Application Report
SZZA045 - January 2004

Implementing a 4-mA to 20-mA Current Loop on TI DSPs

Steve Sams and Jim Lyday — Signal Processing Solutions/Semiconductor Group

ABSTRACT

Use of 4-mA to 20-mA (4−20 mA) current loops has become the standard in the process-control industry due to their increased resistance to noise compared to voltage-modulated signals. TI digital signal processors (DSPs) increasingly are becoming the standard for microcontrollers in a control application. Therefore, an opportunity exists to connect the many sensors and electromechanical devices on the market with a current loop, using microcontrollers to translate the signals. Using TI DSPs and Burr−Brown devices, an example illustrates applications that combine these two technologies to create a complete current-loop solution.

Contents

List of Figures

List of Tables

1 4-mA to 20-mA (4−20 mA) Current Loop

Current loops are a method of transmitting information. The information is carried as a current of varying levels of intensity, representing a continuum of values similar to the voltage output of an analog-to-digital (A/D) converter, but without the noise and line-length concerns. For the 4−20-mA current loop, 4 mA normally represents the zero-value output of the sensor, and 20 mA represents the full-scale output.

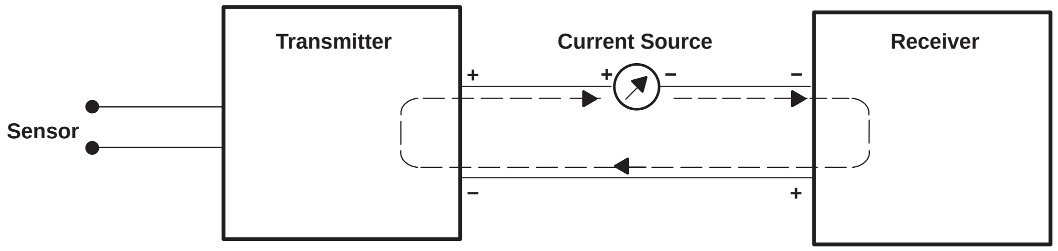

Figure 1. Example of a Current Loop

A typical current loop includes a sensor, transmitter, receiver, and current source (or power supply). The sensor measures a physical parameter (e.g., temperature) and provides an output voltage. The transmitter converts the sensor’s output to a proportional 4−20-mA dc current. The receiver converts the 4−20-mA current into a voltage for additional processing and/or display. The current source supplies the power for the entire system. In our example, the power supply is built into the transmitter.

The 4−20-mA current has many advantages. It is not susceptible to noise, making its use ideal for industrial environments. The signal can travel over long distances and is limited only by the current source, allowing information to be sent or received from remote locations. Wire-break detection is simple due to the 4-mA offset of the zero value. (If the receiver reads 0 mA, there is a break in the circuit.)

2 DSP Applications

Digital signal processors (DSPs), with their capabilities and characteristics, are a perfect complement to current loops. The capabilities consist of real-time signal processing and complex algorithm computing. Two of the complementary characteristics are low power consumption and being a complete system on chip. In a complete DSP current-loop solution, the DSP can be located at the receiver or the transmitter of the current loop.

Specifically, TI C2xxx DSPs particularly are well suited for control applications. All feature built-in A/D converters, complex event managers, pulse-width modulation (PWM), and many other control-oriented features. Some include onboard flash memory for program stability and speed. They can be used to monitor a variety of process characteristic values and to calculate a response accordingly. Furthermore, their ease of high-level language programmability via emulator enables rapid code development and easy software upgrades.

2.1 Receiver Applications

At the receiver, the DSP adds computing power and versatility to the current loop. When receiving the data, the DSP can process the data with more complex algorithms than programmable logic controllers (PLCs), which are commonly used in control loops. The DSP can collect large amounts of data. In a recent survey, system integrators “cited information gathering as the emerging trend in automation system requirements.”[6] Systems that start out as simple control and monitoring “evolve into comprehensive data mining systems, performing analysis on trends, costs, tolerances, etcetera.”[6]

The industry standard for sensors and monitoring equipment is the 4−20-mA current loop because of its high noise tolerance. However, DSPs cannot directly handle a current input. Most DSPs accept only digital data, requiring external current-to-voltage conversion, followed by A/D conversion.

