DALLAS SEMICONDUCTOR

Application Note 105

High Resolution Temperature Measurement With Dallas Direct-to-Digital Temperature Sensors

www.dalsemi.com

INTRODUCTION

This application note describes the principle of operation of Dallas Semiconductor's line of direct–to–digital temperature sensors, and outlines a method of achieving high (<0.05°C) resolution with these devices. An example C code listing is given for use with the DS1620.

DIRECT–TO–DIGITAL TEMPERATURE SENSOR PRINCIPLE OF OPERATION

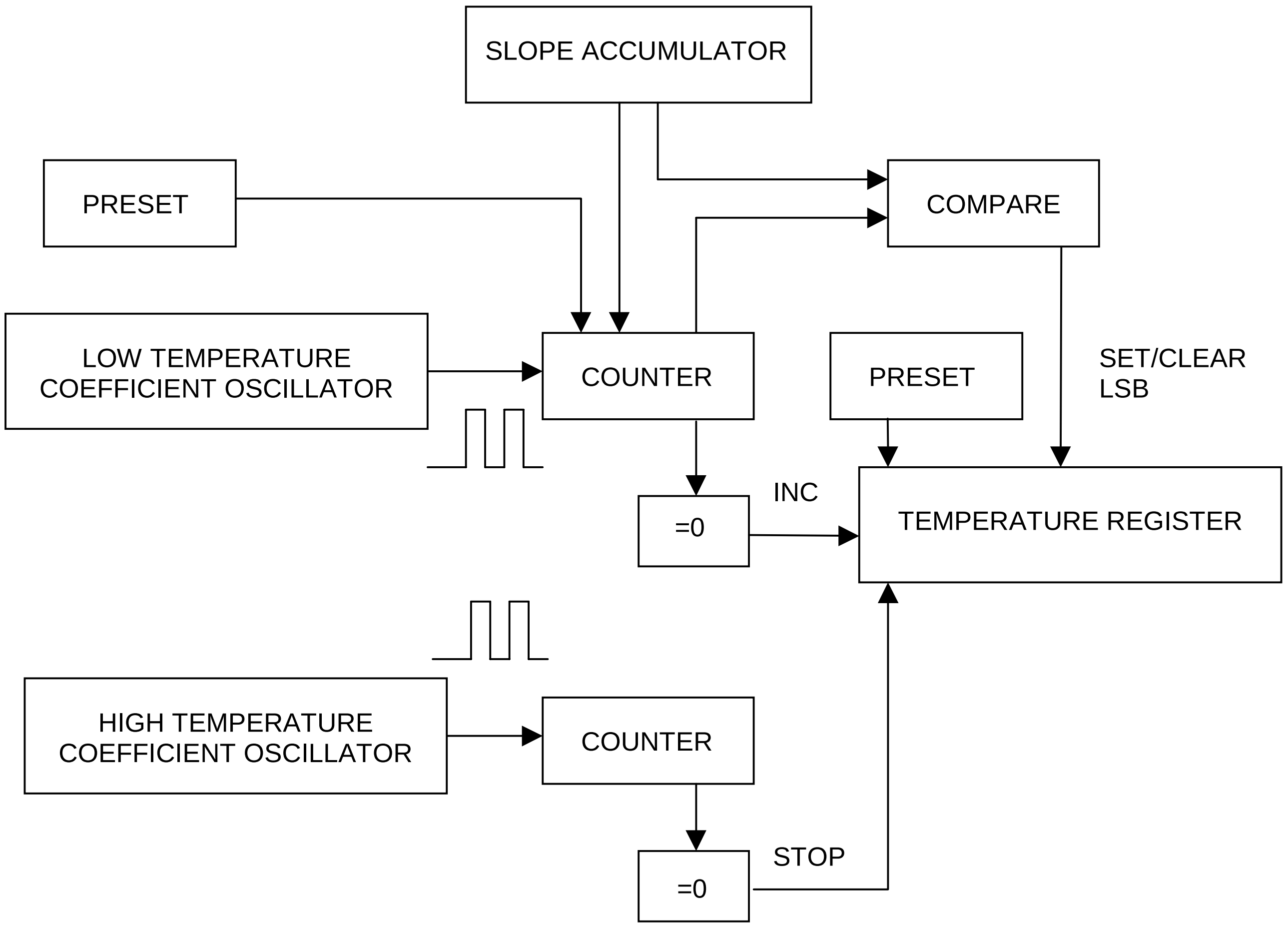
The Dallas Direct–to–Digital Temperature Sensors measure temperature through the use of an onboard proprietary temperature measurement technique. A block diagram of the temperature measurement circuitry is shown in Figure 1.

