AN14103

NAFE applications with MCUXpresso

Rev. 1 — 6 May 2024 Application note

Document information

Information	Content
Keywords	NAFE, analog front-end, software, MCUXpresso, platform, measurement, industrial, voltage, current, temperature, weight
Abstract	This article introduces the precompiled NAFE software library for NXP proprietary MCUXpresso platform. The library consists of ready-to-use measurement script for the industrial applications (voltage, current, temperature and weight) with NAFE evaluation KIT as discussed in AN14102.

1 Introduction

This application note introduces the precompiled NAFE software library for the NXP proprietary MCUXpresso platform. The library consists of ready-to-use measurement scripts for industrial applications (voltage, current, temperature, and weight) with the NAFE evaluation KIT as discussed in AN14102 "Industrial application measurements using NXP AFE".

The application note contains the code snippets for all four applications and walks the reader through different steps of the scripts to achieve these measurements.

2 MCUXpresso IDE

The MCUXpresso Integrated Development Environment (IDE) is a fully featured, free software development environment for NXP's ARM-based microcontroller units (MCU) and includes all the tools necessary to develop high-quality embedded software applications. The MCUXpresso IDE is based on the Eclipse IDE and includes the industry-standard ARM GNU toolchain. The MCUXpresso brings developers an easy-to-use and unlimited code-size development environment for NXP MCUs based on Cortex-M cores (LPC, Kinetis, and iMX RT). The IDE combines the best of the widely popular LPCXpresso and Kinetis Design Studio IDEs, providing a common platform for all NXP Cortex-M microcontrollers. It provides an intuitive and powerful interface with profiling, power measurement on supported boards, GNU tool integration and library, multicore-capable debugger, trace functionality, and more. MCUXpresso IDE debug connections support Freedom, Tower, EVK, LPCXpresso, and custom development boards with industry-leading open-source and commercial debug probes, including MCU-Link, MCU-Link Pro, LPC-Link2, PEmicro, and SEGGER.

2.1 MCUXpresso installation

Download and install the IDE from the NXP website and follow the instructions in the installation window.

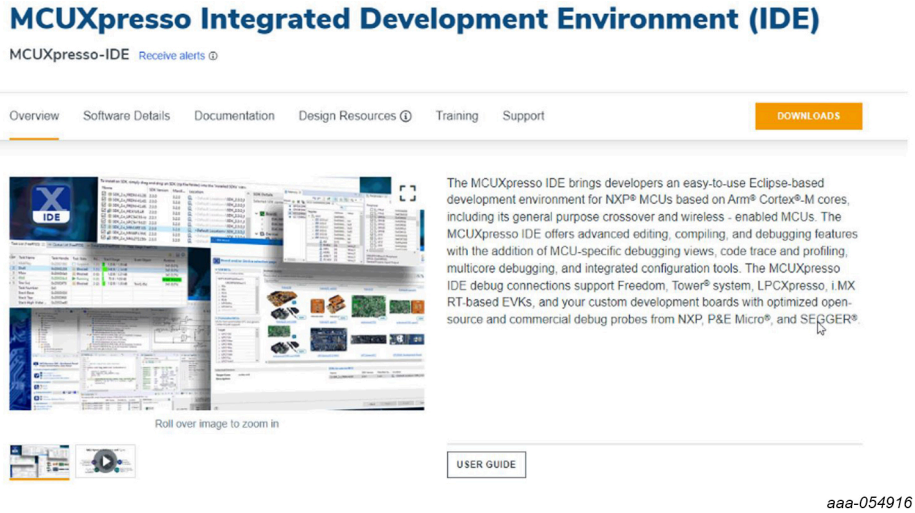
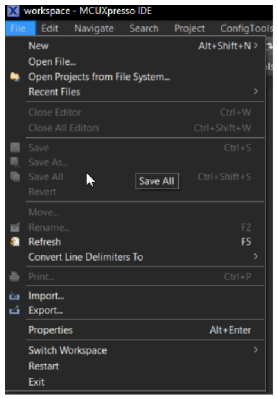


Figure 1. IDE installation

2.2 MCUXpresso setup

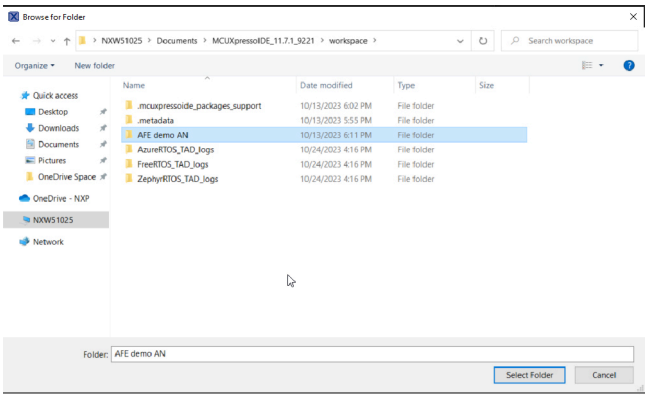
Download the software development kit (SDK) pack “NAFE MCUXpresso Demo Project for LPC54S018” from Highly Configurable 8 Channel ±25 V Universal Input Analog Front-End | NXP Semiconductors

Click **File** ⇒ **Open Projects from File System** and select the directory where the SDK pack is downloaded.



aaa-054899

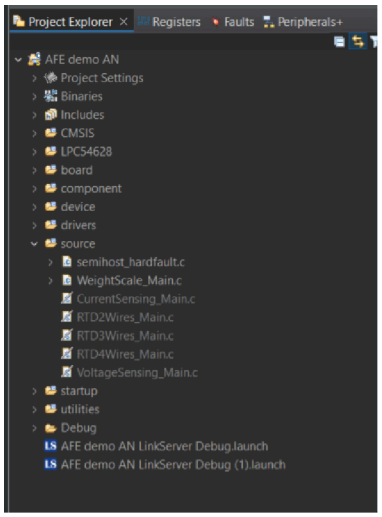
Figure 2. File selection



aaa-054898

Figure 3. File location

Once the directory is selected, Eclipse IDE will automatically import the entire project into a workspace as shown in Figure 4.

