Laboratory 2: Basic I/O and ADC-based Readout of Temperature and Voltage

Karim Elgammal
Computer Engineering, McGill University
260920556

Rafid Saif
Electrical Engineering, McGill University
260903802

***Abstract*—This investigation involves developing a C program for the STM32L4S5 microcontroller to detect the pressing of a button and test the LED2 display by toggling the value every time the button is pressed. The program is then configured to read out the reference voltage and the temperature using the ADC unit, by converting them through their respective calculations. The program control is designed to switch the readouts between the voltage and temperature, and a final program is produced that alternates between the ADC readings, allowing the state of the LED to reflect the procedure that is happening.**

***Index Terms*—STM32L4S5 microcontroller, C program, LED, ADC, reference voltage, temperature, program control**

I. Introduction

The investigation at hand takes on 4 distinctive development parts. Firstly, the development of a program which toggles an LED on the board by control of a push button. Secondly, the ADC unit is configured to read the reference voltage, where the code is augmented to convert the value read from the ADC to a valid voltage value. The third part of this lab is concerned with the same procedure as its predecessor; however, in this step we aim to achieve a valid temperature reading from the ADC. The final part combines all the preceding parts into one program, where the ADC is conditionally configured for temperature readings or for voltage readings, based on the state of the LED, which is controlled by a pushbutton. The Cube MX GUI is used to generate the initial code skeleton for the program, and the reference manual for the STM32L4S5 microcontroller is consulted to ensure correct initialization of readings to avoid errors in the procedure.

II. Methodology and Analysis

A. Step 1: Toggling an LED by a PushButton

This part of the laboratory is concerned with writing a program which detects the pressing of a push button on our board. This detection is then used to toggle the value of the LED2 present on the board. In order to realize this functionality; we firstly initialized and listed the relevant pins on CubeMX, more specifically, we initialized pins PC13 and PB14 as our push button and LED light, respectively. Specifying the relevant pins and setting PC13 as an input and PB14 as an output allows us to use the initialization code which CubeMX generates. Further, by placing labels to the pins, we call the pins in our program by a descriptive name, which increases readability of our code. The generated initializations can be found in figure 1; where the type of pin, mode and pull parameters are set. We structured the program to run in an infinite loop as seen in figure 2; where we check if the button is being pressed through HAL_GPIO_ReadPin. If the button is indeed being pressed, we toggle the LED by calling HAL_GPIO_TogglePin, then wait for the button to be released before continuing. This allows us to only receive proper 'presses' as an input to our button. This design structure is seemingly the simplest implementation of this functionality.

```
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : PBUTTON_Pin */
  GPIO_InitStruct.Pin = PBUTTON_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(PBUTTON_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LED2_Pin */
  GPIO_InitStruct.Pin = LED2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LED2_GPIO_Port, &GPIO_InitStruct);

}
```

Fig. 1. Generated Initialization of Pins

```
while(1)
{
    //First, we check if our button is currently being pressed
    if(HAL_GPIO_ReadPin(PBUTTON_GPIO_Port, PBUTTON_Pin) == GPIO_PIN_RESET)
    {
        //If it is, then we toggle the state of the LED and wait for the button to be unpressed
        HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
        while(HAL_GPIO_ReadPin(PBUTTON_GPIO_Port, PBUTTON_Pin) == GPIO_PIN_RESET)
        {
        }
    }
```

Fig. 2. Toggling LED by PushButton in While Loop

B. Step 2: Using ADC to Measure Reference

The ADC was configured using CubeMX default settings, choosing the VREFINT channel. This sets up the ADC with single conversion mode, without interrupts. Based on the requirements of low power, this is a better option than continuous conversion mode, as that would keep converting far more frequently than we need for a relatively slow-updating application. Using interrupts could be superior to this, however. For the channel configuration, CubeMX automatically selects 2.5 clock cycles and the macro ADC_CHANNEL_VREFINT. The configuration is shown in figure 3.

For the ADC conversion, a function was defined: ADC_GetVal(). This function takes an ADC_HandleTypeDef pointer, starts a conversion, reads the value and then stops the ADC. The function is shown in figure 4. This is an unsigned integer representing the ADC value of the voltage reading. We divide that by the value stored in memory, which represents the factory calibrated VREFINT value at 3.0 V for VREF+. We can thus obtain our calibrated VREF+ by dividing the measured ADC value by the value from memory and multiplying by the reference 3.0 V. Following this procedure, we obtained a VREF+ of between 3.3 V and 3.5 V, which is what we need to consider when scaling future ADC readings.

```
hadc1.Instance = ADC1;
hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
hadc1.Init.Resolution = ADC_RESOLUTION_12B;
hadc1.Init.ScanConvMode = ENABLE;
hadc1.Init.ContinuousConvMode = ENABLE;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DMAContinuousRequests = DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
  Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_VREFINT;
sConfig.Rank = 1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
```

Fig. 3. Internal Voltage ADC Configuration

```
uint32_t ADC_GetVal(ADC_HandleTypeDef* hadc)
{
    HAL_ADC_Start(hadc);
    HAL_ADC_PollForConversion(hadc, 5);
    uint32_t adc_val = HAL_ADC_GetValue(hadc);
    HAL_ADC_Stop(hadc);
    return adc_val;
}
```

Fig. 4. ADC_GetVal() Function

C. Step 3: Using ADC to Measure Temperature

For the ADC conversion, the same procedure was initially followed except the channel chosen was ADC_CHANNEL_TEMPSENSOR. The calibrated ADC readings for TCAL1 at 30 degrees and TCAL2 at 110 degrees were obtained from their memory locations. We obtained a ratio between their ADC value difference and the temperature difference. Multiplying this by the difference between the measured ADC value and the lower reference gives us the relative temperature according to the equation below. Note, the ADC value measured was scaled by the ratio of our VREF+ and the reference VREF+.