A/D conversion is the process of changing continuously varying data into discrete digital quantities. The magnitude of the data depends upon the number of bits and a reference voltage. As an example, if a hypothetical 3-bit A/D converter were used on a 4−20-mA current loop, the resolution of the A/D conversion is $(20-4)/(2^3-1)$, or ~2.29 mA. For the current loop, a possible conversion chart is shown in Table 1.

Table 1. 3-Bit A/D Conversion Chart for 4−20-mA Current Loop

Digital Value	4−20-mA Current-Loop Value (mA)
000	x = 4
001	4 < x = 6.29
010	6.29 < x = 8.57
011	8.57 < x = 10.86
100	10.86 < x = 13.14
101	13.14 < x = 15.43
110	15.43 < x = 17.71
111	17.71 < x = 20

NOTE: Clearly, three bits would be insufficient; the table is for illustrative purposes only.

The C2xxx DSPs have built-in two-channel A/D converters. By utilizing the two channels, two A/D conversions can occur simultaneously. The C24x family features 10-bit converters, while the C28xx family has 12-bit A/D converters. This saves on-board chip space, power consumption, and glue logic. If more resolution is required, the DSPs have a 16-bit bus that connects easily with external 16-bit A/D converters. However, the DSP still cannot directly handle current signals; it can only perform conversions on voltages.

A common method of converting a 4−20-mA signal to voltage is to use a 250-Ω precision resistor, connected between the signal and ground. This causes a 5-V drop at full scale and uses 100 mW of power. The RCV420 device has a sense resistance of 75 Ω, causing a 1.5-V drop at full scale and using 30 mW of power.

The Burr-Brown RCV420 device provides a sense-chip conversion from a variety of input current-loop configurations and can convert to a variety of output voltage ranges. In addition, the RCV420 allows voltage offset correction and voltage range correction.

With TI DSP devices monitoring system sensors, sophisticated algorithms can be implemented to determine system conditions and calculate appropriate responses. These can range from using Fast Fourier Transforms (FFTs) to filter for particular characteristics to simply monitoring conditions at threshold levels. The DSP then can respond via digital response, PWM, digital-to-analog (D/A) conversion, or other options.

2.2 Transmitter Applications

When a DSP is used to provide external control to a system, often it is necessary to provide an analog control signal. The C2xxx families have 16-bit buses, so it is possible to provide 2^{16} or 65536 steps of high-speed resolution when using a 16-bit parallel input D/A converter, such as the TI DAC8541. If high speed is not necessary, many DSPs can accept serial data through a serial-port interface (SPI).

Once the signal has been converted to an analog voltage, many systems require the control signal in a current-loop format. The XTR110 can accept a 0−5-V signal and convert it to a 4−20-mA signal.

Many current-loop converters require external powering of the current loop, but the XTR110 can self-power the loop. It then can be used to control a variety of control devices such as valves, motors, solenoids, etc.

Feedback from the target control devices can be supplied by a current loop to implement a control loop that ensures optimum adjustment by the system. A proportional-integral-derivative (PID) equation loop implements feedback that is optimized to provide the fastest possible response without creating a nonconvergent oscillation about the desired target.[8]

The proportional term of the PID loop simply limits the response as it approaches a target. This form of process control has been around since the early days of steam engines. James Watt used a ball to limit steam flow as it approached the maximum acceptable value. This control acts as a governor, with the limitation that the proportional controller stops working when the process variable is close to the setpoint, and leaves a nonzero error.

The integral term of the PID loop eliminates the error left from the proportional term of the PID loop. This has been understood since the 1930s. At its simplest, this process automatically resets the setpoint to an artificial value so that the process variable matches the desired setpoint. “This automatic reset operation is mathematically identical to integrating the error and adding that total to the output of the controller’s proportional term.”[8] Proportional-integral (PI) action does not ensure error-free feedback. If the integral action is too aggressive, a PI controller can cause closed-loop instability by overcorrecting for an error and creating a new error of greater magnitude in the opposite direction. When this occurs, the controller experiences hunting, which is the phenomenon of the controller driving its output back and forth between fully on and fully off.

