para una descripción de `peer2peer`). La función `fw_userEntry` contiene una llamada a `adc_open`, que abre el 3er canal del ADC para la lectura (sólo si el dispositivo no es un coordinador, es decir, su dirección lógica es diferente de 0). Como la conversión ADC no es una operación instantánea, una llamada a `adc_open` incluye también una realimentación que se invoca cada vez que se completa una conversión ADC. A su vez, `process_data` convierte el valor sin formato devuelto por el convertidor ADC en una representación de punto flotante.

El MCU Atmega tiene un convertidor ADC de 10 bits, por lo que el valor decimal sin formato debe ser escalado al máximo número expresable en 10 bits, para lo que se divide por $2^{24}=1024$. El valor resultante debe ser escalado por el voltaje de referencia, en este caso 1,25v. Una vez hecha esta conversión, se obtiene una representación digital de la entrada de voltaje en una línea ADC particular. Este valor requiere que se le haga una correlación de acuerdo a las propiedades del sensor específico.

Para el caso particular del PPT Keller (ver [2]), que mide la presión atmosférica en una escala lineal de `-1atm` a `4atm`, basta con realizar una aproximación lineal. El resto del código es bastante sencillo. Después de reunir 5 muestras de entrada ADC, se toma el último, se convierte en *string* y se envía por la red. El nodo receptor (siempre el coordinador) simplemente transfiere el mensaje en formato *string* al UART.

Consejos y trucos para el desarrollo

No es aconsejable procesar las primeras salidas del convertidor ADC porque pueden contener errores relativos a la inicialización lógica del ADC, especialmente si el dispositivo acaba de ser despertado. Así, se deben descartar los las primeras 5-10 muestras de entrada al ADC.

Las placas MeshBean que trabajan como dispositivo final no pueden ser alimentadas por el conector USB, porque los niveles de voltaje del USB son demasiado inestables, incluso aunque el PC esté adecuadamente puesto a tierra. A su vez, una fuente de alimentación con ruido contamina el voltaje de referencia

utilizado por el convertidor ADC, traduciendo todas las lecturas de entrada del sensor a valores poco fiables.

Se sugiere a los desarrolladores que comprueben el ruido de sus fuentes de alimentación. Los experimentos realizados demuestran que las tarjetas MeshBeans alimentadas por batería pueden producir valores ADC con la tolerancia admitida para el ATmega 1281 MCU (+/-2 bits LSB)

Material suplementario

`EndDevice.c` - Código fuente para aplicaciones de usuario final.

Apéndice A: Código fuente

```
/**
* 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
```