Garmin Power Sensor Test Fixture

Final Report

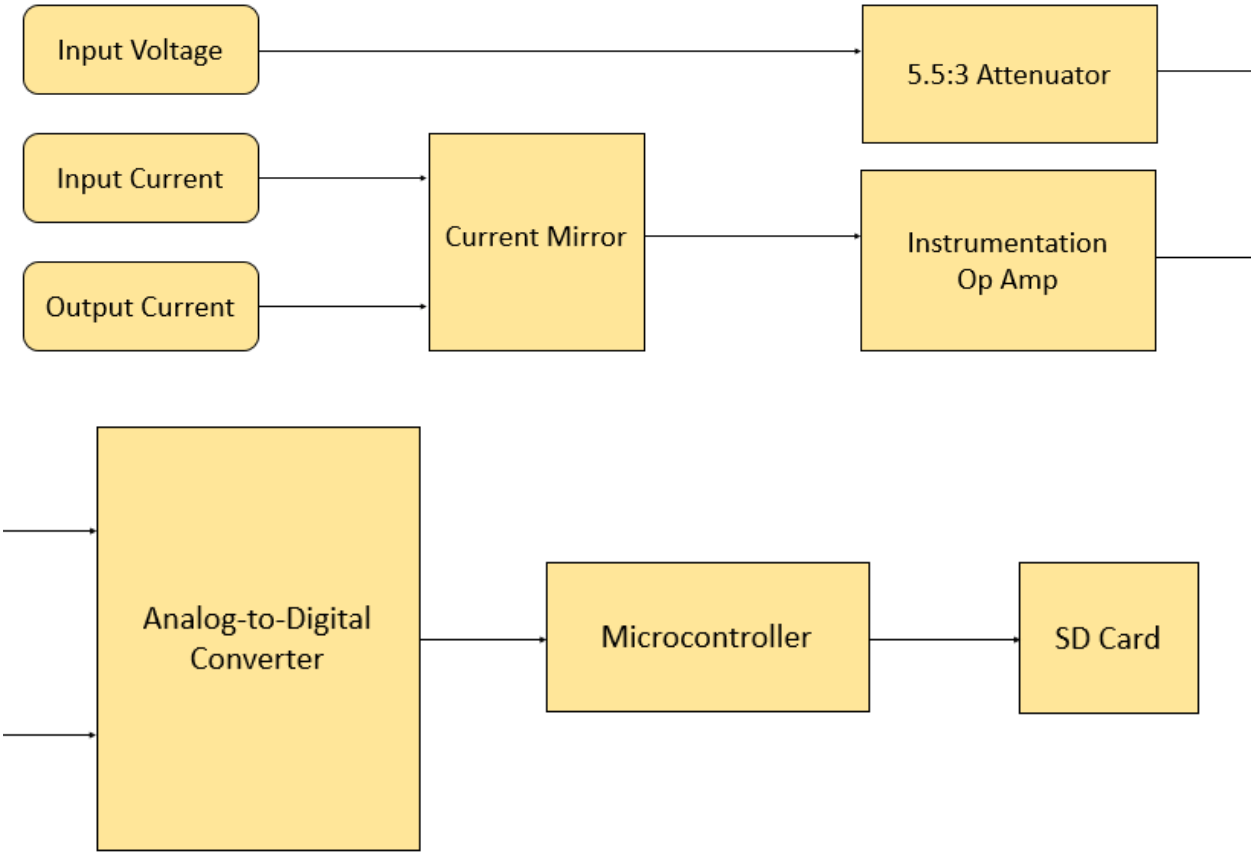
Group Number: May 1729
Client: Jeramie Vens
Team Leader: Brandon Floyd
Key Concept Holder: Amna Aftab
Key Concept Holder: Xi Zhu
Webmaster: Stephen Julich
Communication Leader: Francis Wagner
Adviser: Dr. Degang Chen
Team Website: http://may1729.sd.ece.iastate.edu

Contents

Introduction

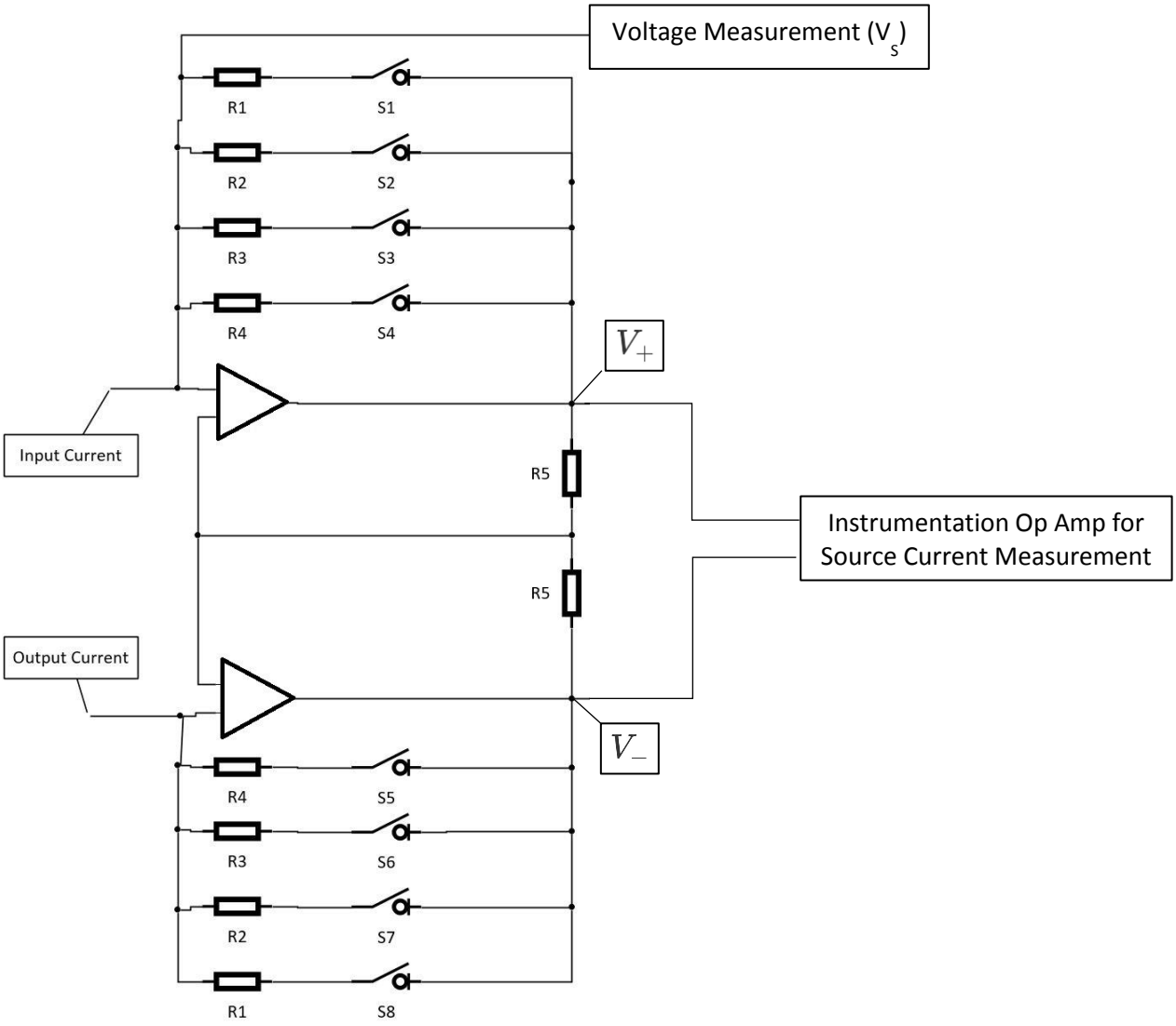
During product development at Garmin, engineers want to be able to measure the power consumed by parts of circuits within their fitness electronics. Our power sensor fixture has one channel with two probes capable of measuring currents from 10 µA to 10 mA and voltages from 0 to 5.5 V. We use a microcontroller to calculate power consumption with the aforementioned measurements, and an SD card will be capable of storing the data. Garmin engineers hope to hook up our fixture to different circuit sections of a gadget and let it run overnight to obtain a comprehensive set of power consumption data points.

Design Overview



Current Mirror Design

The current mirror functions as would an ammeter in that the circuit is broken to achieve the measurement and the output current is equal to the input current. From the input current, we can read voltage and current to calculate power.



$$V_+ = V_S - I_S \cdot R_F$$

$$V_- = V_S + I_S \cdot R_F$$

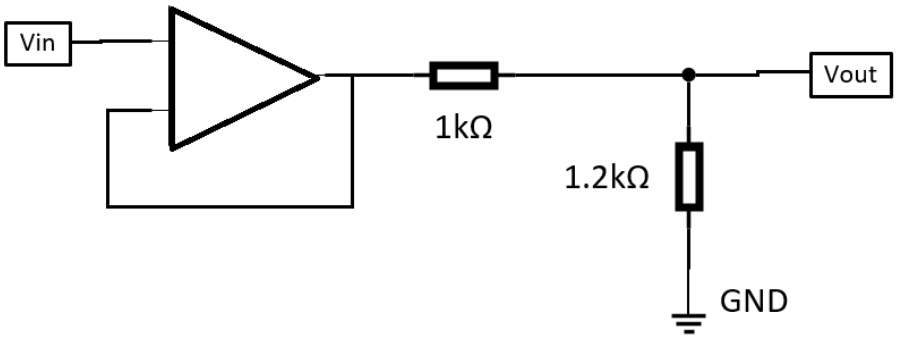
To accommodate the entire input current range of 10µA-10mA, we use four subranges determined by manual switches. S1/S8, S2/S7, S3/S6, and S4/S5 must be switched on for the desired range. The switches will be operated by engineers familiar with the device to be measured.