Each temperature sensor measures temperature by counting the number of clock cycles that an oscillator with a low temperature coefficient goes through during a gate period determined by a high temperature coefficient oscillator. The counter is preset with a base count that corresponds to –55°C. If the counter reaches zero before the gate period is over, the temperature register, which is also preset to the –55°C value, is incremented, indicating that the temperature is higher than –55°C.

At the same time, the counter is then preset with a value determined by the slope accumulator circuitry. This circuitry is needed to compensate for the parabolic behavior of the oscillators over temperature. The counter is then clocked again until it reaches zero. If the gate period is still not finished, then this process repeats.

The slope accumulator is used to compensate for the nonlinear behavior of the oscillators over temperature, yielding a high resolution temperature measurement (0.5°C for almost all the products). This is done by changing the number of counts necessary for the counter to go through for each incremental degree in temperature. To obtain the desired resolution, therefore, both the value of the counter and the number of counts per degree C (the value of the slope accumulator) at a given temperature must be known.

TEMPERATURE MEASUREMENT CIRCUITRY Figure 1



INCREASING TEMPERATURE RESOLUTION

Most of Dallas' direct–to–digital temperature sensors provide 0.5°C resolution directly. This is accomplished by the device determining whether to set or clear the least significant bit (LSB), based on the actual temperature. The device attempts to keep errors within ½ LSB, by quantizing different readings into the LSB step size. For example, a part which is ramping up in temperature from 25°C to +26°C, or down in temperature from –10°C to –11°C, would exhibit this behavior:

ACTUAL TEMPERATURE	SET/CLEAR LSB	DIRECT READING	ACTUAL TEMPERATURE	SET/CLEAR LSB	DIRECT READING
25	Clear	25	-10	Clear	-10
25.1	Clear	25	-10.1	Clear	-10
25.2	Clear	25	-10.2	Clear	-10
25.3	Set	25.5	-10.3	Set	-10.5
25.4	Set	25.5	-10.4	Set	-10.5
25.5	Set	25.5	-10.5	Set	-10.5
25.6	Set	25.5	-10.6	Set	-10.5
25.7	Set	25.5	-10.7	Set	-10.5
25.8	Clear	26	-10.8	Clear	-11
25.9	Clear	26	-10.9	Clear	-11
26	Clear	26	-11	Clear	-11

This example shows then that every reading is rounded up (in absolute value) by ½ LSB. For most parts, this is 0.25°C. This is important to remember, since in doing calculations to achieve higher resolutions, this rounding factor must be taken into account.

With the exception of devices intended for battery management (DS2434 and DS2435), the temperature sensors can measure temperature over the range of –55°C to +125°C in 0.5°C increments. For Fahrenheit usage, a lookup table or conversion factor must be used.

Higher resolutions may be obtained by reading the temperature and truncating the least significant bit from the read value. From the example above, it should be apparent that this must be done on the raw, 9–bit number, in two's–complement arithmetic, in order for the correct reading to result. After the truncation, the number can then be converted into a signed integer. This value is referred to below as temp_read. The value left in the counter can then be read by issuing a special command protocol to the sensor. This value is the count remaining (count_remain) after the gate period has ceased. Reading the value of the slope accumulator (by using another command protocol or set of protocols, as outlined below) yields the number of counts per degree C (count_per_degree) at that temperature. Once these parameters are all known, the actual temperature can be calculated from the following equation:

$$TEMPERATURE = temp\_read - {}^{1}/_{2}LSB + \frac{(count\_per\_degree - count\_remain)}{count\_per\_degree}$$

A simple routine in C, called CalcHiResTemp, is given in the listing of Figure 7.