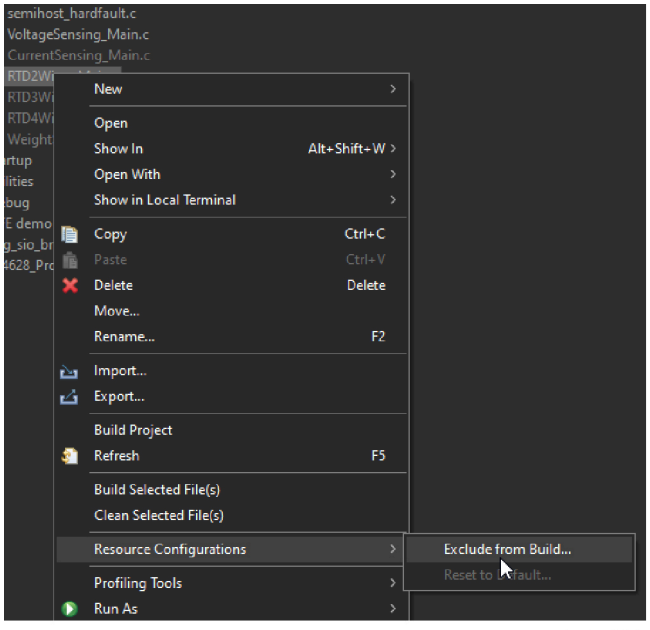


aaa-054900

Figure 4. Import into workspace

The source folder has main files corresponding to applications – voltage sensing, current sensing, RTD, and weigh scale. Follow the steps to run the selected application:

1. Right click the desired xx_main.c file (inside source folder) to compile it and go to **Resource Configurations** ⇒ **Exclude from Build**.



aaa-054909

Figure 5. Resource configuration

2. Select **Debug** and **Release** to exclude from the compilation as shown in Figure 6.

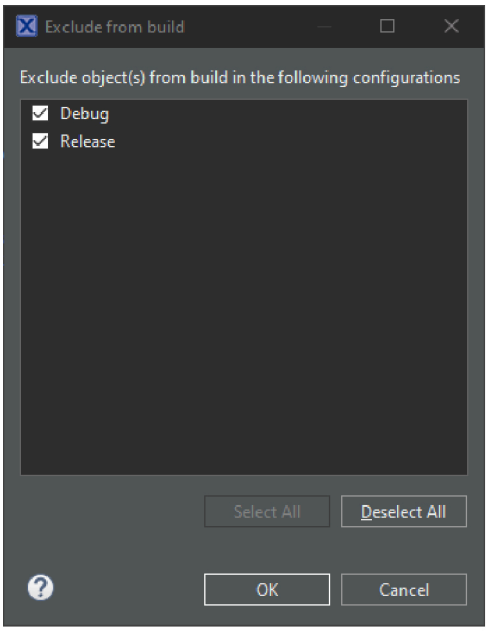


Figure 6. Select Debug and Release

3. Repeat steps 1 and 2 to exclude the old, active main file from the compilation.
4. Configure the serial terminal and add a COMx port as shown in Figure 7 to run the project. Click the blue button to open the terminal sheet, choose **Serial Terminal**, and **Serial port** as applicable.

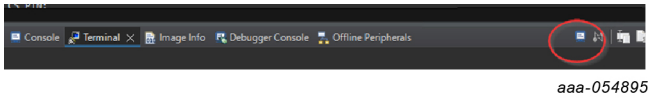


Figure 7. Configure terminal and port

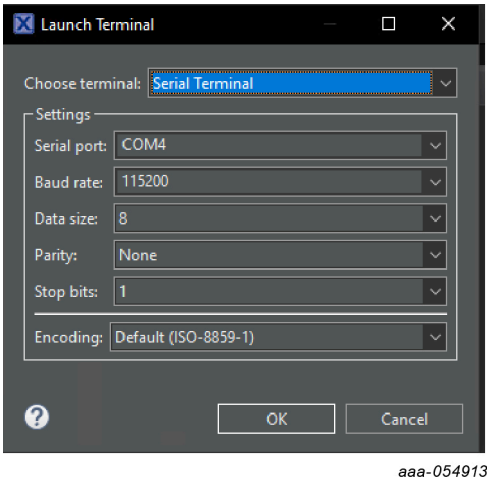


Figure 8. Select terminal and port

3 NAFE drivers

Figure 9 shows the project structure:

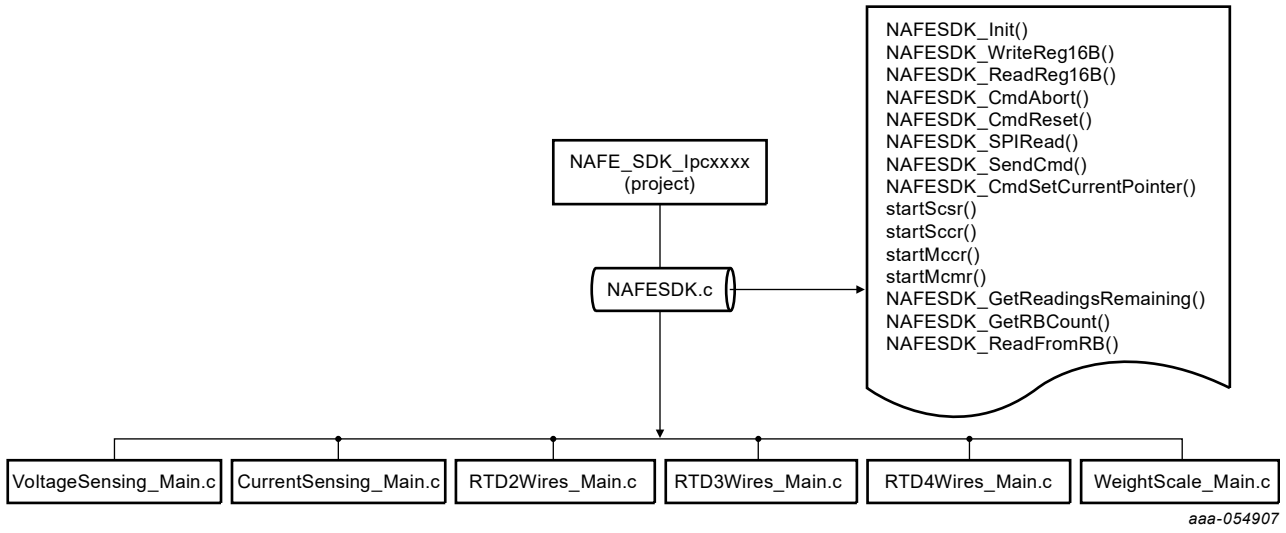


Figure 9. Project structure

The NAFE driver uses the following software components to perform various functions:

- *NAFESDK_Init*(**nafe_t*** instance, **uint32_t** spiClkFreq): Initialize the LPC peripheral.
- *NAFESDK_WriteReg16B*(**nafe_t** *instance, **uint8_t** reg_addr, **uint16_t** val): Write 16-bit register, refer to the NAFE13388 data sheet.
- *NAFESDK_ReadReg16B*(**nafe_t** *instance, **uint8_t** reg_addr): Read 16-bit register, refer to the NAFE13388 data sheet..
- *NAFESDK_CmdAbort*(**nafe_t** *instance): Send abort command to stop continuous conversion.
- *NAFESDK_CmdReset*(**nafe_t** *instance): Send reset command to reset the device.
- *NAFESDK_SPIRead*(**nafe_t** *instance, **uint8_t** *buf, **uint32_t** len): Read NAFE data via SPI bus.
- *NAFESDK_SendCmd*(**nafe_t** *instance, **uint16_t** cmd, **uint8_t** dir, **uint8_t** *buf, **uint32_t** len): Send command.
- *NAFESDK_CmdSetCurrentPointer*(**nafe_t** *instance, **uint8_t** adc_chl): Set channel pointer.
- *startScsr*(**nafe_t** *instance): Start single channel single reading.
- *startSccr*(**nafe_t** *instance, **int** numReads): Start and handle (via DMA) single channel continuous reading.
- *startMccr*(**nafe_t** *instance, int numReads, **uint16_t** channelMask, **bool** drdybPinSeq): Start and handle (via DMA) multichannel. continuous reading, channel mask is a mask for enabled channels, drdybPinSeq is to choose data ready at each conversion or at the end of sequence.
- *startMcmr*(**nafe_t** *instance, **uint16_t** channelMask, **bool** drdybPinSeq): Start and handle (via DMA) multichannel multireading, channel mask is a mask for enabled channels, drdybPinSeq is to choose data ready at each conversion or at the end of sequence.
- *NAFESDK_GetReadingsRemaining*(**nafe_t** *instance): Get the remaining samples of the current DMA reading.
- *NAFESDK_GetRBCount*(**nafe_t** *instance): Get the empty positions into ring buffer.
- *NAFESDK_ReadFromRB*(**nafe_t** *instance, **uint8_t** *buf, **uint32_t** len): Read data inside ring buffer.

4 NAFE application specific code

This section describes various application cases implemented on the MCUXpresso. Refer to AN14102 “Industrial application measurements using NXP AFE“ for application theory and device configuration. Every application has the following common high-level steps:

- Reset the NAFE to avoid any carryover configuration.
- Assignment of channel pointer and register configuration as per the application requirements discussed in AN14102 "Industrial application measurements using NXP AFE".
- NAFE conversion and data capture.
- Code conversion to meaningful data: voltage, current, temperature, or weight.

In the following subsections, the full code for the application is presented first, followed by highlighted blocks to explain the step-by-step sequence executed by the code.

4.1 Voltage sensing

The following code snippet shows the steps to configure the NAFE for voltage sensing according to the example presented in AN1402 "Industrial application measurements using NXP AFE":

1. Perform a device reset before a new configuration of the NAFE registers with the NAFESDK_CmdReset() function, followed by a 10 ms delay.
2. Set Current Register Pointer with the NAFESDK_CmdSetCurrentPointer() function. With this action, the subsequent actions will be linked to the chosen channel.
3. Configure CH_CONFIG0, CH_CONFIG1, and CH_CONFIG2 through the NAFESDK_WriteReg16B() function.
4. To use the value of the PGA gain setup, save it inside an array after the conversion to appropriately scale the voltage.
5. Repeat steps 3, 4, and 5 for every channel. In Figure 10, six channels are used.

```
NAFESDK_CmdReset(&instance);
    delay(10);

    // Voltage Reference A
    NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x1711); // AI1P to AICOM - GAIN 0.2x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x28); // SINC4 - 12000Sps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C00); // Delay 16.493us
    gains[0]=0.2;

    // Voltage Reference B
    NAFESDK_CmdSetCurrentPointer(&instance,1); // Set Pointer to ch1
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x7131); // AI1N to AICOM - GAIN 0.4x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x1028); // SINC4 - 12000Sps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C00); // Delay 16.493us
    gains[1]=0.4;

    // Power Supply rail C
    NAFESDK_CmdSetCurrentPointer(&instance,2); // Set Pointer to ch2
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x2711); // AI2P to AICOM - GAIN 0.2x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x18);  // SINC4 - 24000Sps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C00); // Delay 16.493us
    gains[2]=0.2;

    // Power Supply rail D
    NAFESDK_CmdSetCurrentPointer(&instance,3); // Set Pointer to ch3
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x2731); // AI1N to AICOM - GAIN 0.4x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x1018); // SINC4 - 24000Sps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C00); // Delay 16.493us
    gains[3]=0.4;

    // Sensor E
    NAFESDK_CmdSetCurrentPointer(&instance,4); // Set Pointer to ch4
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x3731); // AI3P to AICOM - GAIN 0.4x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x1038); // SINC4 - 6000Sps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x3800); // Delay 33.4us
    gains[4]=0.4;

    // Sensor F
    NAFESDK_CmdSetCurrentPointer(&instance,5);
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x7391); // AI3N to AICOM - GAIN 2x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x4038); // SINC4 - 6000ps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x3800); // Delay 33.4us
    gains[5]=2;

    // Sensor F
    NAFESDK_CmdSetCurrentPointer(&instance,6);
    NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x44F1); // AI4P to AI4N - GAIN 16x
    NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x7071); // SINC4+1 - 50ps
    NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C00); // Delay 16.493us
    gains[6]=16;
```