Range Min	**Range Max**	**Switches ON**	**Feedback Resistance**	**Precision**
0 A	10µA	S1/S8	20kΩ (R1)	+/- 0.01µA
0 A	100 µA	S2/S7	2kΩ (R2)	+/- 0.1µA
0 A	1mA	S3/S6	200Ω (R3)	+/- 1µA
0 A	10mA	S4/S5	20Ω (R4)	+/- 10µA

We chose R5 to be 2kΩ. While the value is somewhat arbitrary, 2kΩ optimized circuit performance given the feedback resistance values listed above. The source voltage is tapped by a voltage buffer followed by an attenuator. The attenuator scales the voltage for ADC input. V+ and V- are the inputs to the instrumentation op amp, and they are used to measure the source current.

Attenuator Design

The attenuator (voltage divider) scales voltage readings from 0-5.5V to 0-3V. The maximum source voltage specified by the client was 5.5 V, and the maximum voltage input to the ADC is 3V.



$$V_{out} = V_{in} \cdot \frac{R_1}{R_1 + R_2}$$

Texas Instruments Operational Amplifier OPA121

- Used for both the current mirror and attenuating sections of the circuit
- Typical input offset voltage of +/- 0.5V
- Typical input offset current of +/- 0.7pA
- Common mode input range of +/- 11V
- Open loop voltage gain of 114 dB

Texas Instruments Instrumentation Amplifier INA163

- Used for current measurement off of current mirror circuit
- Low noise, low distortion
- Gain can reach 100 dB if necessary
- Accurate and flexible

Equations relating INA readings to source voltage and source current:

$$V_{dm} = V_+ - V_- = 2I \cdot R_F$$

ADC Design

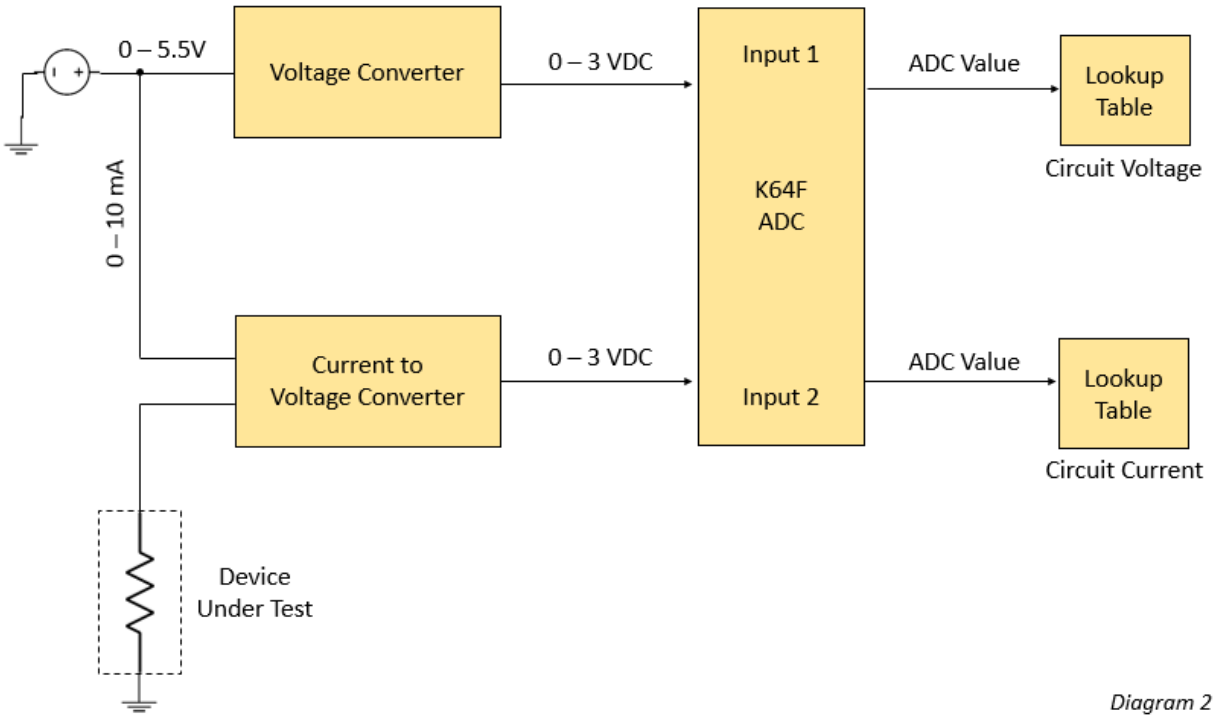


Diagram 2

Design satisfies the following criteria specified by the client:
- Current measured at a minimum rate of 10 ksamples/second
- Voltage measured at a minimum rate of 1 sample/second
- Unattended data logging for up to 16 hours

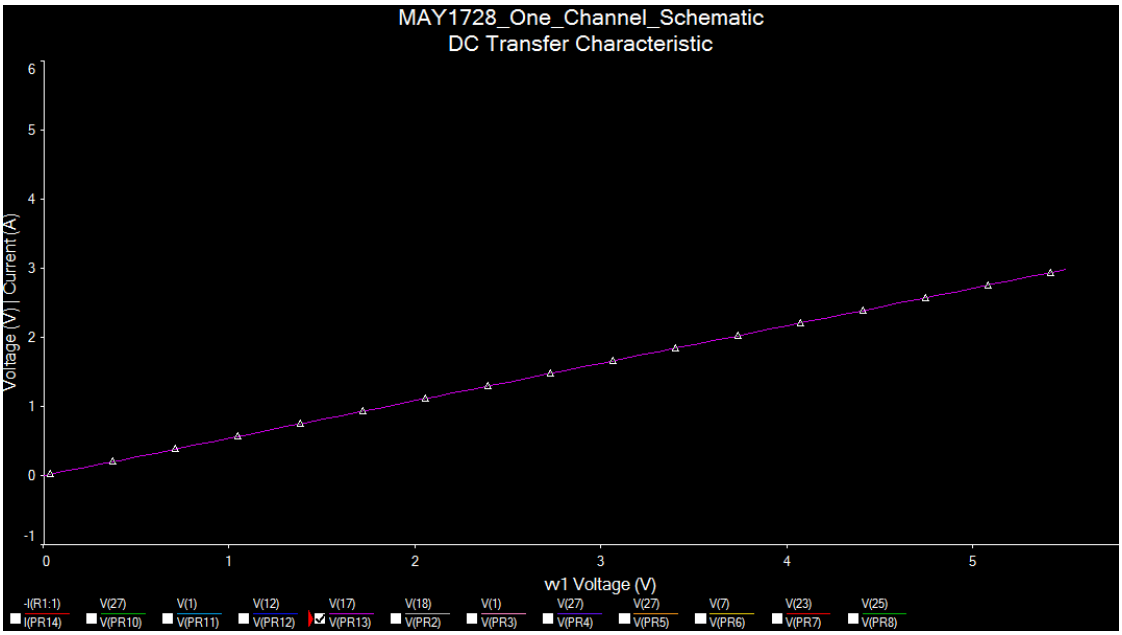
FDRM-K64F Development Board

- 120 MHz ARM Cortex-M4 Core
- 1 MB Flash Memory
- 256 KB RAM
- Two 16-bit SAR ADCs
- Micro SD Card Reader
- PCB plugin

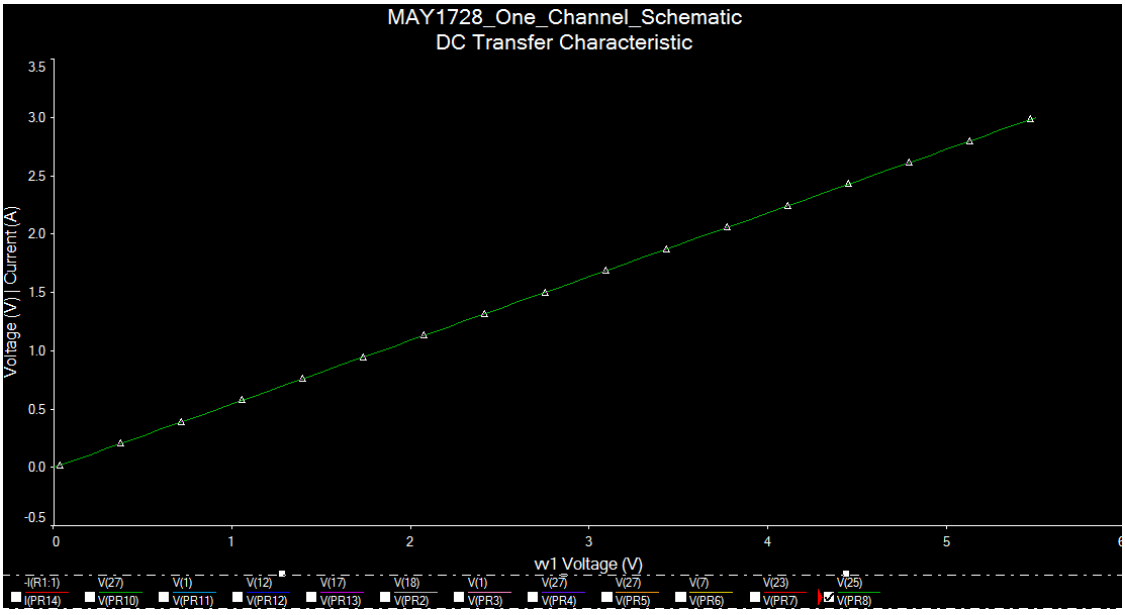
Implementation

Once the switches have been set based on the expected input current, the device is hooked up to the PBC by two probes – one to measure source current and one to measure voltage. The graphs below show the linear output based on a sweep of input voltage across the range specified by the client. The ADC will calculate the power based on these measurements and store them on an SD card.