PROCEDURES FOR READING COUNTER VALUES AND CALCULATING HIGH RESOLUTION TEMPERATURE READINGS

The following is a list of procedures for performing high resolution temperature readings from various digital thermometers. For all parts, conversions must be done in oneshot mode (if applicable). XXh refers to the protocol to send to the part; if two protocols are listed together (e.g., 84h A0h), both protocols should be sent in the given order, without resetting the part between protocols. A variable name XXX is designated as "named XXX."

NOTE:

The high–resolution temperature equation is slightly different for the DS1821 as compared to the DS1620, DS1623, DS1625, DS2434, and DS2435.

DS1620, DS1623, and DS1625:

1. Issue Start Convert protocol (EEh).
2. When conversion is finished, read 9–bit temperature value (AAh).
3. Truncate half–degree bit from reading.
4. Convert truncated value from 2's complement to signed integer (named temp_read).
5. Read 9–bit counter value (A0h; named count_remain).
6. Send undocumented Load Counter protocol (41h).
7. Read 9–bit counter value (A0h; named count_per_degree).

8. Calculate high–resolution temperature using the high resolution temperature equation given in the previous section. ½ LSB = 0.25.

DS1621, DS1624, and DS1820:
The procedure to find the high resolution temperature parameters and the calculation to use are given in the respective product data sheets for these products. Note that since the DS1624 already provides a 13–bit number with 0.03125°C resolution, no further processing is possible to achieve any higher temperature resolution.

DS1821:
1. Issue Start Convert protocol (EEh).
2. When conversion is finished, read 8–bit temperature value (AAh).
3. Convert value from 2's complement to signed integer (named temp_read).
4. Read 9–bit counter value (A0h; named count_remain).
5. Send undocumented Load Counter protocol (41h).
6. Read 9–bit counter value (A0h; named count_per_degree).
7. Calculate high–resolution temperature using the high resolution temperature equation given in the previous section. Note that for the DS1821, ½ LSB = 0.5.

DS2434 and DS2435:
1. Issue Start Convert protocol (D2h).
2. When conversion is finished, read 8–bit temperature value (B2h 61h).
3. Convert temperature value from 2's complement to signed integer (named temp_read).
4. Read 9–bit counter value (84h A0h; named count_remain).
5. Send undocumented Load Counter protocol (84h 41h).
6. Read 9–bit counter value (84h A0h; named count_per_degree).
7. Calculate high–resolution temperature using the high resolution temperature equation given in the previous section. ½ LSB = 0.25.