aaa-055694

Figure 10. NAFE channels configuration for voltage sensing

Once the NAFE is configured, the conversion command is issued, digital code is captured, the code is translated to voltage (marked up by the red box) and the results are printed on the console using the script shown in Figure 11.

```
while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        for (int i=0;i<6;i++){
                NAFESDK_CmdSetCurrentPointer(&instance,i); // Set Pointer to ch i (0 -> 5)
                ticks[i] = tic(); // Start counter to monitor the duration for each conversion
                startScsr(&instance); // Start a single channel conversion (cmd ID 0x2000)
                while(GPIO_PinRead(GPIO, instance.gpio_rdy_port, instance.gpio_rdy_pin)==0); /* wait for RDY pin to high */
                ticks[i] = toc(ticks[i]); // Stop counter to monitor the duration for each conversion
                NAFESDK_SPIRead(&instance, buf, 3); // Read a 3 Byte (24 bit) ADC conversion
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltages[i] = sINT24_TO_VG(sINT24, gains[i]); //Apply Conversion from 24 bit to voltage value
                }
        PRINTF("------------Conversion------------\r\n");
        PRINTF("\r\nVRefA:%6fV, VRefB:%6fV, RailC:%6fV, RailD:%6fV, SensorE:%6fV, SensorF:%6fV\r\n",
        voltages[0],voltages[1],voltages[2],voltages[3],voltages[4],voltages[5]);
        time_tot = 0;
        for (int i=0;i<6;i++){
                time = (ticks[i]) / (SystemCoreClock / 1000.F);
                PRINTF("Elapsed Time for ch%d = %3fms\r\n", i ,time);
                time_tot += time;
                }
        PRINTF("Total Elapsed Time = %3fms\r\n", time_tot);
        PRINTF("\r\n-----------------------------------\r\n");
        }
```

aaa-054905

Figure 11. ADC conversion and results print out

```
while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        for (int i=0;i<6;i++){
                NAFESDK_CmdSetCurrentPointer(&instance, i); // Set Pointer to ch i (0 -> 5)
                ticks[i] = tic(); // Start counter to monitor the duration for each conversion
                startScsr(&instance); // Start a single channel conversion (cmd ID 0x2000)
                while(GPIO_PinRead(GPIO, instance.gpio_rdy_port, instance.gpio_rdy_pin)==0); /* wait for RDY pin to high */
                ticks[i] = toc(ticks[i]); // Stop counter to monitor the duration for each conversion
                NAFESDK_SPIRead(&instance, buf, 3); // Read a 3 Byte (24 bit) ADC conversion
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltages[i] = sINT24_TO_VG(sINT24, gains[i]); //Apply Conversion from 24 bit to voltage value
                }
        PRINTF("------------Conversion------------\r\n");
        PRINTF("\r\nVRefA:%6fV, VRefB:%6fV, RailC:%6fV, RailD:%6fV, SensorE:%6fV, SensorF:%6fV\r\n",
        voltages[0],voltages[1],voltages[2],voltages[3],voltages[4],voltages[5]);
        time_tot = 0;
        for (int i=0;i<6;i++){
                time = (ticks[i]) / (SystemCoreClock / 1000.F);
                PRINTF("Elapsed Time for ch%d = %3fms\r\n", i ,time);
                time_tot += time;
                }
        PRINTF("Total Elapsed Time = %3fms\r\n", time_tot);
        PRINTF("\r\n-----------------------------------\r\n");
        }
```

aaa-054906

Figure 12. ADC conversion

The following is a step-by-step explanation of the highlighted block in Figure 12:

1. Set Current Register Pointer with the NAFESDK_CmdSetCurrentPointer() function. With this action, the subsequent single conversion will be linked to the chosen channel.
2. Start a counter in order to measure the conversion's elapsed time.
3. Start a single-channel single-reading (SCSR) conversion with the startScsr() function. This function sends the command ID 0x2000, as explained in the *"CMD reading modes"* section of the NAFE13388 data sheet in Secure Files.
4. Wait for the data-ready pin to go high and capture the valid data.
5. Start a counter in order to measure the conversion's elapsed time.