Measured source current as a function of the input voltage



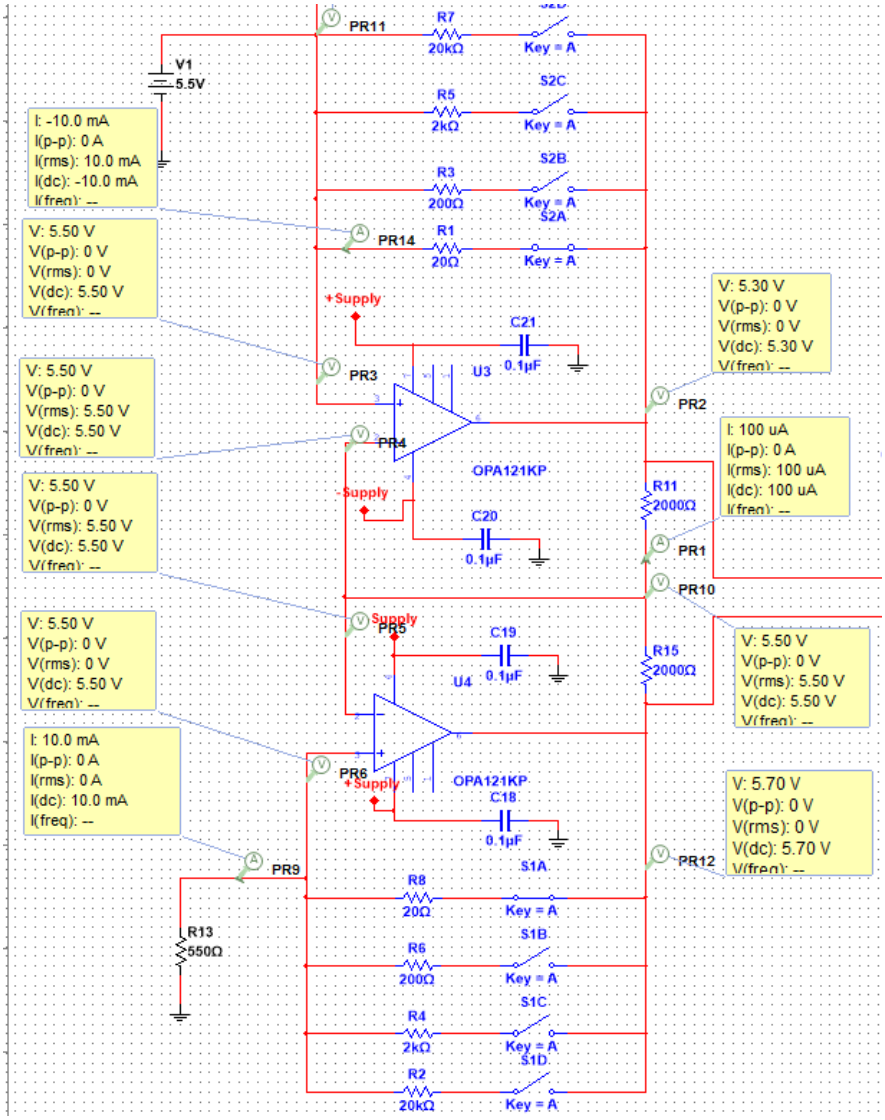
Measured source voltage as a function of the input voltage



Testing Process

Current Mirror Testing

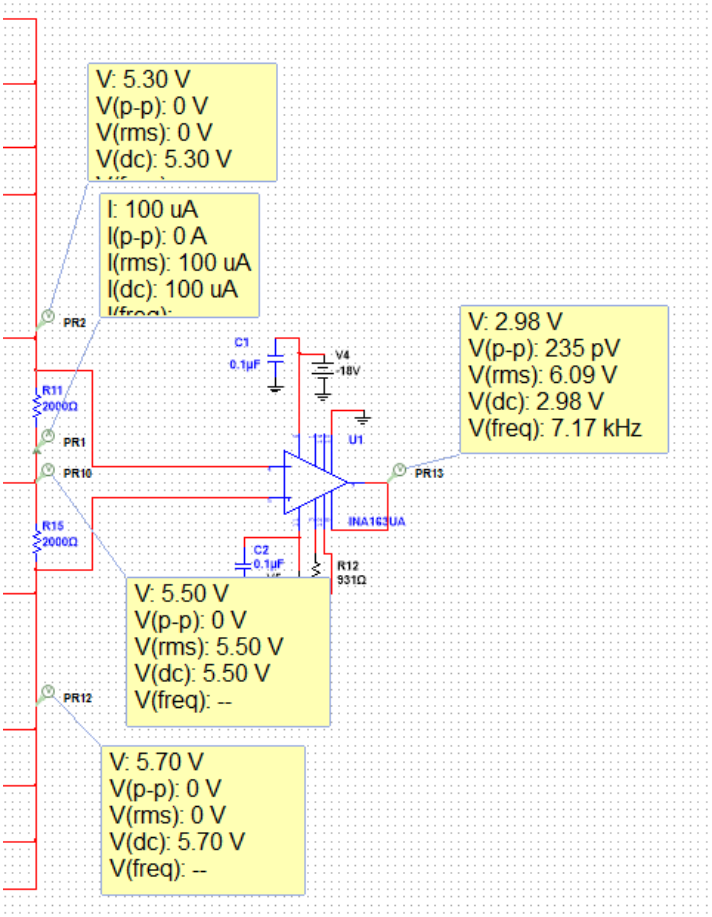
For the simulation shown below, we provided a 5.5V source voltage and a 10mA source current. Both values are the maximums specified by the client. The current falls in the highest range, so the switches are set so that the 20Ω resistors are the feedback resistors. A 550Ω resistor acts as the device under testing. As desired, the input current is equal to the output current.



Each OA121 has a +/- 10 V Supply and 0.1 µF decoupling capacitors.

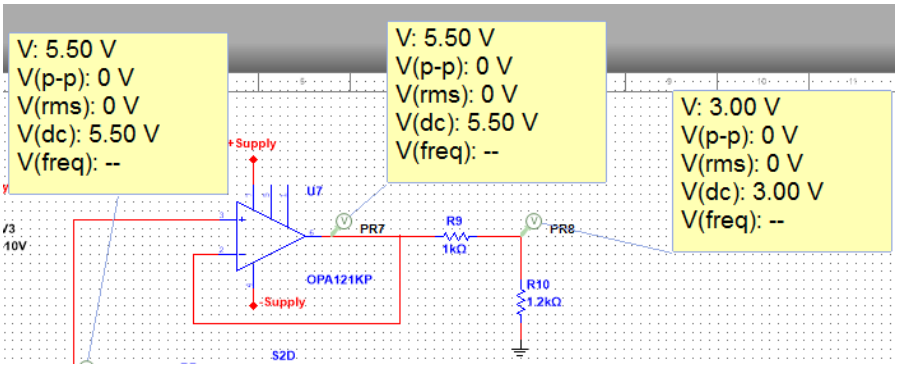
Instrumentation Op Amp (INA) Testing

The screenshot below shows the instrumentation op amp portion of the circuit under the same conditions of the current mirror simulation shown in the previous section. The INA has +/- 18V power supplies. As shown in the “Implementation” section, the output of the INA is linear with respect to the voltage input.



Attenuator Testing

The attenuator brings the maximum voltage input to the maximum ADC input. The relationship between the input and output voltage is linear.



Appendix 1: Operation Manual

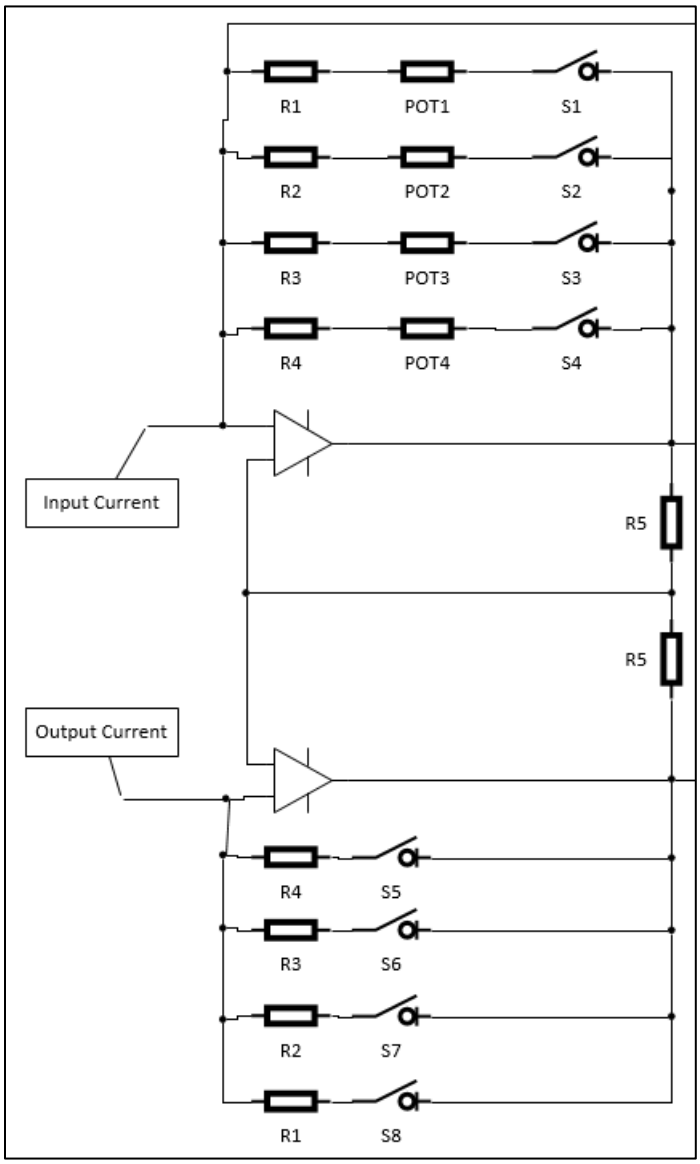
1. A +/- 10V power supply is needed to test the circuit.
2. The device needs a test device for it to monitor.
3. Connect the test device to the main device
4. Connect the +10V and -10V power supply cables to the +Supply and -Supply reference nodes respectively.
5. An oscilloscope and multimeter will be needed to view the results of the experiment.
6. Connect cables from the oscilloscope to both the input and output nodes of the circuit and measure the current
7. Ensure that the input current matches the output current
8. Connect cables from the oscilloscope to the input node and the critical current mirror nodes
9. Ensure that the ADC input voltage is linearly proportional to voltage readings at the critical current mirror modes
10. Connect cables from the multimeter to the input node
11. Read the input capacitance with the multimeter and ensure that the ADC input capacitance does not exceed 5 pF
12. When checking other components, continuously monitor the input and output voltages and ensure that they match continually. The current of the test circuit being measured cannot be disrupted.