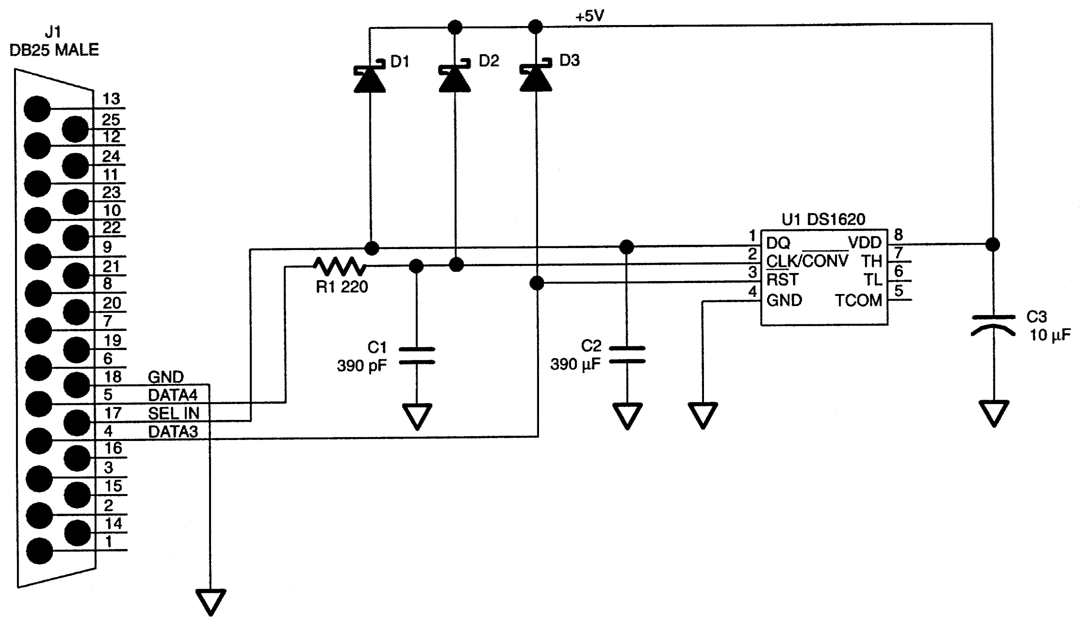
EXAMPLE C CODE

The following example is a simple exerciser for the DS1620 from an IBM–PC compatible computer. It reads temperature and displays it in both the normal 0.5°C resolution and in high resolution. The DS1620 is inter-faced to the parallel port of the PC using the circuit shown in Figure 2, which is the schematic of the DS1620K demo kit.

The hardware for the DS1620K demo kit "steals" power from a parallel port using D1, D2, D3, and C3. Not all PC parallel ports are able to supply sufficient current to make this hardware work, so be advised that if this circuit does not work, try connecting a +5V power source to the +5V line in this circuit and try again. R1 and C1 serve to filter the CLK line and prevent negative under-shoots. Likewise, C2 helps prevent negative under-shoot on the DQ line.

The code is written in an attempt to make it easy to adapt to any of the digital temperature sensors. All that needs to be done to use it with other devices is to change the command header file to account for the protocols and resolution of the device being used, and change the files which manipulate the hardware interface to the device to account for either 1, 2 or 3–wire interfaces.

DS1620K HARDWARE SCHEMATIC Figure 2



THREEWIR.H C CODE LISTING Figure 3

```
/* THREEWIR.H
This file defines the method for setting the 3–wire interface lines from an IBM–PC
compatible computer. The interface is through the parallel port. The parallel port
lines used are as follows:


/SELIN          pin 17   DQ
Data4           pin 5    CLK
Data3           pin 4    /RST
GND             pin 18   GND



The inportb, outportb, and delay functions are defined in most PC–based C compil-ers
in dos.h
*/



#include <dos.h>



#define CLK_HI          p_data|= 0x08; outportb(p_addr,p_data); delay(1);
#define CLK_LOW         p_data&=~0x08; outportb(p_addr,p_data); delay(1);
#define RSTB_HI         p_data|= 0x04; outportb(p_addr,p_data); delay(1);
#define RSTB_LOW        p_data&=~0x04; outportb(p_addr,p_data); delay(1);
#define DQ_HI           outportb(p_addr+2,0x02); delay(1);
#define DQ_LOW          outportb(p_addr+2,0x0A); delay(1);
#define READ_BACK       –((inportb(p_addr+2) & 0x08) >> 3)+1



#define p_addr peek(0,0x0408)  /*  Finds the address of the parallel port     */



char  p_data  =  0xFF;
```