The derivative term of the PID loop can prevent hunting and enable fast response to fundamental changes in the system. The derivative action is active only when the error is changing. When an abrupt change to the error is introduced (e.g., the setpoint is changed), the derivative action forces the controller to start corrective action immediately, before the integral or proportional action responds. When successfully implemented, a full PID controller can appear to anticipate the requirements necessary to maintain the process variable at the new setpoint.[8] Unfortunately, adding the derivative action to a full PID controller does not guarantee a perfect feedback loop. If the derivative action is too aggressive, it can cause hunting by itself. In applications that require slow and steady changes in the controller's output, it may be advisable to use a PI controller, leaving off the derivative action completely. Also, for applications that involve noisy measurements, the derivative term contributes to the output of the control every time the process variable appears to change.

Thus, while the PID controller is not perfect, it had become the state of the art by the 1950s and remains predominant to this day.[8] A DSP, with its ability to implement a PID equation, is an ideal choice to control a feedback loop.

3 A DSP and Current-Loop Example

For reference, an example of a system using current loops to provide signals to and from a DSP is shown in Figure 2. Such a system of can use a sophisticated mathematical algorithm to calculate the current state of the system, predict the direction of the system, and implement an effective PID loop to dynamically adjust the system.

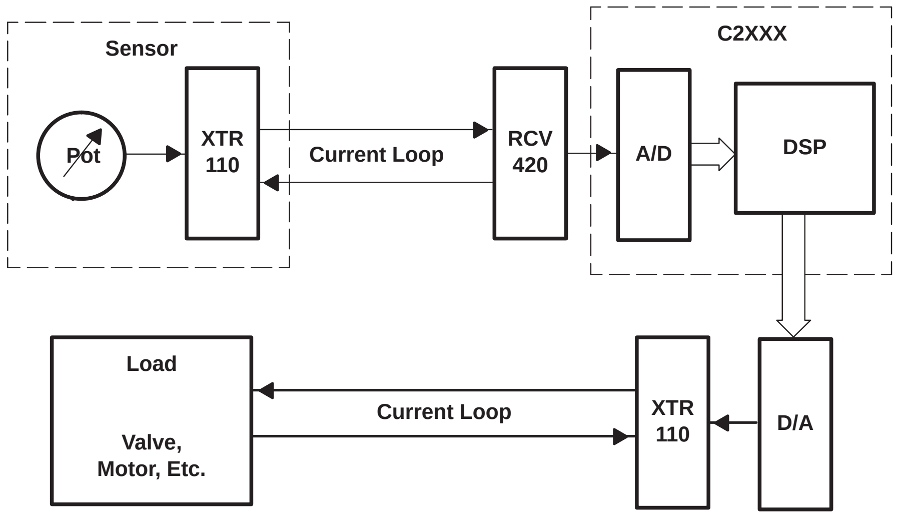

Figure 2. DSP and Current-Loop Example

This system was built and demonstrated in a laboratory environment. A sensor was represented by a potentiometer connected between 5 V and ground, with the wiper arm connected to an XTR110. This configuraiton provided an easily adjustable laboratory simulation of a system sensor with a 4−20-mA output, such as a thermocouple.

This current loop then was provided to an RCV420, which converted the signal to 0−5 V. This signal was fed to a C2xxx DSP for A/D conversion. The DSP used in the laboratory was the 320F240, which is available in both commercial and military configurations.

The DSP calculated an appropriate response and sent out a target value intended for the system control device, in this case a 4−20-mA controlled valve. This control required conversion of the output voltage to a 4−20-mA current, which was implemented by an XTR110 device.

A PID feedback loop was not implemented in the laboratory, but one easily could have been implemented by using the potentiometer built into the valve and routing the resultant voltage into an unused channel on the 10-bit A/D converter on the 320F240.

4 Conclusion

Current loops are a critical element of many industrial control applications because of their superior line length and noise suppression. DSPs increasingly are becoming critical elements of the control system, providing capabilities to interpret system conditions, calculate a response, and control the response via PID loops. Utilizing TI DSPs, DACs, ADCs, and current-loop transmitters/receivers, a complete control system can be implemented with a minimum of cost, board area, and power consumption.