The highlighted block in Figure 13 reads the ADC buffer and translates the ADC output code to voltage value.

```
while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        for (int i=0;i<6;i++){
                NAFESDK_CmdSetCurrentPointer(&instance,i); // Set Pointer to ch i (0 -> 5)
                ticks[i] = tic(); // Start counter to monitor the duration for each conversion
                startScsr(&instance); // Start a single channel conversion (cmd ID 0x2000)
                while(GPIO_PinRead(GPIO, instance.gpio_rdy_port, instance.gpio_rdy_pin)==0); /* wait for RDY pin to high */
                ticks[i] = toc(ticks[i]); // Stop counter to monitor the duration for each conversion
                NAFESDK_SPIRead(&instance, buf, 3); // Read a 3 Byte (24 bit) ADC conversion
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltages[i] = sINT24_TO_VG(sINT24, gains[i]); //Apply Conversion from 24 bit to voltage value
                }
        PRINTF("------------Conversion------------\r\n");
        PRINTF("\r\nVRefA:%6fV, VRefB:%6fV, RailC:%6fV, RailD:%6fV, SensorE:%6fV, SensorF:%6fV\r\n",
        voltages[0],voltages[1],voltages[2],voltages[3],voltages[4],voltages[5]);
        time_tot = 0;
        for (int i=0;i<6;i++){
                time = (ticks[i]) / (SystemCoreClock / 1000.F);
                PRINTF("Elapsed Time for ch%d = %3fms\r\n", i ,time);
                time_tot += time;
                }
        PRINTF("Total Elapsed Time = %3fms\r\n", time_tot);
        PRINTF("\r\n----------------------------------\r\n");
        }
```

aaa-054893

Figure 13. ADC code read and voltage translation

1. To read an ADC conversion, send a SPI request to read 3 bytes (24 bits). To do this, use the NAFESDK_SPIRead() function, specifying the number of bytes (3) and the buffer (array of uint8_t).
2. With some bitwise operations, arrange the read 3 bytes to build a 24-bit word that will be saved into a int32_t type.
3. Do a conversion from 24-bit to a decimal signed number (see ADC code to voltage translation in the NAFE13388 data sheet), sINT24 = [(adc_res + 2^{23}) % 2^{24}] - 2^{23}.
4. Do a conversion to scaled voltage (see ADC code to voltage translation in the NAFE13388 data sheet), VG = 10 / 2^{24} * (sINT24 / GAIN). Save it on an array where each result will be stored.

The result is displayed on the console using the highlighted code shown in Figure 14.

```
while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        for (int i=0;i<6;i++){
                NAFESDK_CmdSetCurrentPointer(&instance,i); // Set Pointer to ch i (0 -> 5)
                ticks[i] = tic(); // Start counter to monitor the duration for each conversion
                startScsr(&instance); // Start a single channel conversion (cmd ID 0x2000)
                while(GPIO_PinRead(GPIO, instance.gpio_rdy_port, instance.gpio_rdy_pin)==0); /* wait for RDY pin
                to high */
                ticks[i] = toc(ticks[i]); // Stop counter to monitor the duration for each conversion
                NAFESDK_SPIRead(&instance, buf, 3); // Read a 3 Byte (24 bit) ADC conversion
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltages[i] = sINT24_TO_VG(sINT24, gains[i]); //Apply Conversion from 24 bit to voltage value
                }
        PRINTF("------------Conversion------------\r\n");
        PRINTF("\r\nVRefA:%6fV, VRefB:%6fV, RailC:%6fV, RailD:%6fV, SensorE:%6fV, SensorF:%6fV\r\n",
        voltages[0],voltages[1],voltages[2],voltages[3],voltages[4],voltages[5]);
        time_tot = 0;
        for (int i=0;i<6;i++){
                time = (ticks[i]) / (SystemCoreClock / 1000.F);
                PRINTF("Elapsed Time for ch%d = %3fms\r\n", i ,time);
                time_tot += time;
                }
```

aaa054897

Figure 14. Code for result display

The console prints the voltage measured on each channel and the actual readout time (including the console's print time), as calculated in AN14102 "Industrial application measurements using NXP AFE".

```
->Click any button to start one Conversion sequences<-
------------Conversion------------

Ch0:2.496153V, Ch1:0.252321V, Ch2:0.588778V, Ch3:0.594468V, Ch4:0.139143V, Ch5:0.135468V, Ch6:0.000776V
Elapsed Time for ch0 = 0.107373ms
Elapsed Time for ch1 = 0.107105ms
Elapsed Time for ch2 = 0.065268ms
Elapsed Time for ch3 = 0.065268ms
Elapsed Time for ch4 = 0.207673ms
Elapsed Time for ch5 = 0.207405ms
Elapsed Time for ch6 = 19.997614ms
Total Elapsed Time = 20.757706ms
----------------------------------
```

aaa-054914

Figure 15. Voltage sensing console printout

4.2 Current sensing

```
NAFESDK_CmdReset(&instance);
delay(10);

// Pressure transducer
NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x4430); // AI4P to AI4N - GAIN 0.4x
NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x1028); // SINC4 - 12000Sps
NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x4400); // Delay 16.5us
gains[0]=1;


while(1)
{
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.

        // For Cycle for each channel
        for (int i=0;i<1;i++){
                NAFESDK_CmdSetCurrentPointer(&instance,i); // Set Pointer to ch i (0 -> 0)
                ticks[i] = tic(); // Start counter to monitor the duration for each conversion
                startScsr(&instance); // Start a single channel conversion (cmd ID 0x2000)

                while(GPIO_PinRead(GPIO, instance.gpio_rdy_port, instance.gpio_rdy_pin)==0); /* wait for RDY pin high */
                ticks[i] = toc(ticks[i]); // Stop counter to monitor the duration for each conversion
                NAFESDK_SPIRead(&instance, buf, 3); // Read a 3 Byte (24 bit) ADC conversion


                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltages[i] = sINT24_TO_VG(sINT24, gains[i]); //Apply Conversion from 24 bit to voltage value
        }

        PRINTF("------------Conversion-------------\r\n");
        PRINTF("\r\nVoltage Shunt=%6fV, Current Shunt(0.25ohm)=%6fmA\r\n", voltages[0],(voltages[0]/0.25)*1000);
        time_tot = 0;
        for (int i=0;i<1;i++){
                time = (ticks[i]) / (SystemCoreClock / 1000.F);
                PRINTF("Elapsed Time for ch%d = %3fms\r\n", i ,time);
                time_tot += time;
        }
        PRINTF("Total Elapsed Time = %3fms\r\n", time_tot);
        PRINTF("\r\n-----------------------------------\r\n");
    }

}
```