DS3WIRE.C C CODE LISTING Figure 4

```
/* DS3wire.C
These routines handle a Dallas 3–wire interface. Methods of changing the individual
bit states in the hardware need to be defined in ”threewir.h”.
*/


#include ”threewir.h”


/*
Bit level drivers, used in routines for handling the protocol and data interface to
a 3–wire device.
*/


/*  Drives RSTB signal high  (state=1)  or  low  (state=0)  */


void rb3w( int state )
     {
        if (state==0) {RSTB_LOW;}
        else {RSTB_HI}
     }


/* Drives CLK signal high  (state=1)  or  low  (state=0) */


void c3w( int state )
     {
        if (state==0) {CLK_LOW}
        else {CLK_HI}
     }


/* Drives DQ signal high  (state=1)  or  low  (state=0) */
void dq3w( int state )
     {
        if (state==0) {DQ_LOW}
        else {DQ_HI}
     }


/* Reads data back from port  */


int rd_back()
   {
   char i;
   i=READ_BACK;
   return( i );
   }
```

```
/* Writes a 0 (w_bit=0) or a 1 (w_bit=1) to a three wire device.  */

void write_bit( int w_bit )
   {
   if ( w_bit == 0 ) dq3w( 0 );
   else dq3w( 1 );
   c3w( 0 );
   c3w( 1 );
   dq3w( 1 );
   }


/* Reads a 0 (return 0) or a 1 (return 1) from a three wire device.  */


   int read_bit()
   {
   int i;


   c3w( 0 );
   i=rd_back();
   c3w( 1 );


   return(i);
   }


/*
Routines to read and write a Dallas 3-wire part. The three wire interface typically
requires that the part be sent an eight-bit protocol, followed by data. The
exception to this are the digital potentiometers, which use no protocols but send
only data. All transfers are LSB first.

If writing, the data must follow the protocol immediately, but there is no fixed
number of bits that may be data.

If reading, the part must be written with an eight bit protocol; the DQ pin must
then be changed to READ back data. As with writing, there is no fixed number of bits
that may be data.

These routines are set up to handle an arbitrary data length UP TO the size of an
int (which is typically 16 bits for most C compilers).
*/
```

```
/* write_part
Write a command and data to a 3–wire part. param is the integer value of the data to
be written after the write protocol. n_bits is the number of bits to transfer in
param.
*/

void write_part(int protocol, int param, int n_bits)
{
int index,data;

rb3w(1);                                        /* Raise the /RST line  */
for (index = 0; index<8; index++)               /* protocol is 8 bits   */
   {
   data = protocol>>index;
   data &= 0x01;
   write_bit(data);
   }
for (index = 0; index<n_bits; index++)          /* Write out data    */
   {
   data = param>>index;
   data &= 0x01;
   write_bit(data);
   }
rb3w(0); /*                                     Drop /RST line       */
}

/* read_part
Reads data from a 3–wire part. protocol is command to be written to the part, and
n_bits is number of bits to read after the protocol is sent. Returns int value of
reading.
*/

int read_part(int protocol, int n_bits)
{
int index,data;
int r_data = 0;

rb3w(1);                                        /* Raise the /RST line  */
for (index = 0; index<8; index++)               /* protocol is 8 bits   */
   {
   data = protocol>>index;
   data &= 0x01;
   write_bit(data);
   }

for (index = 0; index<n_bits; index++)          /* Read in data      */
   {
   r_data |= read_bit()<<index;
   }
rb3w(0);
return(r_data);
}
```

DS1620CMD.H Figure 5

```
/* 1620CMD.H
This file defines the protocols and bit masks needed for a specific device: the
DS1620.
*/


/* protocols */
#define     Read_Temp         0xAA
#define     Start_Convert     0xEE
#define     Stop_Convert      0x22
#define     Write_TH          0x01
#define     Write_TL          0x02
#define     Read_TH           0xA1
#define     Read_TL           0xA2
#define     Write_Config      0x0C
#define     Read_Config       0xAC
#define     Read_Counter      0xA0
#define     Load_Counter      0x41


/* masks for configuration/status register      */
#define     mode_mask         0x03   /* masks off all but CPU and 1SHOT bits  */
#define     nvb_mask          0x10   /* masks off all but NVB bit             */
#define     done_mask         0x80   /* masks off all but DONE bit            */
#define     flags_mask        0x60   /* masks off all but THF and TLF flag
                                     bits                                     */
#define     flag_offset       5      /* number of right shifts to right-justify
                                     flags                                    */


#define     res               (int) (1.0/LSB)
                                    /*  resolution  factor,  used  in  application  to
calculate
                                    actual temperatures from raw data         */


float       LSB = 0.5;        /* The value of a LSB on the
                              DS1620 in degrees C                             */
```