5 References

1. *4−20mA Current Loop Primer* (Datel Application Note DMS-AN20), DATEL, Inc., http://www.datel.com/data/meters/dms-an20.pdf.
2. Vance VanDoren, "Loop Tuning Fundamentals," *Control Engineering*, July 2003: 30−32.
3. Morten Moller, *How They Work: The 4−20mA Current Loop*, sensorland.com, http://www.sensorland.com/HowPage028.html.
4. *RCV420 Precision 4mA to 20mA Current Loop Receiver* (TI Literature Number SBVS019), Texas Instruments. http://focus.ti.com/lit/ds/symlink/rcv420.pdf.
5. Mark R. Stitt and David Kunst, *IC Building Blocks Form Complete Isolated 4−20mA Current-Loop Systems* (TI Literature Number SBOA017), Texas Instruments. http://focus.ti.com/lit/an/sboa017/sboa017.pdf.
6. Vance VanDoren, "System Integrators Broaden Their Horizons," *Control Engineering,* May 2003: 47−51.
7. *XTR110 Precision Voltage-to-Current Converter/Transmitter* (TI Literature Number SBOS141), Texas Instruments. http://focus.ti.com/lit/ds/symlink/xtr110.pdf.
8. Vance VanDoren, "PID: Still the One," *Control Engineering*, October 2003: 32−37.

Appendix A. Code to Interface Between DSP and 4−20-mA Current Loops

The attached code is used to demonstrate the hardware interface between the DSP and the 4−20-mA current loops.

A.1 App_MAIN.c

App_MAIN.c contains the main routine. It initializes several variables, initializes the onboard ADC, and initializes the wait states for writing to the DAC on the F243 EVM board. Next, it reads in a value from the ADC. This value has been converted from a current loop to a voltage (value provided simulates system process data). It right shifts the data to align with the DAC, then writes out the DAC for conversion to analog voltage that is converted to a current loop that drives a valve controller.

```
/* Code used to demo interface of F243 with 4-20mA signals      */
/* Code was generated for the F243 EVM board utilizing routines */
/* provided with the Code Composer / EVM software set           */
/* App_MAIN.c                                                   */

#include ”sys243.h”
#include ”prot243.h”

/*declarations for external test routines*/

/*variable declarations */
unsigned int error_code;    /* ls byte = error code */
unsigned int mon_level;
unsigned int mon_chan;
unsigned int valve_chan;
volatile unsigned int configdata;


void set_wait(void)
{
  configdata = IOWSB1;   /* 2 waits for off chip I/O */
  OUTMAC( _WSGR, configdata);
}

void bad_trap(void)
{
  while(1);                      /* a place to hang if illegal trap */
}

void app_init(void)        /*initializing some variables for startup*/
{
  error_code = 0;                /* show no error */
  mon_chan = 1;   /* use ADC channel 1 for simulated process monitor */
  valve_chan = 1;             /* use DAC channel 1 for valve control  */
  set_wait();        /* set up wait states for I/O, located in dac.c */
}

void main(void)
{
  unsigned int error_state;

  app_init();                            /* init variables & hardware */
  atod_init();                           /* located in dac.c */
  while (error_code !=1)
  {
      mon_level = read_atod(mon_chan);   /* read in the data via onboard ADC*/
   mon_level >>= 4;                /* right shift 4 to align w/ DAC */
   write_dac(mon_chan,mon_level);
/* adjust valve position to equal relative input voltage */
  }

error_finish:
  goto error_finish;                   /* a place to break if error */
}
```