aaa-054896

Figure 16. Current sensing

The current sensing application uses a code sequence similar to voltage sensing to reset, configure, issue a conversion command, and capture digital code from the NAFE. The current is calculated by dividing the measured voltage by the known resistor value of 250 ohms, as in the example discussed in AN1402 “Industrial application measurements using NXP AFE". The code prints the below results on the console.

```
->Click any button to start one Conversion sequences<-
-----------Conversion------------
Voltage Shunt=0.000401V, Current Shunt(0.25ohm)=1.602173mA
Elapsed Time for ch0 = 0.107164ms
Total Elapsed Time = 0.107164ms
-----------------------------------
```

aaa-054904

Figure 17. Current sensing console print out

4.3 4-wire RTD sensing

AN14102 "Industrial application measurements using NXP AFE" discusses the theory and implementation of 2-wire, 3-wire, and 4-wire applications using the NAFE. As the code sequence is similar, this document will focus on the 4-wire RTD application using MCUXpresso.

```
// RTD Sensing and Forcing
NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x11B1); // AI1P to AI1N - GAIN 4x
NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x5079); // SINC4+1 - 1125Sps
NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C80); // Delay 16.5us
NAFESDK_WriteReg16B(&instance, CH_CONFIG3, 0xB010); // Current - Positive Pol - 750uA - AI2P
gain=4;

while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
        startSccr(&instance,conversionNumber); // Start a Single Channel Continuous Conversion
        while(NAFESDK_GetReadingsRemaining(&instance) != 0) {}; // Wait for conversion end.
        NAFESDK_CmdAbort(&instance);
        voltageSum = 0; // Initialize result variable
        len = NAFESDK_GetRBCount(&instance); // get length of buffer
        // convert sample to voltage and sum it in order to perform an average
        PRINTF("\r\n-----------Raw Conversion------------\r\n");
        for(i=0; i<len / ADC_READ_SIZE; i++){
                NAFESDK_ReadFromRB(&instance, buf, 3);
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltage = sINT24_TO_VG(sINT24, gain); //Apply Conversion from 24 bit to voltage value
                voltageSum += voltage;
                PRINTF("Sample %d: Voltage RTD=%6fV, Temperature RTD=%3fC\r\n", i,voltage,((voltage/ISource)-R0)/(A*R0));
                }
        //Calculation of Temperature.
        PRINTF("-------------------------------------");
        voltage = voltageSum/conversionNumber; //Average Calculation
        PRINTF("\r\n-----------Average Conversion------------");
        PRINTF("\r\nVoltage RTD=%6fV, Temperature RTD=%3fC", voltage,((voltage/ISource)-R0)/(A*R0)); //Calculation of
        Temperature.
        PRINTF("\r\n-----------------------------------\r\n");
    }
```

aaa-054892

Figure 18. 4-wire RTD sensing

In the highlighted snippet shown in Figure 19, startSccr() initiates a continuous conversion command and the direct memory access (DMA) handler is configured to sync with the NAFE signal (data ready or SYNC) in order to catch and save every valid conversion inside a buffer.

```
// RTD Sensing and Forcing
NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x11B1); // AI1P to AI1N - GAIN 4x
NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x5079); // SINC4+1 - 1125Sps
NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C80); // Delay 16.5us
NAFESDK_WriteReg16B(&instance, CH_CONFIG3, 0xB010); // Current - Positive Pol - 750uA - AI2P
gain=4;

while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
        startSccr(&instance,conversionNumber); // Start a Single Channel Continuos Conversion
        while(NAFESDK_GetReadingsRemaining(&instance) != 0) {}; // Wait for conversion end.
        NAFESDK_CmdAbort(&instance);
        voltageSum = 0; // Initialize result variable
        len = NAFESDK_GetRBCount(&instance); // get length of buffer
        // convert sample to voltage and sum it in order to perform an average
        PRINTF("\r\n-----------Raw Conversion------------\r\n");
        for(i=0; i<len / ADC_READ_SIZE; i++){
                NAFESDK_ReadFromRB(&instance, buf, 3);
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltage = sINT24_TO_VG(sINT24, gain); //Apply Conversion from 24 bit to voltage value
                voltageSum += voltage;
                PRINTF("Sample %d: Voltage RTD=%6fV, Temperature RTD=%3fC\r\n", i,voltage,((voltage/ISource)-R0)/(A*R0));
                }
        //Calculation of Temperature.
        PRINTF("-----------------------------------");
        voltage = voltageSum/conversionNumber; //Average Calculation
        PRINTF("\r\n-----------Average Conversion------------");
        PRINTF("\r\nVoltage RTD=%6fV, Temperature RTD=%3fC", voltage,((voltage/ISource)-R0)/(A*R0)); //Calculation of
        Temperature.
        PRINTF("\r\n-----------------------------------\r\n");
```

aaa-054910

Figure 19. RTD code conversion and capture

The NAFESDK_GetReadingsRemaining() function returns the free positions of the buffer. The conversion is complete when the position becomes 0.

```
// RTD Sensing and Forcing
NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x11B1); // AI1P to AI1N - GAIN 4x
NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x5079); // SINC4+1 - 1125Sps
NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C80); // Delay 16.5us
NAFESDK_WriteReg16B(&instance, CH_CONFIG3, 0xB010); // Current - Positive Pol - 750uA - AI2P
gain=4;

while(1){
        PRINTF("\r\n->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
        startSccr(&instance,conversionNumber); // Start a Single Channel Continuos Conversion
        while(NAFESDK_GetReadingsRemaining(&instance) != 0) {}; // Wait for conversion end.
        NAFESDK_CmdAbort(&instance);
        voltageSum = 0; // Initialize result variable
        len = NAFESDK_GetRBCount(&instance); // get length of buffer
        // convert sample to voltage and sum it in order to perform an average
        PRINTF("\r\n-----------Raw Conversion------------\r\n");
        for(i=0; i<len / ADC_READ_SIZE; i++){
                NAFESDK_ReadFromRB(&instance, buf, 3);
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltage = sINT24_TO_VG(sINT24, gain); //Apply Conversion from 24 bit to voltage value
                voltageSum += voltage;
                PRINTF("Sample %d: Voltage RTD=%6fV, Temperature RTD=%3fC\r\n", i,voltage,((voltage/ISource)-R0)/(A*R0));
                }
        //Calculation of Temperature.
        PRINTF("-----------------------------------");
        voltage = voltageSum/conversionNumber; //Average Calculation
        PRINTF("\r\n-----------Average Conversion------------");
        PRINTF("\r\nVoltage RTD=%6fV, Temperature RTD=%3fC", voltage,((voltage/ISource)-R0)/(A*R0)); //Calculation of
        Temperature.
        PRINTF("\r\n-----------------------------------\r\n");
    }
```