DEVFUNC.C C CODE LISTING Figure 6

```
/* Device Specific Functions
These are device specific functions for all 3–wire Dallas Temperature Sensors,
supporting all documented functions of the parts, including high resolution
temperature measurements.

Since these routines may be used with many different devices, no attempt is made in
these routines to interpret the data written or read; the application program which
uses these routines must assure that the data written or read is in the appropriate
format for use by these routines and by the application.
*/


/*
Place device–appropriate include file here for specific device commands.
*/
#include <stdio.h>
#include ”1620cmd.h”                  /* This example for DS1620           */


/*prototypes*/


extern void write_part();
extern int read_part();


/* StartTempConvert
This routine issues the Start Convert command. No further data is required and no
data is read back.
*/
void StartTempConvert(void)
{
     write_part(Start_Convert,0,0);
}


/* StopTempConvert
This routine isses the Stop Convert command. No further data is required and no data
is read back.
*/
void StopTempConvert(void)
{
     write_part(Stop_Convert,0,0);
}
```

```
/* ReadTemp
This routine reads back the value of the converted temperature reading and returns
that value. num_bits is the number of bits to read back (up to size of int (16)).
Remember that this only returns the raw data; the calling program must know how to
interpret the data (for example, this routine will return an int with 9 bits when
called as ReadTemp(9); the user must then know that the LSB is 0.5•C, and make the
appropriate data conversion. Likewise, interpreting the sign of the returned
temperature is up to the calling program.)
*/
int ReadTemp (int num_bits)
{


      int temperature;
      temperature = read_part(Read_Temp,num_bits);
      return(temperature);
}


/* WriteTempHigh
This routine writes to the TH thermostat register. Param is value to place in TH,
and should be 9 bits wide. As with ReadTemp, the calling program must assure that
the data sent here is in the appropriate device–specific data form to be used
properly (is LSB 0.5•C or 1•C?, etc).
*/
void WriteTempHigh (int param, int num_bits)
{
      write_part(Write_TH,param,num_bits);
}


/* WriteTempLo
This routine writes to the TL thermostat register. Param is value to place in TL,
and should be num_bits bits wide. As with ReadTemp, the calling program must assure
that the data sent here is in the appropriate device–specific data form to be used
properly (is LSB 0.5•C or 1•C?, etc).
*/
void WriteTempLo (int param, int num_bits)
{
      write_part(Write_TL, param, num_bits);
}


/* ReadTempHigh
This routine reads back from the part the value stored in the TH thermostat regis-
ter and returns that value. num_bits is the number of bits to read back (up to size
of int (16)). Remember that this only returns the raw data; the calling pro-gram
must know how to interpret the data (for example, this routine will return an int
with 9 bits when called as ReadTempHigh(9); the user must then know that the LSB is
0.5•C, and make the appropriate data conversion. Likewise, interpreting the sign of
the returned temperature is up to the calling program.)
*/
int ReadTempHigh (int num_bits)
}
```

```
      int  tHigh;

      tHigh = read_part(Read_TH,num_bits);

      return(tHigh);
}


/* ReadTempLo
This routine reads back from the part the value stored in the TL thermostat register
and returns that value. num_bits is the number of bits to read back (up to size of
int (16)). See the caveat for ReadTempHigh regarding returned data.
*/
int ReadTempLo (int num_bits)
{
      int tLow;

      tLow = read_part(Read_TL,num_bits);

      return(tLow);
}


/* LoadCounter
This routine sends the Load Counter command. No further data is required and no data
is read back.
*/
void LoadCounter (void)
{
      write_part(Load_Counter,0,0);
}


/* ReadCounter
This routine reads back from the part the value in the temp sensor count register
and returns that value. num_bits is the number of bits to read back (up to size of
int (16)).
*/
int ReadCounter (int num_bits)
{
      int count;

      count = read_part(Read_Counter,num_bits);

      return(count);
}
```

```
/* WriteConfig
This routine writes to the configuration register. Param is the byte to be written
into the configuration register, and it is always 8 bits wide.
*/
void WriteConfig (int param)
{
      write_part(Write_Config,param,8);
}


/* ReadConfig
This routine reads back the value of the configuration byte and returns that value.
It always returns 8 bits.
*/
int ReadConfig(void)
{
      int regvalue;

      regvalue = read_part(Read_Config,8);
      return(regvalue);
}


/* CheckIfBusy
This routine will check to see if the device is busy and cannot process further
writes to EEPROM. It checks the NVB flag in the configuration register. If the
device is busy, the function returns a 1; if it is not busy, the function will
return a 0.
*/
int CheckIfBusy(void)
{
      if ((ReadConfig() & nvb_mask) == nvb_mask)
              return(1);
      else
              return(0);
}


/* CheckIfDone
This routine will check to see if the device has completed a temperature conver-
sion.  It checks the DONE flag in the configuration register. If the device is busy,
the function returns a 1; if it is not busy, the function will return a 0.  NOTE: If
the mode is set so that the temp sensor is doing continuous conversions, DONE will
NEVER be set, so make sure you know what mode you're in if you use this check
routine.
*/
int CheckIfDone(void)
{
      if ((ReadConfig() & done_mask) == done_mask)
              return(0);
      else
              return(1);
}
```

```
/* SetMode
This routine sets the operating mode of the digital temperature sensor. Mode is
determined by two bits in the configuration register: CPU and 1SHOT. The following
modes are possible:


      Mode        CPU         1SHOT
      1           0           0           Standalone, continuous conversions
                                          (typical for thermostat)
      2           0           1           Standalone, one shot (used only if
                                          conversions can be started by CONV pin)
      3           1           0           CPU, continuous conversion (typical for
                                          thermostat with readback)
      4           1           1           CPU, one shot (typical for low–power
                                          temp readings)


This routine will set the configuration register bits appropriately for the MODE
value passed to the routine in mode. It returns no data.
*/
void SetMode(int mode)
{
      WriteConfig (mode–1);
}


/* ReadFlags
This routine will read the state of the two temperature flags, THF and TLF. It
returns an integer, t_flag, which is the number of the two bits (0 = 00, 1=01, 2=10,
3=11).
*/
int ReadFlags(void)
{
      int t_flag;

      t_flag =( ReadConfig() & flags_mask) >> flag_offset;
      return(t_flag);
}
```