Appendix 2: Alternate/Initial Designs

The non-destructive current measurement was the focus of the design process. For this reason, the current mirror circuit underwent the most changes.

Current Mirror with Potentiometers

In order for the current mirror to function properly, it is important for the feedback resistors to be equal. To achieve this, we could pay for high accuracy resistors or include potentiometers to be adjusted until the resistances are sufficiently equal. In the end, we chose the former method. If we had chosen the design shown below, the potentiometers would have to be adjusted only once for each range. We went without the potentiometers because they complicate the manual setup. Also, we were willing to pay for the high accuracy resistors.

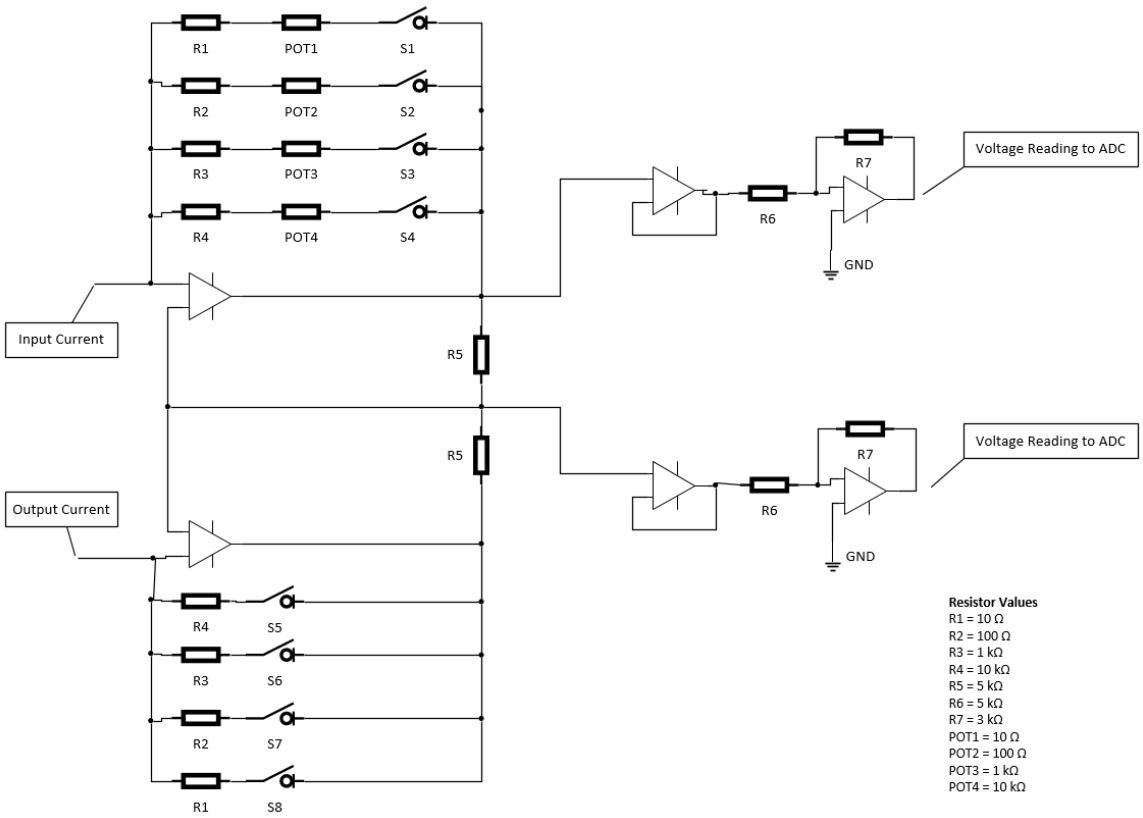


Eight Channels to One Channel

Our client wanted to have eight channels (two probes for each) capable of measuring power for eight devices simultaneously. To simplify and focus our design, we revised the project scope to one channel. Once the first channel functions properly, the replication of channels is trivial. The scope revision also lowered projects costs significantly.

Two Voltage Readings Instead of Instrumentation Op Amp

Initially, we favored this design over an instrumentation op amp because we preferred its apparent simplicity. The two voltage readings shown below would have been used to calculate the source current, but the source voltage measurement became more difficult with this design. A third voltage measurement at the input (as in our final design) would have done the trick, but three inputs per channel were too many for the ADC.



Appendix III: Other Considerations

We did not practice solid project management skills, and our design suffered as a result of it.