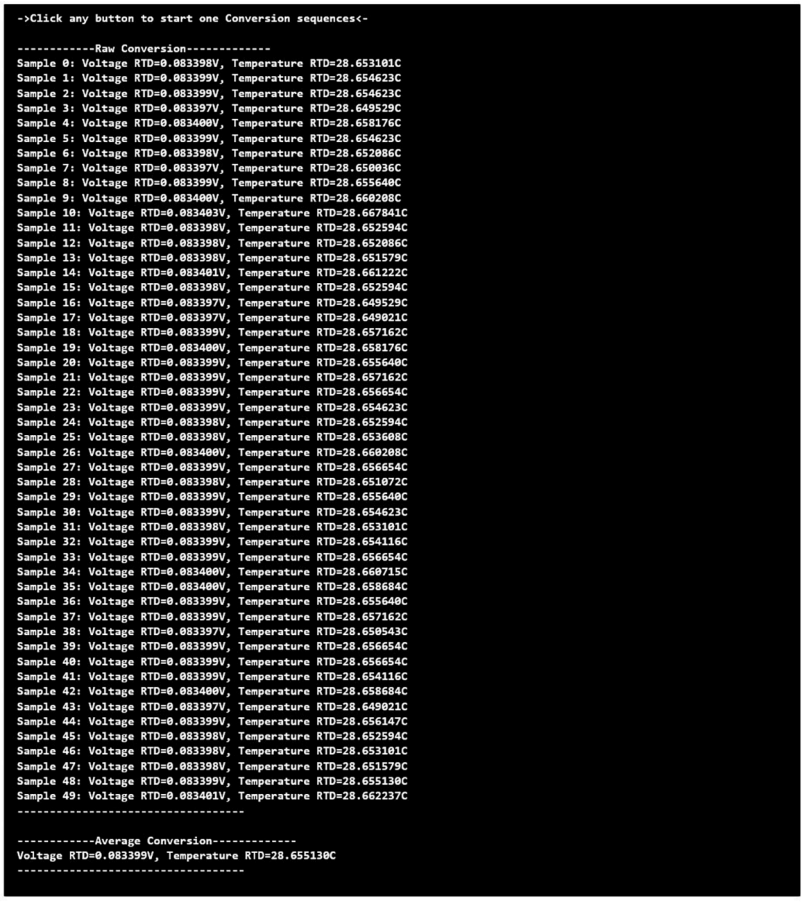
aaa-054911

Figure 20. RTD voltage to temperature translation

Once the 50 samples are complete, 3 bytes of output code are extracted from the DMA buffer using the NAFESDK_ReadFromRB() inside the for loop as highlighted in Figure 20.

The remaining lines of code handle the serial console and convert the voltage to a temperature value using the transfer function discussed in AN14102 "Industrial application measurements using NXP AFE".

The console prints the raw voltage measurement across the RTD and the converted temperature value for all 50 samples. Any temperature change the RTD probe is exposed to during this time, for example, if it is dipped in a cup of ice, will be captured in the console readout.



aaa-054908

Figure 21. Raw voltage and temperature console print out

4.4 Weight scale (load cell)

Refer to AN14102 "Industrial application measurements using NXP AFE" for the theory of weight scale application and implementation using the NAFE.

The NAFE offset needs to be compensated and can be calibrated with a known weight for an accurate measurement. The below code in Figure 22 asks the user to remove the weight from the scale, if any, to measure the scale offset using the NAFE startSccr() and NAFESDK_GetReadingsRemaining() functions. Three conversions are performed and an average value is stored in a variable called offset.

```
NAFESDK_CmdReset(&instance);
delay(10);

// Ch0 Force/Sense Configuration
NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
NAFESDK_WriteReg16B(&instance, CH_CONFIG0, 0x11F1); // AI1P to AI1N - GAIN 16x
NAFESDK_WriteReg16B(&instance, CH_CONFIG1, 0x70A4); // SINC4+1 - 100Sps
NAFESDK_WriteReg16B(&instance, CH_CONFIG2, 0x2C80); // Delay 16.5us
NAFESDK_WriteReg16B(&instance, CH_CONFIG3, 0x3410); // Voltage - Positive Pol - 6V - AI2P
gain=16;

PRINTF("-> Calculation of Coefficient <-\r\n");
PRINTF("\r\n-> 1 Step :Offset Calculation <-\r\n");
PRINTF("Leave the weight scale without weight and click any buttons. \r\n");
GETCHAR(); // Wait a keyboard import to start a conversion sequence.

NAFESDK_CmdSetCurrentPointer(&instance,0); // Set Pointer to ch0
startSccr(&instance,conversionNumber); // Start a Single Channel Conversion, in order to get "conversionNumber" results
while(NAFESDK_GetReadingsRemaining(&instance) != 0) {}; // Wait for conversion end.
NAFESDK_CmdAbort(&instance);

voltage = 0; // Initialize result variable
len = NAFESDK_GetRBCount(&instance); // get length of buffer
// convert sample to voltage and sum it in order to perform an average
for(i=0; i<len / ADC_READ_SIZE; i++)
{
        NAFESDK_ReadFromRB(&instance, buf, 3);
        adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
        sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
        voltage += sINT24_TO_VG(sINT24, gain); //Apply Conversion from 24 bit to voltage value
}
offset = voltage/conversionNumber; //Average Calculation
PRINTF("-> 1 Step : Done <-\r\n");
```