SIMPLE DS1620 EXERCISER C CODE LISTING Figure 7

```
/* DS1620 Exerciser
Example program to show how to perform various functions of a DS1620 Digital Ther-
mometer.  This example is console–driven; that is, it expects input from the user
via a console interface, to set the mode and TH, TL register values.
*/


#include <stdio.h>
#include ”devfunc.c”
#include ”ds3wire.c”


/*prototypes*/
void    StartTempConvert(void);
void    StopTempConvert(void);
int     ReadTemp(int);
void    WriteTempHigh(int,int);
void    WriteTempLo(int,int);
int     ReadTempHigh(int);
int     ReadTempLo(int);
void    LoadCounter(void);
int     ReadCounter(int);
void    WriteConfig(int);
int     ReadConfig(void);
int     CheckIfBusy(void);
int     CheckIfDone(void);
void    SetMode(int);
int     ReadFlags(void);
void    InitTempSensor(void);
void    SetThermostat(void);
float   CalcHiResTemp(int,int,int);


/* InitTempSensor
This routine will initialize the temperature sensor to a mode specified by the user.
*/
void InitTempSensor(void)
{
      int mode, test;

      printf(”1. Standalone mode, Continuous Conversions\n”);
      printf(”2. Standalone mode, One Shot\n”);
      printf(”3. CPU mode, Continuous Conversions\n”);
      printf(”4. CPU mode, One Shot\n”);
      printf(”Enter mode desired:”);
      scanf(”%d”,&mode);
      SetMode(mode);                      /*Write mode bits to Config register */
      do                                        /* Wait until EEPROM write
                                                complete             */

      {
         test = CheckIfBusy();
         printf(”Device Busy”);
      } while (test);


}


/* SetThermostat
This routine sets the thermostat registers to the high and low values specified by
the user.
*/
void SetThermostat(void)
{
      float    t_hi, t_lo, high_t, low_t;       /* Since TH and TL are 9–bit
                                                numbers with a 0.5•C
                                                LSB, they must be of type
                                                float. This program
                                                must then assure the data
                                                passed to the other
                                                routines is formatted
                                                properly.             */
      int      hi_t, lo_t, busy;


      printf(”Enter a High Temperature Limit:”);
      scanf(”%f”,&t_hi);                        /* Get TH limit */
      printf(”Enter a Low Temperature Limit:”);
      scanf(”%f”,&t_lo);                        /* Get TL limit */


      hi_t = res*t_hi;                          /* turn into a 9–bit integer –
                                                note that this routine
                                                does not check to see if the
                                                user enters some number that
                                                can’t be used, like 22.635. An
                                                actual application would
                                                want to do such error trap
                                                ping.                 */
      WriteTempHigh(hi_t,9);                    /* Write to TH register*/


      do                                        /* Wait until EEPROM write
                                                complete */
      {
         busy = CheckIfBusy();
         printf(”Device Busy”);
      } while (busy);
      lo_t= res*t_lo;                          /* same comments as for TH
                                                                   */
      WriteTempLo(lo_t,9);                     /* Write to TL register
                                                                   */


      do                                       /* Wait until EEPROM write
                                               complete            */
      {
         busy = CheckIfBusy();
         printf("Device Busy");
      } while (busy);


}


/* CalcHiResTemp
This routine handles the calculation of temperature in high resolution mode.
Requires that the part has provided the values of temperature, count_per_degree,
and count_remain. This routine assumes 9 bit data is read back, and that the user
has defined the value of LSB and res in the appropriate DSXXXXCMD.H file.
*/
float   CalcHiResTemp(int temp9, int count_per_degree, int count_remain)
{
      int   temp_read;
      float frac_deg;


      temp_read = temp9/res;                   /* temp9 is raw data - convert
                                               for LSB value       */
      if(temp_read>>8>0)                       /* check to see if MSB set -
                                               works for 9-bit data.   */
      temp_read -=256;                         /* convert to negative temper
                                               ature               */

      if(count_per_degree !=0)
         frac_deg = (count_per_degree-count_remain)/((float)count_per_degree);
