Development of an Electrostatic Energy Harvester for Implantable Devices

Auteur : Irabor, George
Promoteur(s) : Redouté, Jean-Michel
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil électricien, à finalité spécialisée en "electronic systems and devices"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/20233

University of Liège - School of Engineering and Computer Science

Development of an Electrostatic Energy Harvester for Implantable Devices Annexe

An Autonomous Approach to Powering Medical Implants

Supervisor: Dr. Jean-Michel Rédouté

Master's thesis completed in order to obtain the degree of Master of Science in Electrical Engineering

by George Irabor

Academic year 2023-2024

Appendix A

Full Program Implementation

```
1
2
3
4       /* USER CODE BEGIN Header */
5   /* USER CODE END Header */
6   /* Includes
        ------------------------------------------------------------------*/
7   #include "main.h"
8
9   /* Private includes
        ----------------------------------------------------------*/
10  /* USER CODE BEGIN Includes */
11  #include <stdio.h>
12  #include <string.h>
13  #include <stdbool.h>
14  /* USER CODE END Includes */
15
16  /* Private typedef
        -----------------------------------------------------------*/
17  /* USER CODE BEGIN PTD */
18  // Enumerate system states
19  typedef enum {
20      INVESTMENT,
21      HARVESTING,
22      REIMBURSEMENT,
23      RECOVERY
24  } SystemState;
25  /* USER CODE END PTD */
26
27  /* Private define
        ------------------------------------------------------------*/
28  /* USER CODE BEGIN PD */
```

```
29 #define ADC_BUF_LEN 79 //Recovery phase buffer
30 #define MAX_CAP_THRESHOLD 3000  // Set based on observed values
31 #define MIN_CAP_THRESHOLD 200    // Set at 200 to switch just
      before reaching the maximum
32 #define EPSILON 20          // Small buffer to prevent frequent
      toggling near threshold values
33 //#define SAMPLES 5  // Number of samples for averaging
34
35 /* USER CODE END PD */
36
37 /* Private macro
      -------------------------------------------------------------*/
38 /* USER CODE BEGIN PM */
39 /* USER CODE END PM */
40
41 /* Private variables
      ---------------------------------------------------------*/
42 ADC_HandleTypeDef hadc1;
43 DMA_HandleTypeDef hdma_adc1;
44
45 TIM_HandleTypeDef htim2;
46 TIM_HandleTypeDef htim16;
47
48 UART_HandleTypeDef huart2;
49
50 /* USER CODE BEGIN PV */
51 uint16_t adc_buf[ADC_BUF_LEN];
52 volatile uint16_t maxValue;
53 volatile uint16_t maxVal;
54 volatile uint16_t HarvestVoltage;
55 volatile uint32_t smoothedValue;
56 SystemState system_state = RECOVERY;  // Initial state
57 uint32_t adcValues[SAMPLES];
58 uint8_t sampleIndex = 0;
59 bool bufferFull = false;
60
61 /* Variable to report ADC analog watchdog status:   */
62 /*   RESET <=> voltage into AWD window   */
63 /*   SET   <=> voltage out of AWD window */
64 volatile uint8_t  ubAnalogWatchdogStatus = RESET;  /* Set into
      analog watchdog interrupt callback */
65 /* USER CODE END PV */
66
67 /* Private function prototypes
      -----------------------------------------------*/
68 void SystemClock_Config(void);
69 static void MX_GPIO_Init(void);
70 static void MX_DMA_Init(void);
71 static void MX_USART2_UART_Init(void);
```

```
72 static void MX_TIM2_Init(void);
73 static void MX_TIM16_Init(void);
74 static void MX_ADC1_Init(void);
75 /* USER CODE BEGIN PFP */
76 /* USER CODE END PFP */
77
78 /* Private user code
   ---------------------------------------------------------*/
79 /* USER CODE BEGIN 0 */
80 void MicroDelay(uint16_t microseconds)
81 {
82     __HAL_TIM_SET_COUNTER(&htim16, 0);  // if htim16 is the timer
       instance
83     __HAL_TIM_ENABLE(&htim16);
84     while (__HAL_TIM_GET_COUNTER(&htim16) < microseconds);
85     __HAL_TIM_DISABLE(&htim16);
86 }
87
88 void GeneratePulse(void) {
89     // Start ADC conversion
90     HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buf, ADC_BUF_LEN);
91     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET); //Connect
       Op amp output to ADC
92     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, GPIO_PIN_SET); // Start
       charging the capacitor
93
94       //Start the ADC and take many samples with the DMA