A.2 DAC.c

DAC.c contains the routines write_dac() and read_atod(). These routines handle the hardware interface to place data out on the DAC and read in from the onboard ADC.

```
/*-------------------------------------------------------*/
/*  SPECTRUM DIGITAL INC.                                */
/*-------------------------------------------------------*/
/*  tests external dac to on chip ad requires loop back  */
/*  connector                                            */
/*-------------------------------------------------------*/
/*  file name: dac.c                                     */
/*  Started on:  02/11/1997    by mls                    */
/*  Last update: 02/18/2000    by rrp for 243            */
/*                                                       */
/*  11/15/02                                             */
/*  Updated by Steve Sams to read from controls and output*/
/*  to valve controller via DAC                          */
/*-------------------------------------------------------*/

#include ”dac.h”
#include ”ioreg.h”
#include ”prot243.h”
#include ”sys243.h”

volatile unsigned int dacdata;
volatile unsigned int ADCVals[8];
volatile unsigned int CmpValLow[8];
volatile unsigned int CmpValHigh[8];
volatile unsigned int CmpValErrL[8];
volatile unsigned int CmpValErrH[8];

/*
  ======================  CODE FOLLOWS  ============================
*/
unsigned int dac_delay(volatile unsigned int ctr2)
{
    volatile unsigned int ctr1;

    ctr1 = 0;
    while (ctr2)
    {
   ctr1++;
   ctr2--;
    }
    return(ctr1);
}

#define MS_TIME_LOOP   0x0680

unsigned int wait_ms( volatile unsigned int delay_val )
{
    unsigned int i;
    unsigned int ms_ctr;


    for ( i = 0; i < delay_val; i++ )
    {
   ms_ctr = MS_TIME_LOOP;

   while ( ms_ctr )
   {
       ms_ctr--;
   }
    }

    return( delay_val );
}



void write_dac(unsigned int dac_num, unsigned int dac_val)
{
  unsigned int write_dac_status;

  if (dac_num < 4)
  {
    dacdata = dac_val;                /* put in global location */

    switch (dac_num)
    {
      case DAC0:
        OUTMAC( _DAC0, dacdata);
        break;

      case DAC1:
        OUTMAC( _DAC1, dacdata);
        break;

      case DAC2:
        OUTMAC( _DAC2, dacdata);
        break;

      case DAC3:
        OUTMAC( _DAC3, dacdata);
        break;

    }

    asm("    nop" );
    asm("    nop" );
    asm("    nop" );
    asm("    nop" );
    asm("    nop" );
    asm("    nop" );


    OUTMAC( _DAC_XFER, dacdata);      /* just need a write, any data */
  }

}

void clear_fifos( void )
{
    volatile unsigned int dummy_val;
    volatile unsigned int ADCtrl2;

    ADCtrl2 = *(volatile unsigned int *)ADCTRL2;

    while( (ADCtrl2 & ADCFIFO1ST )) /* FIFO Empty */
    {
       dummy_val = *(volatile unsigned int *) ADCFIFO1;   /* read fifo #1 */
       dummy_val = *(volatile unsigned int *) ADCFIFO1;   /* read fifo #1 */
       ADCtrl2 = *(volatile unsigned int *)ADCTRL2;
    }

    while( (ADCtrl2 & ADCFIFO2ST )) /* FIFO Empty */
    {
       dummy_val = *(volatile unsigned int *) ADCFIFO2;   /* read fifo #1 */
       dummy_val = *(volatile unsigned int *) ADCFIFO2;   /* read fifo #1 */
       ADCtrl2 = *(volatile unsigned int *)ADCTRL2;
    }
}



void ad2_init(void)
{

    volatile unsigned int ADCtrl1;
    volatile unsigned int ADCtrl2;
    volatile unsigned int RdADCtrl1;
    volatile unsigned int RdADCtrl2;

    /* Set prescaller */

    ADCtrl2 = ADC_PRESCALE1;
    *(volatile unsigned int *)ADCTRL2 = ADCtrl2;


    /* Enable AD 1 */

    ADCtrl1 = ADC_SOFT | ADC2EN |ADC1EN;
    *(volatile unsigned int *)ADCTRL1 = ADCtrl1;

    /* Clear out any old results */
    clear_fifos();

    RdADCtrl2 = *(volatile unsigned int *)ADCTRL2;
    RdADCtrl1 = *(volatile unsigned int *)ADCTRL1;

    if ( RdADCtrl1 & ADCINTFLAG )
    {
       *(volatile unsigned int *)ADCTRL1 = RdADCtrl1;
    }

    RdADCtrl1 = *(volatile unsigned int *)ADCTRL1;


}

#define ADC_ERROR  0xFFFF


volatile unsigned int read_atod(unsigned int chan_num)
{
    volatile unsigned int ADCtrl1;
    volatile unsigned int ADCtrl2;
    volatile unsigned int TimeOutCtr;
    volatile unsigned int ADCFifoVal1;
    volatile unsigned int ADCFifoVal2;
    volatile unsigned int chan_num1;
    volatile unsigned int chan_num2;

    chan_num1 = chan_num;
    chan_num2 = chan_num;

    if (chan_num1 > 8)
    {
   return( (unsigned int )ADC_ERROR );
    }
    else
    {
   chan_num1 <<= 1;
   chan_num2 <<= 4;
    }

     /* Make sure Conversion is not in progress */

     do
     {
   ADCtrl1 = *(volatile unsigned int *)ADCTRL1;
     }while( ADCtrl1 & ADCEOC );


    /* set the mux */

    ADCtrl1 = *(volatile unsigned int *)ADCTRL1;

    ADCtrl1  = ( ADC_SOFT | chan_num2 | chan_num1 | ADC2EN | ADC1EN );
    *(volatile unsigned int *)ADCTRL1 = ADCtrl1;

    /* turn on enable bits and start of conversion */
    ADCtrl1  |= ( ADCIMSTART | ADCSOC);
    asm("   nop");
    asm("   nop");
    asm("   nop");
    *(volatile unsigned int *)ADCTRL1 = ADCtrl1;

    asm("   nop");
    asm("   nop");
    asm("   nop");
    /* Read Back For Debug */
    ADCtrl1 = *(volatile unsigned int *)ADCTRL1;

    asm("   nop");
    asm("   nop");
    asm("   nop");

    TimeOutCtr = 0xffff;


    while ( TimeOutCtr-- )
    {
   ADCtrl1 = *(volatile unsigned int *)ADCTRL1;

   /* Finished the Conversion */
   if(!(ADCtrl1 & ADCEOC ))
   {
       break;
   }
    }

    if( TimeOutCtr )
    {
   ADCFifoVal1 = *(volatile unsigned int * )ADCFIFO1;
   /* ADCFifoVal1 >>= 6;           shift it down -- commented out – want original
val SS*/
   ADCFifoVal2 = *(volatile unsigned int * )ADCFIFO2;
   /* ADCFifoVal2 >>= 6;           shift it down  -- commented out – want
original val SS*/
   return( ADCFifoVal1 );
    }

    return( ( unsigned int )ADC_ERROR );
}
```