Appendix IV (Code)

```
/*************************************************************
 * Senior Design Project MAY1729
 *
 * 8 channel ADC - 10k samples per second to SD card
 *
 * Stephen Julich v1.0 26 May 2017
 *
 * Code contains a mix of bare metal register programming, functions
 * from the SDK 2.0 API, and sample/demo code from NXP and Freescale
 *
 *************************************************************/

/*************************ToDo*********************************
Programmable Interrupt Timers
DMA Transfers for ADC?
*************************************************************/

/*
 * Copyright (c) 2015, Freescale Semiconductor, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * o Redistributions of source code must retain the above copyright notice, this list
 *   of conditions and the following disclaimer.
 *
 * o Redistributions in binary form must reproduce the above copyright notice, this
 *   list of conditions and the following disclaimer in the documentation and/or
 *   other materials provided with the distribution.
 *
 * o Neither the name of Freescale Semiconductor, Inc. nor the names of its
 *   contributors may be used to endorse or promote products derived from this
 *   software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "fsl_debug_console.h"
#include "board.h"
#include "fsl_adc16.h"
#include "fsl_dac.h"
#include "stdio.h"
#include "clock_config.h"
#include "pin_mux.h"
#include "fsl_rtc.h"
```

```
#include "inttypes.h"
#include "fsl_mpu.h"
#include <string.h>
#include "fsl_uart.h"
#include "fsl_gpio.h"
#include "ff.h"
#include "diskio.h"
#include "fsl_pit.h"

/*******************************************************************************
 * Definitions
 ******************************************************************************/

//Define ADC0,1 Base Addresses
#define DEMO_ADC16_BASEADDR_0 ADC0
#define DEMO_ADC16_BASEADDR_1 ADC1

//Define ADC channel groups
#define DEMO_ADC16_CHANNEL_GROUPA 0U
#define DEMO_ADC16_CHANNEL_GROUPB 1U

//Define ADC channels
#define DEMO_ADC16_USER_CHANNEL0 0U /* PTB2, ADC0_SE12, J4-2 */
#define DEMO_ADC16_USER_CHANNEL1 1U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL3 3U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL4 4U /* PTB2, ADC0_SE12, J4-2 */
#define DEMO_ADC16_USER_CHANNEL5 5U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL12 12U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL13 13U /* PTB2, ADC0_SE12, J4-2 */
#define DEMO_ADC16_USER_CHANNEL14 14U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL15 15U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL18 18U /* PTB2, ADC0_SE12, J4-2 */
#define DEMO_ADC16_USER_CHANNEL19 19U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_ADC16_USER_CHANNEL20 20U /* PTC1, ADC0_SE15, J1-5 */
#define DEMO_DAC_BASEADDR DAC0

//Define ADC0,1 interrupts
#define DEMO_ADC16_IRQn_0 ADC0_IRQn
#define DEMO_ADC16_IRQn_1 ADC1_IRQn

//Define ADC0,1 IRQ handlers
#define DEMO_ADC16_IRQ_HANDLER_FUNC0 ADC0_IRQHandler
#define DEMO_ADC16_IRQ_HANDLER_FUNC1 ADC1_IRQHandler

//Define DAC voltage output values
#define DAC_1_0_VOLTS 310U
#define DAC_1_5_VOLTS 466U
#define DAC_2_0_VOLTS 621U
#define DAC_2_5_VOLTS 776U
#define DAC_3_0_VOLTS 931U

//Define ADC ref voltage
#define VREF_BRD 3.300

//Define ADC bits/sample
#define SE_12BIT 4096.0
#define SE_10BIT 1024.0
```

```
//Define channels per ADC
#define NUM_CHANNELS 8

/* Define buffer size (in bytes) for read/write operations */
#define BUFFER_SIZE (5000U)


/*******************************************************************************
 * Prototypes
 ******************************************************************************/
// Initialize ADC & DAC
static void Init_DAC(void);
static void Init_ADC(void);

//Get ADC values from ADC 0
static void Get_Conversion0(void);
//Get ADC values from ADC 1
static void Get_Conversion1(void);

//Enable measurement chanels
static void Get_Active_Channels(void);

//get console input
static void ReceiveFromConsole(char *buf, uint32_t size);

//Start the Real Time Clock
static void RTC_Start(void);

//Set clock source for RTC
static void BOARD_SetRtcClockSource(void);

//User menu to select ADC mode as single or continuous (timed)
static void Mode_Set(void);

//Choose DAC output voltage (used for testing ADC inputs)
static void DAC_Set(void);

//Program startup screen display
static void Init_Screen(void);

//First level user menu
static void Init_Menu(void);

//Write voltage conversion values to the buffer
static void Get_Voltage_Conversions(void);

//Write current conversion values to the buffer
static void Get_Current_Conversions(void);

//Print last voltage and curent conversion values to screen
static void Print_Conversions(void);

//Prompt user for measurement interval
static void Get_Interval(void);

//Set RTC alarm to measurement interval
static void Set_Interval(void);
```

```
//Print total samples and samples per channel
static void Print_Num_Conversions(void);

//Count the number of active channels
static uint8_t Channel_Count(void);

//Time delay in ms
static void delay(uint32_t milliseconds);

//Initialize the SD card
static void Init_SD(void);


/*******************************************************************************
 * Variables
 ******************************************************************************/

//ADC
volatile bool g_Adc16ConversionDoneFlag = false;
volatile uint16_t g_Adc16ConversionValue = 0;
static adc16_channel_config_t g_adc16ChannelConfigStruct0;
static adc16_channel_config_t g_adc16ChannelConfigStruct1;
static uint8_t g_active_channels;
static uint32_t g_ADC_Channel_Group = 0U;
static uint32_t g_ADC_Mode;
static uint8_t g_range_setting;
static uint32_t g_voltage_values[2][NUM_CHANNELS];
static uint32_t g_current_values[2][NUM_CHANNELS];
static uint64_t g_num_conversions = 0;
static uint8_t g_conversion_exit;

//init as const in real tables
static uint16_t lookup_range0[1024];
static uint16_t lookup_range1[1024];
static uint16_t lookup_range2[1024];
static uint16_t lookup_range3[1024];

//RTC
static volatile uint8_t g_AlarmPending = 0U;
static uint16_t g_hours;
static uint16_t g_minutes;
static uint16_t g_seconds;
static rtc_config_t rtcConfig;
static rtc_datetime_t g_date;
static rtc_datetime_t g_alarm_date;
static uint64_t g_total_alarm_seconds = 0;

//SD card reader
static FATFS g_fileSystem; /* File system object */
static FIL g_fileObject;   /* File object */
static uint8_t g_bufferWrite[BUFFER_SIZE]; /* Write buffer */
static uint32_t g_bufferWriteSize; //current size of buffer

/*******************************************************************************
 * Code
 ******************************************************************************/

//Initialize the DAC to provide test voltage for the ADC
```

```
static void Init_DAC(void) {

  dac_config_t dacConfigStruct;

        DAC_GetDefaultConfig(&dacConfigStruct);
        DAC_Init(DEMO_DAC_BASEADDR, &dacConfigStruct);
        DAC_Enable(DEMO_DAC_BASEADDR, true); /* Enable output. */
}

//Initialize the ADC and do calibration
static void Init_ADC(void) {

  adc16_config_t adc16ConfigStruct0;
  adc16_config_t adc16ConfigStruct1;

  ADC16_GetDefaultConfig(&adc16ConfigStruct0);
  ADC16_GetDefaultConfig(&adc16ConfigStruct1);
  adc16ConfigStruct0.clockSource = kADC16_ClockSourceAlt2;
  adc16ConfigStruct1.clockSource = kADC16_ClockSourceAlt2;
  adc16ConfigStruct0.clockDivider = kADC16_ClockDivider1;
  adc16ConfigStruct1.clockDivider = kADC16_ClockDivider1;
  adc16ConfigStruct0.enableHighSpeed = true;
  adc16ConfigStruct1.enableHighSpeed = true;
  adc16ConfigStruct0.resolution = kADC16_ResolutionSE10Bit;
  adc16ConfigStruct1.resolution = kADC16_ResolutionSE10Bit;

  ADC16_Init(DEMO_ADC16_BASEADDR_0, &adc16ConfigStruct0);
  ADC16_Init(DEMO_ADC16_BASEADDR_1, &adc16ConfigStruct1);

  ADC16_SetHardwareAverage (DEMO_ADC16_BASEADDR_0,    kADC16_HardwareAverageDisabled );
  ADC16_SetHardwareAverage (DEMO_ADC16_BASEADDR_1,    kADC16_HardwareAverageDisabled );

  /* Make sure the software trigger is used. */
  ADC16_EnableHardwareTrigger(DEMO_ADC16_BASEADDR_0, false);
  ADC16_EnableHardwareTrigger(DEMO_ADC16_BASEADDR_1, false);

        #if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
                if (kStatus_Success == ADC16_DoAutoCalibration(DEMO_ADC16_BASEADDR_0))
                {
                        PRINTF("\r\nADC16_DoAutoCalibration() Done for ADC 0.");
                }
                else
                {
                        PRINTF("ADC16_DoAutoCalibration() Failed for ADC 0.\r\n");
                }
                if (kStatus_Success == ADC16_DoAutoCalibration(DEMO_ADC16_BASEADDR_1))
                {
                        PRINTF("\r\nADC16_DoAutoCalibration() Done for ADC 1.");
                }
                else
                {
                        PRINTF("ADC16_DoAutoCalibration() Failed for ADC 1.\r\n");
                }
        #endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION */
}

//IRQ handler for ADC0
void DEMO_ADC16_IRQ_HANDLER_FUNC0(void)
```

```
{
  g_Adc16ConversionDoneFlag = true;
  /* Read conversion result to clear the conversion completed flag. */
  g_Adc16ConversionValue = ADC16_GetChannelConversionValue(DEMO_ADC16_BASEADDR_0, g_ADC_Channel_Group);
}

//IRQ handler for ADC1
void DEMO_ADC16_IRQ_HANDLER_FUNC1(void)
{
  g_Adc16ConversionDoneFlag = true;
  /* Read conversion result to clear the conversion completed flag. */
  g_Adc16ConversionValue = ADC16_GetChannelConversionValue(DEMO_ADC16_BASEADDR_1, g_ADC_Channel_Group);
}

//IRQ handler for RTC
void RTC_IRQHandler(void)
{
  if (RTC_GetStatusFlags(RTC) & kRTC_AlarmFlag)
  {
    g_AlarmPending = 1U;

    /* Clear alarm flag */
    RTC_ClearStatusFlags(RTC, kRTC_AlarmInterruptEnable);
  }
}

//Get voltage conversions from ADC0
static void Get_Conversion0(void)
{
 // float voltRead;
        g_Adc16ConversionDoneFlag = false;


            g_adc16ChannelConfigStruct0.enableInterruptOnConversionCompleted = true;

          #if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
            g_adc16ChannelConfigStruct0.enableDifferentialConversion = false;
          #endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */

            ADC16_SetChannelConfig(DEMO_ADC16_BASEADDR_0, g_ADC_Channel_Group,
&g_adc16ChannelConfigStruct0);
            //ADC16_SetChannelMuxMode(DEMO_ADC16_BASEADDR_1,kADC16_ChannelMuxB);

            while (!g_Adc16ConversionDoneFlag)
            {
            }
            g_num_conversions++;
}

//Get current conversions from ADC1
static void Get_Conversion1(void)
{
  //float voltRead;
        g_Adc16ConversionDoneFlag = false;

            g_adc16ChannelConfigStruct1.enableInterruptOnConversionCompleted = true;

          #if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
```

```
            g_adc16ChannelConfigStruct1.enableDifferentialConversion = false;
        #endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */

            ADC16_SetChannelConfig(DEMO_ADC16_BASEADDR_1, g_ADC_Channel_Group,
&g_adc16ChannelConfigStruct1);
            //ADC16_SetChannelMuxMode(DEMO_ADC16_BASEADDR_1,kADC16_ChannelMuxB);

            while (!g_Adc16ConversionDoneFlag)
            {
            }
            g_num_conversions++;
}

//Specify measurement channels to be enabled
static void Get_Active_Channels(void) {

  int loop_exit = 0;

      while(!loop_exit) {
            PRINTF("\r\nPlease enter the channel to be enabled (1-8), 9 for all, or 0 to exit.\r\n--> ");
            uint8_t msg = ' ';
            while ((msg < '0') || (msg > '9'))
            {
                  msg = GETCHAR();
                  PUTCHAR(msg);
                  PUTCHAR('\b');
            }
            switch (msg) {
                  case '1':
                        if(g_active_channels & 0x01) {
                              g_active_channels = g_active_channels & 0xFE;
                              PRINTF("\r\nChannel %c disabled\r\n", msg);
                        }
                        else {
                              g_active_channels = g_active_channels | 0x01;
                              PRINTF("\r\nChannel %c enabled\r\n", msg);
                        }
                        break;
                  case '2':
                        if(g_active_channels & 0x02) {