95     MicroDelay(10); // Charge for a very short time
96     //Check the values and choose the largest value
97     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, GPIO_PIN_RESET); //
       Discharge the capacitor
98     // Stop ADC conversion
99
100    HAL_ADC_Stop_DMA(&hadc1);
101    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET);
       //disconnect Op amp output to ADC
102    processData();
103    HAL_Delay(500); //Delay before sending the next pulse
104
105 }
106
107 void processData() {
108     maxVal = 0;
109     for (int i = 0; i < 38; i++) {
110         if (adc_buf[i] > maxVal) maxVal = adc_buf[i];
111
112     }
113    uint16_t max = 1 + ((maxVal - 1200) * (1000 - 1)) / (1800 -
       1200); //Values set based on observed maximum and minimum
```

```
    if (max <= 0){
        max = 0;
    }

    maxValue = max;
    if (maxValue > 4000){
        maxValue = 9999; //For Debugging
    }

}



void Update_GPIO_States(SystemState state) {
    switch (state) {
    //System states for use on the PCB directly. They are unused
       here as the PCB design is not optimal
    case RECOVERY:
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
        break;
    case INVESTMENT:
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
        break;
    case HARVESTING:
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
        break;
    case REIMBURSEMENT:
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
        //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
        break;
    }
}
//Function to smooth out ADC values
uint32_t getSmoothedValue(uint32_t newValue) {
    adcValues[sampleIndex] = newValue;
    sampleIndex++;
    if (sampleIndex >= SAMPLES) {
        sampleIndex = 0;
        bufferFull = true;
    }

    uint32_t sum = 0;
    uint32_t count = bufferFull ? SAMPLES : sampleIndex;
    for (uint8_t i = 0; i < count; i++) {
        sum += adcValues[i];
    }
```

```
162
163     return sum / count;  // Return the average
164 }
165 /* USER CODE END 0 */
166
167 /**
168   * @brief  The application entry point.
169   * @retval int
170   */
171 int main(void)
172 {
173   /* USER CODE BEGIN 1 */
174
175   /* USER CODE END 1 */
176
177   /* MCU
        Configuration--------------------------------------------------------*/
178
179   /* Reset of all peripherals, Initializes the Flash interface
        and the Systick. */
180   HAL_Init();
181
182   /* USER CODE BEGIN Init */
183   /* USER CODE END Init */
184
185   /* Configure the system clock */
186   SystemClock_Config();
187
188   /* USER CODE BEGIN SysInit */
189   /* USER CODE END SysInit */
190
191   /* Initialize all configured peripherals */
192   MX_GPIO_Init();
193   MX_DMA_Init();
194   MX_USART2_UART_Init();
195   MX_TIM2_Init();
196   MX_TIM16_Init();
197   MX_ADC1_Init();
198   /* USER CODE BEGIN 2 */
199
200
201
202
203   /* USER CODE END 2 */
204
205   /* Infinite loop */
206   /* USER CODE BEGIN WHILE */
207   while (1)
208     {
```

```
209         switch (system_state) {
210             case RECOVERY:
211 
212                 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5,
                        GPIO_PIN_RESET); //Keep reimbursement closed
213                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12,
                        GPIO_PIN_SET); // Switch for charging the
                        capacitor
214                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9,
                        GPIO_PIN_SET);  // Measurement is enabled
215 
216                 GeneratePulse();
217                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12,
                        GPIO_PIN_RESET); // Switch for charging the
                        capacitor
218                 if (maxValue <= (MIN_CAP_THRESHOLD + EPSILON)) {
219                     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9,
                            GPIO_PIN_RESET); // Isolate measurement
                            circuit
220                     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6,
                            GPIO_PIN_RESET); //Disconnect Opamp from
                            ADC to prevent damage