A.3 sys243.h

sys243.h declares a variety of constants and defines a pair of useful macros: INMAC and OUTMAC. These macros are convenient for writing and reading from hardware mapped to the I/O space.

```
/*-----------------------------------------------------------*/
/*   Header file for EVM243 Hardware Dependent Routines    */
/*    filename: sys243.h                                   */
/*    Includes: CAN Bus                                    */
/*              Switches                                   */
/*              LEDS                                       */
/*              Internal 203 UART                          */
/*              External Memory                            */
/*-----------------------------------------------------------*/
/*   Spectrum Digital Incorporated                         */
/*   Copyright (c) 1998 Spectrum Digital Inc.              */
/*-----------------------------------------------------------*/
/*   Target: EVM320C243   Assy:504030                      */
/*-----------------------------------------------------------*/
```

```
#define UINT16    unsigned short

#define UINT8     unsigned char
#define GREG      0x0005  /* location of GREG Register  */

#define LOW_BYTE_MASK      0x00FF   /* Mask out upper byte keep low  */
#define HIGH_BYTE_MASK     0xFF00   /* Mask out lower byte keep high */

#define BYTE0        0x0000  /* Byte 0 is low byte  D0–D7      */
#define BYTE1        0x0001   /* Byte 1 is high byte D8–D15    */

/* Wait State Bits */

#define PSWSB0 0x0001
#define PSWSB1 0x0002
#define PSWSB2 0x0004
#define DSWSB0 0x0008
#define DSWSB1 0x0010
#define DSWSB2 0x0020
#define IOWSB0 0x0048
#define IOWSB1 0x0080
#define IOWSB2 0x0100


/*  Macro Expansion Declarations*/
#define _WSGR  0FFFFh /* wait state generator reg in I/O space */

#define STR(x) #x
#define OUTMAC(address,data)  \
 asm(”       LDPK    _”STR(data));  \
 asm(”       OUT     _”STR(data) ”,” STR(address))

#define INMAC(address,data)   \
 asm(”       LDPK    _”STR(data));  \
 asm(”       IN      _”STR(data) ”,” STR(address))
```