      else
         frac_deg = LSB/2.0;


      return(((float)temp_read-(LSB/2.0))+frac_deg);
}


void main(void)
{
      signed int temperature;
      float read_temp;
      int   Countrem,CountC;
      float hires_temp;
      int   busy,key;
      InitTempSensor();                        /* Initialize sensor with mode
                                               Bits                */
      SetThermostat();                         /* Load in Thermostat values,
                                               if used             */
      StartTempConvert();                            /* Start Temperature
                                               Conversion(s)       */

      switch(ReadConfig()& mode_mask)          /* Checks for conversion done.
                                               This switch is used to
                                               determine what to do depending
                                               upon the mode the
                                               DS1620 is in. If continuous
                                               conversions are going on,
                                               temperature readings should
                                               only be made periodically
                                               based on time. If ONE-SHOT
                                               mode is used, then you can
                                               poll the status register for
                                               the DONE bit to set.    */
     {
        case 0:                                /* Continuous conversions – so
                                               read based on time */
           delay(1000);
           break;
        case 1:                                /* One Shot mode – poll status
                                               register for DONE bit */
           do
           {
                 busy =CheckIfDone();
                 printf(”\nConversion in progess”);
           }while(busy);
           break;
        case 2:                                /* Continuous conversions – so
                                               read based on time */


           delay(1000);
           break;
        case 3:                                /* One Shot mode – poll status
                                               register for DONE bit */
           do
           {
                 busy =CheckIfDone();
                 printf(”\nConversion in progess”);
           }while(busy);
           break;
}/* end switch */
```

```
      read_temp = (float)ReadTemp(9)*LSB;       /* Reads temperature and con
                                                verts for LSB value*/
      if ((int)read_temp>>7>0)                  /* Check if MSB set
                                                                   */
         read_temp –=256;                       /* if it is, correct for it
                                                                   */


      printf(”\nThe temperature is:%5.2f”,read_temp);
      /* prints temperature on console, to 0.5•C resolution */


      StopTempConvert();                        /* In case you were previously
                                                in continuous mode, stop
                                                                   */
      SetMode(4);                               /* MUST BE in ONE–SHOT to do
                                                High Res */
      StartTempConvert();                       /* Start Temperature
                                                Conversion(s)      */
      do                                        /* Check to see when conver
                                                sion completes by polling DONE
                                                bit                */
      {
         busy =CheckIfDone();
         printf(”\nConversion in progess”);
      }while(busy);


      read_temp = (float)ReadTemp(9);           /* get raw 9–bit temperature
                                                data               */
      if ((int)read_temp>>8>0)                  /* Check if MSB set*/
         read_temp –=512;                       /* if it is, correct for it –
                                                9 bits now!        */


      temperature = (signed int) read_temp;     /* make sure temp is now a
                                                signed integer     */
      Countrem = ReadCounter(9);                /* reads count remaining
                                                                   */
      LoadCounter();                            /* Loads counter reg with
                                                counts per degree  */
      CountC = ReadCounter(9);

      hires_temp = CalcHiResTemp(temperature,CountC,Countrem);

      printf(”\n The temperature is :%5.2f”,hires_temp);
      /* prints temperature on console, to 0.01•C resolution */


}
```

RELATED PRODUCTS:

DS1620, DS1621, DS1623, DS1625, DS1820, DS1821, DS2434, DS2435