221                     system_state = INVESTMENT;
222                 }
223                 break;
224 
225             case INVESTMENT:
226 
227                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7,
                        GPIO_PIN_SET);  // Start biasing capacitor
228                 HAL_Delay(50);  // Ensure full charge
229                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12,
                        GPIO_PIN_RESET); // Isolate capacitor
230                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7,
                        GPIO_PIN_RESET);  // Turn off biasing
231 
232                 system_state = HARVESTING;
233                 break;
234 
235             case HARVESTING:
236 //                HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buf,
        ADC_BUF_LEN);
237                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9,
                        GPIO_PIN_SET);  // Enable measurement
238                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6,
                        GPIO_PIN_RESET); //Keep ADC from damage
239                 HAL_Delay(1000); //Wait for a while to harvest
240                 system_state = REIMBURSEMENT;
241                 break;
```

```
242
243             case REIMBURSEMENT:
244                 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5,
                        GPIO_PIN_RESET); // Discharge into storage
                        capacitor
245                 HAL_Delay(100);  // Ensure discharge
246                 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5,
                        GPIO_PIN_RESET); // Turn off discharge
247                 system_state = RECOVERY;
248                 HAL_Delay(1000);
249                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9,
                        GPIO_PIN_RESET); //Measurement is disabled
250                 break;
251         }
252
253         // General GPIO or control updates
254         Update_GPIO_States(system_state);
255
256     /* USER CODE END WHILE */
257
258     /* USER CODE BEGIN 3 */
259     }
260   /* USER CODE END 3 */
261 }
262
263 /**
264   * @brief System Clock Configuration
265   * @retval None
266   */
267 void SystemClock_Config(void)
268 {
269   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
270   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
271
272   /** Configure the main internal regulator output voltage
273   */
274   if
        (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1)
        != HAL_OK)
275   {
276     Error_Handler();
277   }
278
279   /** Configure LSE Drive Capability
280   */
281   HAL_PWR_EnableBkUpAccess();
282   __HAL_RCC_LSEDRIVE_CONFIG(RCC_LSEDRIVE_LOW);
283
284   /** Initializes the RCC Oscillators according to the specified
```

```
     parameters
285    * in the RCC_OscInitTypeDef structure.
286    */
287    RCC_OscInitStruct.OscillatorType =
         RCC_OSCILLATORTYPE_LSE|RCC_OSCILLATORTYPE_MSI;
288    RCC_OscInitStruct.LSEState = RCC_LSE_ON;
289    RCC_OscInitStruct.MSIState = RCC_MSI_ON;
290    RCC_OscInitStruct.MSICalibrationValue = 0;
291    RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_10;
292    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
293    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
294    {
295      Error_Handler();
296    }
297
298    /** Initializes the CPU, AHB and APB buses clocks
299    */
300    RCC_ClkInitStruct.ClockType =
         RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
301                                  |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
302    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_MSI;
303    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
304    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
305    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
306
307    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) !=
         HAL_OK)
308    {
309      Error_Handler();
310    }
311
312    /** Enable MSI Auto calibration
313    */
314    HAL_RCCEx_EnableMSIPLLMode();
315  }
316
317  /**
318    * @brief ADC1 Initialization Function
319    * @param None
320    * @retval None
321    */
322  static void MX_ADC1_Init(void)