A.4 prot243.h

prot243.h declares the global routines that are used.

```
/*–––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/*  SPECTRUM DIGITAL INC.                                */
/*–––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/*  Prototype file                                       */
/*                                                       */
/*–––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/*  file name: prot243.h                                 */
/*  Started on:  02/11/1997    by mls                    */
/*  Last update: 02/18/2000    by rrp for 243            */
/*               06/11/2003    by Steve Sams for app demo */
/*–––––––––––––––––––––––––––––––––––––––––––––––––––––––*/

unsigned int dac_delay(volatile unsigned int);
void write_dac(unsigned int, unsigned int);

volatile unsigned int read_atod(unsigned int chan_num);
void atod_init(void);
void clear_fifos( void );
```

```
unsigned int dactest(void);

unsigned int wait_ms( volatile unsigned int delay_val );

void set_wait(void);
void bad_trap(void);
void test_init(void);
void error_process(unsigned int);
```

A.5 ioreg.h

ioreg.h declares the memory locations of the various control registers on the F243.

```
/*----------------------------------------------------------*/
/*  SPECTRUM DIGITAL INC.                                   */
/*----------------------------------------------------------*/
/*  I/O control registers include file                      */
/*                                                          */
/*----------------------------------------------------------*/
/*  file name: ioreg.h                                      */
/*  Started on:  02/11/1997    by mls                       */
/*  Last update: 02/18/2000    by rrp for 243               */
/*----------------------------------------------------------*/

/*
  I/O register addresses
*/
#define OCRA           0x7090
#define OCRB           0x7092
#define ISRA           0x7094
#define ISRB           0x7096

#define PADATDIR       0x7098

#define PBDATDIR       0x709A

#define PCDATDIR       0x709C

#define PDDATDIR       0x709E
/*
   OCRA & OCRB bit definitions
*/
#define OPA0           0x0001
#define TXD_FUNCTION   0x0001
#define OPA1           0x0002
#define RXD_FUNCTION   0x0002
#define OPA2           0x0004
#define OPA3           0x0008
#define OPA4           0x0010
#define OPA5           0x0020
#define OPA6           0x0040
#define OPA7           0x0080
#define OPA8           0x0100
#define OPA9           0x0200
#define OPA10          0x0400
#define OPA11          0x0800
#define OPA12          0x1000
#define OPA13          0x2000
#define OPA14          0x4000
```

```
#define OPA15          0x8000

#define OPB0           0x0001
#define OPB1           0x0002
#define OPB2           0x0004
#define OPB3           0x0008
#define OPB4           0x0010
#define OPB5           0x0020
#define OPB6           0x0040
#define OPB7           0x0080
#define OPB8           0x0100
#define OPB9           0x0200
#define OPB10          0x0400
#define OPB11          0x0800
#define OPB12          0x1000
#define OPB13          0x2000
#define OPB14          0x4000
#define OPB15          0x8000

/*
  IO register bit definitions
*/
#define IOPx0          0x0001
#define IOPx1          0x0002
#define IOPx2          0x0004
#define IOPx3          0x0008
#define IOPx4          0x0010
#define IOPx5          0x0020
#define IOPx6          0x0040
#define IOPx7          0x0080

#define x0DIR          0x0100
#define x1DIR          0x0200
#define x2DIR          0x0400
#define x3DIR          0x0800
#define x4DIR          0x1000
#define x5DIR          0x2000
#define x6DIR          0x4000
#define x7DIR          0x8000
```