                              g_active_channels = g_active_channels & 0xFD;
                              PRINTF("\r\nChannel %c disabled\r\n", msg);
                        }
                        else {
                              g_active_channels = g_active_channels | 0x02;
                              PRINTF("\r\nChannel %c enabled\r\n", msg);
                        }
                        break;
                  case '3':
                        if(g_active_channels & 0x04) {
                              g_active_channels = g_active_channels & 0xFB;
                              PRINTF("\r\nChannel %c disabled\r\n", msg);
                        }
                        else {
                              g_active_channels = g_active_channels | 0x04;
                              PRINTF("\r\nChannel %c enabled\r\n", msg);
                        }
                        break;
```

```
case '4':
		if(g_active_channels & 0x08) {
			g_active_channels = g_active_channels & 0xF7;
			PRINTF("\r\nChannel %c disabled\r\n", msg);
		}
		else {
			g_active_channels = g_active_channels | 0x08;
			PRINTF("\r\nChannel %c enabled\r\n", msg);
		}
		break;
case '5':
		if(g_active_channels & 0x10) {
			g_active_channels = g_active_channels & 0xEF;
			PRINTF("\r\nChannel %c disabled\r\n", msg);
		}
		else {
			g_active_channels = g_active_channels | 0x10;
			PRINTF("\r\nChannel %c enabled\r\n", msg);
		}
		break;
case '6':
		if(g_active_channels & 0x20) {
			g_active_channels = g_active_channels & 0xDF;
			PRINTF("\r\nChannel %c disabled\r\n", msg);
		}
		else {
			g_active_channels = g_active_channels | 0x20;
			PRINTF("\r\nChannel %c enabled\r\n", msg);
		}
		break;
case '7':
		if(g_active_channels & 0x40) {
			g_active_channels = g_active_channels & 0xBF;
			PRINTF("\r\nChannel %c disabled\r\n", msg);
		}
		else {
			g_active_channels = g_active_channels | 0x40;
			PRINTF("\r\nChannel %c enabled\r\n", msg);
		}
		break;
case '8':
		if(g_active_channels & 0x80) {
			g_active_channels = g_active_channels & 0x7F;
			PRINTF("\r\nChannel %c disabled\r\n", msg);
		}
		else {
			g_active_channels = g_active_channels | 0x80;
			PRINTF("\r\nChannel %c enabled\r\n", msg);
		}
		break;
case '9':
		if(g_active_channels & 0xFF) {
			g_active_channels = g_active_channels & 0x00;
			PRINTF("\r\nAll 8 Channels disabled\r\n");
		}
		else {
			g_active_channels = g_active_channels | 0xFF;
			PRINTF("\r\nAll 8 Channels enabled\r\n");
```

```
                                        }
                                        break;
                                case '0':
                                        loop_exit = 1;
                                        break;
                                default:
                                        PRINTF("\r\nPlease input a valid number: 1-8 \r\n");
                                        break;
                        }
                }
}

//Set clock source for RTC
static void BOARD_SetRtcClockSource(void)
{
  /* Enable the RTC 32KHz oscillator */
  RTC->CR |= RTC_CR_OSCE_MASK;
}

//Start the RTC
static void RTC_Start(void) {

        /* Init RTC */
        RTC_GetDefaultConfig(&rtcConfig);
        RTC_Init(RTC, &rtcConfig);
        /* Select RTC clock source */
        BOARD_SetRtcClockSource();
        RTC_StopTimer(RTC);
 /* Enable at the NVIC */
        EnableIRQ(RTC_IRQn);
        RTC_StartTimer(RTC);
        RTC_GetDatetime(RTC, &g_date);
        /* print default time */
        printf("\r\nCurrent datetime: %04d-%02d-%02d %02d:%02d:%02d\r\n", g_date.year, g_date.month, g_date.day,
g_date.hour, g_date.minute, g_date.second);
  PRINTF("\r\n");
}

//Get user keyboard input
static void ReceiveFromConsole(char *buf, uint32_t size)
{
  uint32_t i;

  for (i = 0U; i < size; i++)
  {
    buf[i] = GETCHAR();
    PUTCHAR(buf[i]);
  }
}

//Set the RTC time
static void RTC_Set(void) {

        uint16_t year = 2000U;
                uint16_t month = 1U;
                uint16_t day = 1U;
                uint16_t hour = 0U;
                uint16_t min = 0U;
```

```
                    uint16_t sec = 0U;

                     uint8_t msg;

                    /* Determine what to do next based on user's request */
                        PRINTF("\r\n\t1. Set RTC date and time.\r\n\t2. Accept default date and time.\r\n-->");
                        msg = ' ';
                        while ((msg < '1') || (msg > '2'))
                        {
                           msg = GETCHAR();
                           PUTCHAR(msg);
                           PUTCHAR('\b');
                        }


                        /* Set next state */
                        if (msg == '1') {

                              static char StrNewline[] = "\r\n";
                              static char StrInvalid[] = "Invalid input format\r\n";
                              char recvBuf[20];
                              uint32_t result;
                              PRINTF("\r\nInput date time like: \"2010-10-10 10:10:10\"\r\n");
                              ReceiveFromConsole(recvBuf, 19U);
                              result = sscanf(recvBuf, "%04hd-%02hd-%02hd %02hd:%02hd:%02hd", &year, &month, &day,
&hour, &min,
                                                &sec);
                                    PRINTF("%s", StrNewline);
                                    /* Error message will appear when user enters the invalid g_date time */
                                    if (result != 6U)
                                    {
                                       PRINTF(StrInvalid);
                                    }
                                    else {
                                                                                    g_date.year = (uint16_t)year;
                                                                                    g_date.month =
(uint8_t)month;
                                                                                    g_date.day = (uint8_t)day;
                                                                                    g_date.hour = (uint8_t)hour;
                                                                                    g_date.minute =
(uint8_t)min;
                                                                                    g_date.second =
(uint8_t)sec;
                                    }
}
                          /* RTC time counter has to be stopped before setting the g_date & time in the TSR register */
                          RTC_StopTimer(RTC);
                          /* Set RTC time to default */
                          RTC_SetDatetime(RTC, &g_date);
                          /* Enable RTC alarm interrupt */
                          //RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);
                          /* Enable at the NVIC */
                          EnableIRQ(RTC_IRQn);
                          /* Start the RTC time counter */
                          RTC_StartTimer(RTC);
                          /* Get date time */
                          RTC_GetDatetime(RTC, &g_date);
                          /* print default time */
```

```
                                printf("Current datetime: %04d-%02d-%02d %02d:%02d:%02d\r\n", g_date.year, g_date.month,
g_date.day, g_date.hour, g_date.minute, g_date.second);
                                 //PRINTF("Current year: %d", date.year);
}

//User menu to set the DAC output voltage
static void DAC_Set(void) {

   int menu_exit = 0;
            uint8_t msg = ' ';


   while(!menu_exit) {

            PRINTF("\r\n\r\nSelect DAC output level:\r\n\t1. 1.0 V\r\n\t2. 1.5 V\r\n\t3. 2.0 V\r\n\t4. 2.5 V\r\n\t5. 3.0 V\r\n-->");
                        msg = ' ';
                        msg = GETCHAR();
                        PUTCHAR(msg);
                        switch (msg) {
                                    case '1':
                                                DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, DAC_1_0_VOLTS);
                                                menu_exit = 1;
                                                break;
                                    case '2':
                                                DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, DAC_1_5_VOLTS);
                                                menu_exit = 1;
                                                break;
                                    case '3':
                                                DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, DAC_2_0_VOLTS);
                                                menu_exit = 1;
                                                break;
                                    case '4':
                                                DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, DAC_2_5_VOLTS);
                                                menu_exit = 1;
                                                break;
                                    case '5':
                                                DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, DAC_3_0_VOLTS);
                                                menu_exit = 1;
                                                break;
                                    default:
                                                PRINTF("\r\nPlease input a valid number: 1-5 \r\n");
                                                break;
                        }
   }
}

//User menu to select ADC mode as single or continuous (timed)
static void Mode_Set(void) {

             uint8_t msg = ' ';
             PRINTF("\r\nSelect ADC Measurement Mode:\r\n\t1. Single Measurement\r\n\t2. Continuous Measurement\r\n--
>");
            while ((msg < '1') || (msg > '2')) {

                        msg = GETCHAR();
                        PUTCHAR(msg);
                        PUTCHAR('\b');
            }
```

```
		/* Set next state */
		if (msg == '1') {
			g_ADC_Mode = 1;
			PRINTF("\r\nSingle Measurement Mode Selected\r\n");
		}
		if (msg == '2') {
			g_ADC_Mode = 2;
			PRINTF("\r\nInterval Measurement Mode Selected\r\n");
			Get_Interval();
		}
}

//Program startup screen display
static void Init_Screen(void) {

		PRINTF("\r\n**************************************************************************");
   PRINTF("\r\n*        Iowa State Senior Design Project MAY1729              *");
   PRINTF("\r\n*                                                              *");
   PRINTF("\r\n*              8-Channel Power Sensor                          *");
   PRINTF("\r\n*                                                              *");
   PRINTF("\r\n* Amna Aftab, Brandon Floyd, Stephen Julich, Francis Wagner, Xi Zhu *");
   PRINTF("\r\n*                                                              *");
   PRINTF("\r\n*              Fall 2016/Spring 2017                           *");
   PRINTF("\r\n**************************************************************************\r\n");
   PRINTF("\r\n");
   PRINTF("\r\nPress any key to start...\r\n");
   GETCHAR();
}

//First level user menu
static void Init_Menu(void) {
		uint8_t msg = ' ';
   int menu_exit = 0;
   while(!menu_exit) {

			PRINTF("\r\n\r\nChoose an option:\r\n\t1. Set the RTC \r\n\t2. Set the DAC \r\n\t3. Enable ADC
channels\r\n\t4. Choose ADC mode\r\n\t5. Continue \r\n--> ");
			msg = GETCHAR();
			PUTCHAR(msg);
			switch (msg) {
				case '1':
					RTC_Set();
					break;
				case '2':
					DAC_Set();
					break;
				case '3':
					Get_Active_Channels();
					break;
				case '4':
					Mode_Set();
					break;
				case '5':
					if(g_active_channels != 0) {
						menu_exit = 1;
					}
					else {
```

```
                                        PRINTF("\r\nNo channels enabled. Please select at least 1 channel. \r\n");
                                }
                                break;
                        default:
                                PRINTF("\r\nPlease input a valid number: 1-5 \r\n");
                                break;
                }
  }
  menu_exit = 0;
}


static void Get_Voltage_Conversions(void) {

                /* Get conversions from enabled voltage channels */
                if(g_active_channels & 0x01) {