aaa-054915

Figure 22. Weight scale offset measurement

The below highlighted part of the code in Figure 23 asks the user to put the known weight on the scale and pass the value of it (in grams) to the program. This known value is used to calibrate the NAFE for accurate measurements. The first loop block is used to get user input and confirmation of the value (in kilograms) of the weight put on the weigh scale for calibration. The serial console prints the value provided by user. The console then asks the user if the saved known value is correct. Otherwise, the console asks the user to again input the weight value. Once the user input is confirmed, the calibration coefficient is calculated. This coefficient is required to accurately convert the voltage value into weight.

```
PRINTF("\r\n-> 2 Step :Coefficient Calculation <-\r\n");
while (confirmation){
        PRINTF("Put an object of known weight on the scale\r\n",knownWeight);
        PRINTF("Please, digit the weight on the scale and click enter\r\n");
        SCANF("%i", &knownWeight); // Wait a keyboard import to start a conversion sequence.
        PRINTF("The weight on the scale is %dg, do you confirm(y/n)?\r\n", knownWeight);
        key = GETCHAR();
        if (key == 'y'){confirmation = false;}
        else{confirmation = true;}
}

startSccr(&instance,conversionNumber); // Start a Single Channel Conversion, in order to get "conversionNumber" results
while(NAFESDK_GetReadingsRemaining(&instance) != 0) {}; // Wait for conversion end.
NAFESDK_CmdAbort(&instance);

voltage = 0; // Initialize result variable
len = NAFESDK_GetRBCount(&instance); // get length of buffer
// convert sample to voltage and sum it in order to perform an average
for(i=0; i<len / ADC_READ_SIZE; i++){
        NAFESDK_ReadFromRB(&instance, buf, 3);
        adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
        sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
        voltage += sINT24_TO_VG(sINT24, gain); //Apply Conversion from 24 bit to voltage value
}
voltage = voltage/conversionNumber; //Average Calculation
coefficient = knownWeight / (voltage - offset);
PRINTF("-> 2 Step : Done <-\r\n\r\n");
```

aaa-054894

Figure 23. Weight scale calibration coefficient calculation

Now the device is calibrated for weight measurement and the user can start taking measurements by pressing any keyboard button, as shown in Figure 24. The NAFE measurement (in millivolts) and the converted weight measurement (in kilograms) is printed on the console.

```
while(1)
{
        PRINTF("->Click any button to start one Conversion sequences<-\r\n");
        GETCHAR(); // Wait a keyboard import to start a conversion sequence.
        startSccr(&instance,conversionNumber); // Start a Single Channel Conversion, in order to get "conversionNumber"
        results
        while(NAFESDK_GetReadingsRemaining(&instance) != 0) {}; // Wait for conversion end.
        NAFESDK_CmdAbort(&instance);
        voltage = 0; // Initialize result variable
        len = NAFESDK_GetRBCount(&instance); // get length of buffer
        // convert sample to voltage and sum it in order to perform an average
        for(i=0; i<len / ADC_READ_SIZE; i++)
        {
                NAFESDK_ReadFromRB(&instance, buf, 3);
                adc_res = (buf[0]<<16) + (buf[1]<<8) + (buf[2]<<0); // convert 24 bit result into int32_t format
                sINT24 = ADC_READING_TO_sINT24(adc_res); //Apply Conversion to 24 bit Integer (see datasheet)
                voltage += sINT24_TO_VG(sINT24, gain); //Apply Conversion from 24 bit to voltage value
        }
        voltage = voltage/conversionNumber; //Average Calculation
        PRINTF("Voltage Load Cell=%6fmV, Weight=%1fg\r\n", voltage*1000,coefficient*(voltage-offset)); //Calculation of
        Temperature.
}
```

aaa-054901

Figure 24. Weight scale measurements

Figure 25 shows the console printout of Offset Calculation (➞ 1 Step) and the Coefficient Calculation (➞ 2 Step), followed by the weight measurement.

```
-> Calculation of Coefficient <-

-> 1 Step :Offset Calculation <-
Leave the weight scale without weight and click any buttons.
-> 1 Step : Done <-

-> 2 Step :Coefficient Calculation <-
Put an object of known weight on the scale
Please, digit the weight on the scale and click enter
The weight on the scale is 233g, do you confirm(y/n)?
-> 2 Step : Done <-

->Click any button to start one Conversion sequences<-
Voltage Load Cell=0.291597mV, Weight=233.000000g
->Click any button to start one Conversion sequences<-
Voltage Load Cell=0.260416mV, Weight=107.261124g
->Click any button to start one Conversion sequences<-
Voltage Load Cell=0.269916mV, Weight=145.568665g
->Click any button to start one Conversion sequences<-
Voltage Load Cell=0.270028mV, Weight=146.019348g
->Click any button to start one Conversion sequences<-
```

aaa-054903

Figure 25. Weight scale console print out

5 Revision history

Table 1. Revision history

Document ID	Release date	Description
AN14103 v.1.0	06 May 2024	Initial version

6 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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 must be provided with the distribution.
3. Neither the name of the copyright holder 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.

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

Suitability for use in industrial applications (functional safety) — This NXP product has been qualified for use in industrial applications. It has been developed in accordance with IEC 61508, and has been SIL-classified accordingly. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

i.MX — is a trademark of NXP B.V.

Kinetis — is a trademark of NXP B.V.

Tower — is a trademark of NXP B.V.

Tables

Figures

Contents

Date of release: 6 May 2024
Document identifier: AN14103