A.6 dac.h

dac.h declares memory locations for ADC registers on the F243 and the locations in the I/O memory space where the DAC is located.

```
/*----------------------------------------------------------*/
/*  SPECTRUM DIGITAL INC.                                   */
/*----------------------------------------------------------*/
/*  Off chip parallel DAC include file                      */
/*                                                          */
/*----------------------------------------------------------*/
/*  file name: dac.h                                        */
/*  Started on:  02/11/1997    by mls                       */
/*  Last update: 02/18/2000    by rrp for 243               */
/*----------------------------------------------------------*/

/*
   Offchip DACT test include file for EVM320C240
```

```
     filename:    dac.h
   original:    02/26/97     by: mls
   last update: 02/26/97     by: mls
*/

#define DAC_DELAY_CNT  0x0200
#define LOW_BYTE_MASK  0x00FF
#define MIN_DAC_VAL    0x0000
/* #define MAX_DAC_VAL    0x0FFF  */     /* 12 bit, 0-4095 */
#define MAX_DAC_VAL    0x1000       /* 12 bit, 0-4095 */
#define MIN_ADC_VAL    0x0000
/* #define MAX_ADC_VAL    0x03FF */      /* 10 bit, 0-1023 */
#define MAX_ADC_VAL    0x0400

/*
  Write Only
*/
#define DAC0     0       /* 1st digital to analog converter */
#define DAC1     1       /* 2nd digital to analog converter */
#define DAC2     2       /* 3rd digital to analog converter */
#define DAC3     3       /* 4th digital to analog converter */
#define DAC_XFER  4      /* transfer data */

/*
  Macro Expansion Declarations
*/
#define _DAC_BASE  00000h

/*
  Write Only
*/
#define _DAC0     _DAC_BASE+0 /* 1st digital to analog converter  */
#define _DAC1     _DAC_BASE+1 /* 2nd digital to analog converter  */
#define _DAC2     _DAC_BASE+2 /* 3rd digital to analog converter  */
#define _DAC3     _DAC_BASE+3 /* 4th digital to analog converter  */
#define _DAC_XFER _DAC_BASE+4         /* transfer data */



/*
  2 channel A/D definitions
*/
/*********************************************************************/
/*                Dual 10-bit ADC Module for C240                  */
/*********************************************************************/
struct dual_adc
{
  unsigned int  adctrl1;          /* 0x7032 in data space */
  unsigned int  adcsp1;           /* 0x7033 in data space */
  unsigned int  adctrl2;          /* 0x7034 in data space */
  unsigned int  adcsp2;           /* 0x7035 in data space */
  unsigned int  adcfifo1;         /* 0x7036 in data space */
  unsigned int  adcsp3;           /* 0x7037 in data space */
  unsigned int  adcfifo2;         /* 0x7058 in data space */
};

/*********************************************************************/
/*  Define pointers to memory mapped peripherals                     */
/*********************************************************************/
```

```
#define ADCTRL1    0x07032
/*
  ADC Control Reg 1 at 0x7032 on C240
*/
#define ADCSOC      0x0001
#define ADC1CHSEL   0x000E
#define ADC2CHSEL   0x0070
#define ADCEOC      0x0080
#define ADCINTFLAG  0x0100
#define ADCINTEN    0x0200
#define ADCCONRUN   0x0400
#define ADC1EN     0x0800
#define ADC2EN     0x1000
#define ADCIMSTART  0x2000
#define ADC_FREE    0x4000
#define ADC_SOFT    0x8000

/*ADC Control Reg 2 at 0x7034 on C240*/
#define ADCTRL2       0x07034

#define ADCEXTSOC     0x0100
#define ADCEVSOC      0x0200

#define ADCFIFO1ST    0x0018
#define ADCFIFO2ST    0x00C0

#define ADC_PRESCALE1      0x0000
#define ADC_PRESCALE2      0x0001
#define ADC_PRESCALE4      0x0002
#define ADC_PRESCALE8      0x0003
#define ADC_PRESCALE12     0x0004
#define ADC_PRESCALE16     0x0005
#define ADC_PRESCALE24     0x0006
#define ADC_PRESCALE32     0x0007

/* FIFOS */

#define ADCFIFO1    0x7036
#define ADCFIFO2    0x7038
```

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265