                        g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL0;
                        ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxA);
                        Get_Conversion0();

                        if(g_range_setting == 0) {
                                                g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 1) {
                                                g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 2) {
                                                g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 3) {
                                g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
                        }

                        uint8_t lower = g_Adc16ConversionValue & 0x00ff;
                        uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
                        g_bufferWrite[g_bufferWriteSize] = upper;
                        g_bufferWriteSize++;
                        g_bufferWrite[g_bufferWriteSize] = lower;
                        g_bufferWriteSize++;

                        g_voltage_values[0][0] = g_Adc16ConversionValue;
                        g_voltage_values[1][0] = g_range_setting;
                }

                if(g_active_channels & 0x02) {
                        g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL1;
                        ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxA);
                        Get_Conversion0();

                        if(g_range_setting == 0) {
                                                g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 1) {
```

```
                                                g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 2) {
                                                g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 3) {
                                g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
                        }

                        uint8_t lower = g_Adc16ConversionValue & 0x00ff;
                        uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
                        g_bufferWrite[g_bufferWriteSize] = upper;
                        g_bufferWriteSize++;
                        g_bufferWrite[g_bufferWriteSize] = lower;
                        g_bufferWriteSize++;

                        g_voltage_values[0][1] = g_Adc16ConversionValue;
                        g_voltage_values[1][1] = g_range_setting;
                }

                if(g_active_channels & 0x04) {
                        g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL3;
                        ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxA);
                        Get_Conversion0();

                        if(g_range_setting == 0) {
                                                g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 1) {
                                                g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 2) {
                                                g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 3) {
                                g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
                        }

                        uint8_t lower = g_Adc16ConversionValue & 0x00ff;
                        uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
                        g_bufferWrite[g_bufferWriteSize] = upper;
                        g_bufferWriteSize++;
                        g_bufferWrite[g_bufferWriteSize] = lower;
                        g_bufferWriteSize++;

                        g_voltage_values[0][2] = g_Adc16ConversionValue;
                        g_voltage_values[1][2] = g_range_setting;
                }

                if(g_active_channels & 0x08) {
                        g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL4;
                        ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxB);
                        Get_Conversion0();
```

```
                        if(g_range_setting == 0) {
                                                g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 1) {
                                                g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 2) {
                                                g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 3) {
                                g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
                        }

                        uint8_t lower = g_Adc16ConversionValue & 0x00ff;
                        uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
                        g_bufferWrite[g_bufferWriteSize] = upper;
                        g_bufferWriteSize++;
                        g_bufferWrite[g_bufferWriteSize] = lower;
                        g_bufferWriteSize++;

                        g_voltage_values[0][3] = g_Adc16ConversionValue;
                        g_voltage_values[1][3] = g_range_setting;
                }

                if(g_active_channels & 0x10) {
                        g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL5;
                        ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxB);
                        Get_Conversion0();

                        if(g_range_setting == 0) {
                                                g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 1) {
                                                g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 2) {
                                                g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
                                        }
                        else if(g_range_setting == 3) {
                                g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
                        }

                        uint8_t lower = g_Adc16ConversionValue & 0x00ff;
                        uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
                        g_bufferWrite[g_bufferWriteSize] = upper;
                        g_bufferWriteSize++;
                        g_bufferWrite[g_bufferWriteSize] = lower;
                        g_bufferWriteSize++;

                        g_voltage_values[0][4] = g_Adc16ConversionValue;
                        g_voltage_values[1][4] = g_range_setting;
```

```
		}

		if(g_active_channels & 0x20) {
			g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL12;
			ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxA);
			Get_Conversion0();

			if(g_range_setting == 0) {
								g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
							}
			else if(g_range_setting == 1) {
								g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
							}
			else if(g_range_setting == 2) {
								g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
							}
			else if(g_range_setting == 3) {
				g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
			}

			uint8_t lower = g_Adc16ConversionValue & 0x00ff;
			uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
			g_bufferWrite[g_bufferWriteSize] = upper;
			g_bufferWriteSize++;
			g_bufferWrite[g_bufferWriteSize] = lower;
			g_bufferWriteSize++;

			g_voltage_values[0][5] = g_Adc16ConversionValue;
			g_voltage_values[1][5] = g_range_setting;
		}

		if(g_active_channels & 0x40) {
			g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL13;
			ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxA);
			Get_Conversion0();

			if(g_range_setting == 0) {
								g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
							}
			else if(g_range_setting == 1) {
								g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
							}
			else if(g_range_setting == 2) {
								g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
							}
			else if(g_range_setting == 3) {
				g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
			}

			uint8_t lower = g_Adc16ConversionValue & 0x00ff;
			uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
			g_bufferWrite[g_bufferWriteSize] = upper;
```

```
                g_bufferWriteSize++;
                g_bufferWrite[g_bufferWriteSize] = lower;
                g_bufferWriteSize++;

                g_voltage_values[0][6] = g_Adc16ConversionValue;
                g_voltage_values[1][6] = g_range_setting;
        }

        if(g_active_channels & 0x80) {
                g_adc16ChannelConfigStruct0.channelNumber = DEMO_ADC16_USER_CHANNEL14;
                ADC16_SetChannelMuxMode(ADC0,kADC16_ChannelMuxA);
                Get_Conversion0();

                if(g_range_setting == 0) {
                                g_Adc16ConversionValue =
lookup_range0[g_Adc16ConversionValue];
                        }
                else if(g_range_setting == 1) {
                                g_Adc16ConversionValue =
lookup_range1[g_Adc16ConversionValue];
                        }
                else if(g_range_setting == 2) {
                                g_Adc16ConversionValue =
lookup_range2[g_Adc16ConversionValue];
                        }
                else if(g_range_setting == 3) {
                        g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
                }

                uint8_t lower = g_Adc16ConversionValue & 0x00ff;
                uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
                g_bufferWrite[g_bufferWriteSize] = upper;
                g_bufferWriteSize++;
                g_bufferWrite[g_bufferWriteSize] = lower;
                g_bufferWriteSize++;

                g_voltage_values[0][7] = g_Adc16ConversionValue;
                g_voltage_values[1][7] = g_range_setting;
        }
}

static void Get_Current_Conversions(void) {

        /* Get conversions from enabled current channels */
        if(g_active_channels & 0x01) {
                g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL1;
                ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxA);
                Get_Conversion1();

                if(g_range_setting == 0) {
                                g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
                        }
                else if(g_range_setting == 1) {
                                g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
                        }
                else if(g_range_setting == 2) {
                                g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
                        }
```

```
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][0] = g_Adc16ConversionValue;
		g_current_values[1][0] = g_range_setting;
}

if(g_active_channels & 0x02) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL4;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxB);
		Get_Conversion1();

		if(g_range_setting == 0) {
					g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 1) {
					g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 2) {
					g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][1] = g_Adc16ConversionValue;
		g_current_values[1][1] = g_range_setting;
}

if(g_active_channels & 0x04) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL5;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxB);
		Get_Conversion1();

		if(g_range_setting == 0) {
					g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 1) {
					g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 2) {
					g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
				}
```

```
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][2] = g_Adc16ConversionValue;
		g_current_values[1][2] = g_range_setting;
}

if(g_active_channels & 0x08) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL14;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxA);
		Get_Conversion1();

		if(g_range_setting == 0) {
					g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 1) {
					g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 2) {
					g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][3] = g_Adc16ConversionValue;
		g_current_values[1][3] = g_range_setting;
}

if(g_active_channels & 0x10) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL15;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxA);
		Get_Conversion1();

		if(g_range_setting == 0) {