Following these above steps, however, gave us negative readings of temperature indoors which is clearly incorrect. In solving this, we realized the datasheet specified a minimum conversion time of 5 us. The default setting for conversion time from CubeMX is 2.5 clock cycles. Going back to the CubeMX clock configuration, we noted the ADC clock frequency is 48 MHz. To obtain the minimum number of clock cycles this represents, we can multiply 48 MHz by 5 us, obtaining 240 clock cycles. Referring to the header file stm32l4xx_hal_adc.h, the options available are 247.5 clock cycles or 640.5 clock cycles. To maintain a safe margin, we chose 640.5 clock cycles, and this indeed gave us around 25-30 degrees indoors.

D. Step 4: Conditionally Switching Between Voltage and Temperature Reading

Finally, within this part of the investigation, we combined the preceding parts into one functional program by way of polling. The way we achieve this is as follows; based on the status of our LED (where on/1 is temperature and off/0 is voltage), we reconfigure the ADC to suit the respective reading. This was done by way of two void container functions which configure the respective channel; either ADC_CHANNEL_VREFINT or ADC_CHANNEL_TEMPSENSOR. The configuration involves setting the sampling time as discussed previously and assigning a rank of 1 to the chosen channel. The configuration container functions can be found in figure 5 and 6. As seen from figure 7, these functions are called before we perform a reading, but after we have verified which 'mode' our program is in; reading temperature or voltage. And so, the respective configuration function is called, and then ADC_GetVal is called in order to fetch the reading from the ADC. The value fetched is then fed into the series of calculations we described previously; in order to convert the reading into a valid temperature or voltage.

Further, it is noted that the temperature reading is more accurate when the voltage is read first; and so upon entering the program we fetch an initial reading for voltage to perform the calculations of temperature accurately. Further, within our running while loop, and within the branch which calculates temperature, we update the value of the voltage based on the most recent reading by calling the line: temp_multiplier = vref / vref_cal; This ensures that we are receiving the most updated voltage value to use within the calculation of the temperature.

```
/* Function Name: ADC_Select_Voltage
 *
 * Description: This function allows us to configure the ADC according to the initializations
 * of the reference voltage channel. The function does so by configuring the channel type/name and then
 * sets the according channel to a rank of 1.
 */
void ADC_Select_Voltage(void)
{
      ADC_ChannelConfTypeDef sConfig = {0};
      sConfig.Channel = ADC_CHANNEL_VREFINT;
      sConfig.Rank = 1;
      sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
      sConfig.SingleDiff = ADC_SINGLE_ENDED;
      sConfig.OffsetNumber = ADC_OFFSET_NONE;
      sConfig.Offset = 0;
      if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
      {
        Error_Handler();
      }
}
```

Fig. 5. Voltage Container Configuration Function

```c
/* Function Name: ADC_Select_Temp
 *
 * Description: This function allows us to configure the ADC according to the initializations
 * of the temperature sensor channel. The function does so by configuring the channel type/name and then
 * sets the according channel to a rank of 1. We also use this function to set the sampling time to 640.5
 */
void ADC_Select_Temp(void)
{
      ADC_ChannelConfTypeDef sConfig = {0};
      sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
      sConfig.Rank = 1;
      sConfig.SamplingTime = ADC_SAMPLETIME_640CYCLES_5;
      sConfig.SingleDiff = ADC_SINGLE_ENDED;
      sConfig.OffsetNumber = ADC_OFFSET_NONE;
      sConfig.Offset = 0;
      if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
      {
          Error_Handler();
      }
}
```

Fig. 6. Temperature Container Configuration Function

```c
while(1)
{
    //First, we check if our button is currently being pressed
    if(HAL_GPIO_ReadPin(PBUTTON_GPIO_Port, PBUTTON_Pin) == GPIO_PIN_RESET)
    {
        //If it is, then we toggle the state of the LED and wait for the button to be unpressed
        HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
        while(HAL_GPIO_ReadPin(PBUTTON_GPIO_Port, PBUTTON_Pin) == GPIO_PIN_RESET)
        {
        }
    }
  //Now, depending on the LED, we switch between conversion modes, when LED is 0 (off), we want to read the
  //voltage, and so we call ADC_Select_Voltage
    if(HAL_GPIO_ReadPin(LED2_GPIO_Port, LED2_Pin) == 0)
    {
        ADC_Select_Voltage();
        vref_int = ADC_GetVal(&hadc1);
        vref = vref_cal * vref_int_cal / vref_int; // Based on manual
    //When LED is 1 (on) we want to read the temperature, and so, we call ADC_Select_Temp before commencing conversion as above
    } else
    {
        temp_multiplier = vref / vref_cal;
        ADC_Select_Temp();
        float ts_data = ADC_GetVal(&hadc1); // Implicit ..... to ......
        float ts_relative = temp_multiplier * ts_data - ts_cal1; // Relative to ts_cal1
        temp = temp_range / ts_diff * ts_relative + temp_min; // Based on manual
    }
}
```

Fig. 7. Overall Program Main Code Segment

III. CONCLUSION

In conclusion, the results demonstrated above clearly show appropriate ADC functioning and correct interpretation of these results to get voltage and internal temperature values. It is important to note the configuration procedure for the ADC, as well as important factors such as sampling time; much of this information can be found in the corresponding HAL library header files. The investigation demonstrates the utility of STM32CubeMX in configuration and setting up of various peripherals not limited to the ADC.