					g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 1) {
					g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 2) {
					g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
				}
```

```
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][4] = g_Adc16ConversionValue;
		g_current_values[1][4] = g_range_setting;
}

if(g_active_channels & 0x20) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL18;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxA);
		Get_Conversion1();

		if(g_range_setting == 0) {
					g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 1) {
					g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 2) {
					g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][5] = g_Adc16ConversionValue;
		g_current_values[1][5] = g_range_setting;
}

if(g_active_channels & 0x40) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL19;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxA);
		Get_Conversion1();

		if(g_range_setting == 0) {
					g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 1) {
					g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
				}
		else if(g_range_setting == 2) {
					g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
				}
```

```
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][6] = g_Adc16ConversionValue;
		g_current_values[1][6] = g_range_setting;
	}

	if(g_active_channels & 0x80) {
		g_adc16ChannelConfigStruct1.channelNumber = DEMO_ADC16_USER_CHANNEL20;
		ADC16_SetChannelMuxMode(ADC1,kADC16_ChannelMuxA);
		Get_Conversion1();

		if(g_range_setting == 0) {
			g_Adc16ConversionValue = lookup_range0[g_Adc16ConversionValue];
		}
		else if(g_range_setting == 1) {
			g_Adc16ConversionValue = lookup_range1[g_Adc16ConversionValue];
		}
		else if(g_range_setting == 2) {
			g_Adc16ConversionValue = lookup_range2[g_Adc16ConversionValue];
		}
		else if(g_range_setting == 3) {
			g_Adc16ConversionValue = lookup_range3[g_Adc16ConversionValue];
		}

		uint8_t lower = g_Adc16ConversionValue & 0x00ff;
		uint8_t upper = (g_Adc16ConversionValue & 0xff00) >> 8;
		g_bufferWrite[g_bufferWriteSize] = upper;
		g_bufferWriteSize++;
		g_bufferWrite[g_bufferWriteSize] = lower;
		g_bufferWriteSize++;

		g_current_values[0][7] = g_Adc16ConversionValue;
		g_current_values[1][7] = g_range_setting;
	}
}

//Show last set of conversion values
static void Print_Conversions(void) {

	PRINTF("\r\n\r\nCHANNEL\t\tVOLTAGE\tVALUE\tRANGE\t\tCURRENT\tVALUE\tRANGE\r\n\r\n");
	int i;
	for(i = 0;i < NUM_CHANNELS;i++) {
		PRINTF("%d\t\t%0.3f\t%d\t%d\t\t%0.3f\t%d\t%d\r\n", i+1, (float)(g_voltage_values[0][i] * (VREF_BRD /
SE_10BIT)), g_voltage_values[0][i], g_voltage_values[1][i], (float)(g_current_values[0][i] * (VREF_BRD / SE_10BIT)),
g_current_values[0][i], g_current_values[1][i]);
	}
}

//Get user input for continuous measurement time
```

```
static void Get_Interval(void) {

	static char StrInvalid[] = "Invalid input format\r\n";
	char recvBuf[20];
	uint32_t result;

	uint8_t menu_exit = 0;

	while(!menu_exit) {
		PRINTF("\r\nInput measurement hours, minutes, and seconds like: \"02:05:30\"\r\n");
		ReceiveFromConsole(recvBuf, 8U);
		result = sscanf(recvBuf, "%02hd:%02hd:%02hd", &g_hours, &g_minutes, &g_seconds);
		PRINTF("\r\n");
		/* Error message will appear when user enters the invalid g_date time */
		if (result != 3U)
		{
			PRINTF(StrInvalid);
		}
		else {
			menu_exit = 1;
			g_total_alarm_seconds = (g_hours * 3600) + (g_minutes * 60) + g_seconds;
		}
	}
}

//Set the RTC continuous measurement time
static void Set_Interval(void) {

	RTC_GetDatetime(RTC, &g_alarm_date);
	g_alarm_date.second += g_seconds;
	g_alarm_date.minute += g_minutes;
	g_alarm_date.hour += g_hours;
	if (g_date.second > 59U) {
		g_alarm_date.second -= 60U;
		g_alarm_date.minute += 1U;
	}
	if (g_alarm_date.minute > 59U) {
		g_alarm_date.minute -= 60U;
		g_alarm_date.hour += 1U;
	}
	if (g_alarm_date.hour > 23U) {
		g_alarm_date.hour -= 24U;
		g_alarm_date.day += 1U;
	}
	status_t alarm_flag = RTC_SetAlarm(RTC, &g_alarm_date);
  /* Alarm was successfully set, watch for alarm interrupt */
	if (alarm_flag == kStatus_Success) {
		RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);
	}
	else if (alarm_flag == kStatus_InvalidArgument) {
		PRINTF("Error because the alarm datetime format is incorrect\r\n");
		g_conversion_exit = 1;
	}
	else if (alarm_flag == kStatus_Fail) {
		PRINTF("Error because the alarm time has already passed\r\n");
		g_conversion_exit = 1;
	}
}
```

```
//Print the number of conversions completed
static void Print_Num_Conversions(void) {

	printf("\r\n\r\nTotal number of conversions completed was %d.\r\n", (uint32_t)g_num_conversions);
	uint64_t sps = (g_num_conversions / g_total_alarm_seconds);
	printf("\r\n\r\nConversions were completed at an average rate of %d samples per second.\r\n", (uint32_t)sps);
	uint8_t num_channels = Channel_Count();
	uint64_t csps = (g_num_conversions / g_total_alarm_seconds) / (uint64_t)(num_channels * 2);
	printf("\r\n\r\nThe average sample rate per channel was %d samples per second.\r\n", (uint32_t)csps);
}

//Returns an unsigned byte indicating the total number of channels that are active
uint8_t Channel_Count () {

	uint8_t value = g_active_channels;
  uint8_t count = 0;
  while (value > 0) {          // until all bits are zero
    if ((value & 1) == 1)     // check lower bit
      count++;
    value >>= 1;              // shift bits, removing lower bit
  }
  return count;
}

//CHECK THIS!
/* Delay some time in milliseconds. */
void delay(uint32_t milliseconds)
{
  uint32_t i, j;

  for (i = 0; i < milliseconds; i++)
  {
    for (j = 0; j < 20000U; j++)
    {
      __asm("NOP");
    }
  }
}

//Initialize the SD card reader
static void Init_SD(void) {
	FRESULT error;
	DIR directory; /* Directory object */
	FILINFO fileInformation;

	const TCHAR driverNumberBuffer[3U] = {SDDISK + '0', ':', '/'};

	PRINTF("\r\nPlease insert a card into board.\r\n");

	/* Wait the card to be inserted. */
	int flag = 0;
	#if defined BOARD_SDHC_CD_LOGIC_RISING
	while (!(GPIO_ReadPinInput(BOARD_SDHC_CD_GPIO_BASE, BOARD_SDHC_CD_GPIO_PIN)))
	{
		if(!flag){
			printf("not\n\r");
			flag = 1;
```

```
        }
}

#else
while (GPIO_ReadPinInput(BOARD_SDHC_CD_GPIO_BASE, BOARD_SDHC_CD_GPIO_PIN))
{
        if(!flag){
                printf("so\n\r");
                flag = 1;
        }
}
#endif
PRINTF("Detected SD card inserted.\r\n");

/* Delay some time to make card stable. */
delay(1000U);

if (f_mount(&g_fileSystem, driverNumberBuffer, 0U))
{
        PRINTF("Mount volume failed.\r\n");
}

#if (_FS_RPATH >= 2U)
error = f_chdrive((char const *)&driverNumberBuffer[0U]);
if (error)
{
        PRINTF("Change drive failed.\r\n");
}
#endif

PRINTF("\r\nCreate directory......\r\n");
 error = f_mkdir(_T("/logs"));
 if (error)
 {
         if (error == FR_EXIST)
         {
                 PRINTF("Directory exists.\r\n");
         }
         else
         {
                 PRINTF("Make directory failed.\r\n");
         }
 }

 PRINTF("\r\nCreate a file in that directory......\r\n");
 error = f_open(&g_fileObject, _T("/logs/f_1.dat"), (FA_WRITE | FA_READ | FA_CREATE_ALWAYS));
 if (error)
 {
         if (error == FR_EXIST)
         {
                 PRINTF("File exists.\r\n");
         }
         else
         {
                 PRINTF("Open file failed.\r\n");
         }
 }
```

```
		PRINTF("\r\nList the file in that directory......\r\n");
		if (f_opendir(&directory, "/logs"))
		{
			PRINTF("Open directory failed.\r\n");
		}

		for (;;)
		{
			error = f_readdir(&directory, &fileInformation);

			/* To the end. */
			if ((error != FR_OK) || (fileInformation.fname[0U] == 0U))
			{
				break;
			}
			if (fileInformation.fname[0] == '.')
			{
				continue;
			}
			if (fileInformation.fattrib & AM_DIR)
			{
				PRINTF("Directory file : %s.\r\n", fileInformation.fname);
			}
			else
			{
				PRINTF("General file : %s.\r\n", fileInformation.fname);
			}
		}
}

//Main function
int main(void) {

		FRESULT error;
		UINT bytesWritten;
		g_range_setting = 3;

  BOARD_InitPins();
  BOARD_BootClockRUN();
  BOARD_InitDebugConsole();\

  MPU_Enable(MPU, false);

  Init_Screen();
  RTC_Start();
  Init_DAC();

  EnableIRQ(DEMO_ADC16_IRQn_0);
  EnableIRQ(DEMO_ADC16_IRQn_1);
  Init_ADC();

  int i;

  while(1) {

		//TO DO: set up programmable interupts to hold samples at 10k
//		PIT->MCR = 0;
//		PIT->CHANNEL[0].LDVAL = 0x0003E7FF; // setup timer 1 for 256000 cycles
```

```
//		PIT->CHANNEL[0].TCTRL|= PIT_TCTRL_TIE_MASK | PIT_TCTRL_TEN_MASK; // Enable interrupt and enable timer
//

		g_conversion_exit = 0;

		//Display user menu
		Init_Menu();

		//Prepare SD Card for writing
	Init_SD();

	//init mock lookup tables
	for(i=0;i<1024;i++) {
		lookup_range0[i] = i;
	}
	for(i=0;i<1024;i++) {
		lookup_range1[i] = i;
	}
	for(i=0;i<1024;i++) {
		lookup_range2[i] = i;
	}
	for(i=0;i<1024;i++) {
		lookup_range3[i] = 1023 -i;
	}

		//Write current active channels to the buffer
		g_bufferWrite[0] = g_active_channels;
		g_bufferWriteSize++;

		//Write current range setting to the buffer
		g_bufferWrite[g_bufferWriteSize] = g_range_setting;
		g_bufferWriteSize++;

		//If continuous measurement mode set measurement time
		if(g_ADC_Mode == 2) {
			Set_Interval();
		}

		int loop_num = 0;

			while (!g_conversion_exit) {

				//Read current each loop
				Get_Current_Conversions();

				//Read voltage once per second at 10k samples
				if(loop_num == 10000) {
					Get_Voltage_Conversions();
					loop_num = 0;
				}

				//If measurement mode 1 take 1 measurement and print results
				if(g_ADC_Mode == 1) {
					Print_Conversions();
					g_conversion_exit = 1;
				}

				//If measurement mode 2 write buffer to SD when full
```

```
                        if(g_ADC_Mode == 2) {
                                if(g_bufferWriteSize >= 4096) {
                                        error = f_write(&g_fileObject, g_bufferWrite, g_bufferWriteSize,
&bytesWritten);
                                        if ((error) || (bytesWritten != g_bufferWriteSize))
                                                {
                                            PRINTF("%s Write file failed. \r\n", (char*)error);
                                            PRINTF("%d byteswritten \r\n", bytesWritten);
                                        }
                                        g_bufferWriteSize = 0;
                                        memset(g_bufferWrite, '0', sizeof(g_bufferWrite));
                                }

                                //If measurement interval is complete write remaining buffer contents,
                                //sample information, and most recent results to screen
                                if(g_AlarmPending == 1U) {
                                        Get_Voltage_Conversions();
                                        RTC_DisableInterrupts(RTC, kRTC_AlarmInterruptEnable);
                                        error = f_write(&g_fileObject, g_bufferWrite, g_bufferWriteSize,
&bytesWritten);
                                        if ((error) || (bytesWritten != g_bufferWriteSize))
                                                {
                                                        PRINTF("%s Write file failed. \r\n", (char*)error);
                                                        PRINTF("%s byteswritten \r\n", bytesWritten);
                                        }

                                        PRINTF("Buffersize: %d\n",g_bufferWriteSize);
                                        g_bufferWriteSize = 0;
                                        memset(g_bufferWrite, '0', sizeof(g_bufferWrite));

                                        Print_Conversions();
                                        Print_Num_Conversions();
                                        g_num_conversions = 0;
                                        g_AlarmPending = 0U;
                                        g_conversion_exit = 1;
                                }
                        }
                        loop_num++;
                }
                //Close SD file
        if (f_close(&g_fileObject))
        {
           PRINTF("\r\nClose file failed.\r\n");
           return -1;
        }
        else {
                PRINTF("\r\nClose file successful.\r\n");
        }
  }
}
```