323  {
324
325    /* USER CODE BEGIN ADC1_Init 0 */
326
327    /* USER CODE END ADC1_Init 0 */
328
329    ADC_MultiModeTypeDef multimode = {0};
```

```
330   ADC_ChannelConfTypeDef sConfig = {0};
331
332   /* USER CODE BEGIN ADC1_Init 1 */
333
334   /* USER CODE END ADC1_Init 1 */
335
336   /** Common config
337   */
338   hadc1.Instance = ADC1;
339   hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
340   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
341   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
342   hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
343   hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
344   hadc1.Init.LowPowerAutoWait = DISABLE;
345   hadc1.Init.ContinuousConvMode = ENABLE;
346   hadc1.Init.NbrOfConversion = 1;
347   hadc1.Init.DiscontinuousConvMode = DISABLE;
348   hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
349   hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
350   hadc1.Init.DMAContinuousRequests = ENABLE;
351   hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
352   hadc1.Init.OversamplingMode = DISABLE;
353   if (HAL_ADC_Init(&hadc1) != HAL_OK)
354   {
355     Error_Handler();
356   }
357
358   /** Configure the ADC multi-mode
359   */
360   multimode.Mode = ADC_MODE_INDEPENDENT;
361   if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) !=
        HAL_OK)
362   {
363     Error_Handler();
364   }
365
366   /** Configure Regular Channel
367   */
368   sConfig.Channel = ADC_CHANNEL_8;
369   sConfig.Rank = ADC_REGULAR_RANK_1;
370   sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
371   sConfig.SingleDiff = ADC_SINGLE_ENDED;
372   sConfig.OffsetNumber = ADC_OFFSET_NONE;
373   sConfig.Offset = 0;
374   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
375   {
376     Error_Handler();
377   }
```

```
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief TIM2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */
  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_IC_InitTypeDef sConfigIC = {0};

  /* USER CODE BEGIN TIM2_Init 1 */
  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 0;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 4294967295;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) !=
     HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_IC_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2,
     &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
```

```
425   }
426   sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
427   sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
428   sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
429   sConfigIC.ICFilter = 0;
430   if (HAL_TIM_IC_ConfigChannel(&htim2, &sConfigIC, TIM_CHANNEL_1)
          != HAL_OK)
431   {
432     Error_Handler();
433   }
434   /* USER CODE BEGIN TIM2_Init 2 */
435   /* USER CODE END TIM2_Init 2 */
436
437 }
438
439 /**
440   * @brief TIM16 Initialization Function
441   * @param None
442   * @retval None
443   */
444 static void MX_TIM16_Init(void)
445 {
446
447   /* USER CODE BEGIN TIM16_Init 0 */
448   /* USER CODE END TIM16_Init 0 */
449
450   /* USER CODE BEGIN TIM16_Init 1 */
451   /* USER CODE END TIM16_Init 1 */
452   htim16.Instance = TIM16;
453   htim16.Init.Prescaler = 7;
454   htim16.Init.CounterMode = TIM_COUNTERMODE_UP;
455   htim16.Init.Period = 65535;
456   htim16.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
457   htim16.Init.RepetitionCounter = 0;
458   htim16.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
459   if (HAL_TIM_Base_Init(&htim16) != HAL_OK)
460   {
461     Error_Handler();
462   }
463   /* USER CODE BEGIN TIM16_Init 2 */
464   /* USER CODE END TIM16_Init 2 */
465
466 }
467
468 /**
469   * @brief USART2 Initialization Function
470   * @param None
471   * @retval None
472   */
```

```
473 static void MX_USART2_UART_Init(void)
474 {
475
476   /* USER CODE BEGIN USART2_Init 0 */
477   /* USER CODE END USART2_Init 0 */
478
479   /* USER CODE BEGIN USART2_Init 1 */
480   /* USER CODE END USART2_Init 1 */
481   huart2.Instance = USART2;
482   huart2.Init.BaudRate = 115200;
483   huart2.Init.WordLength = UART_WORDLENGTH_8B;
484   huart2.Init.StopBits = UART_STOPBITS_1;
485   huart2.Init.Parity = UART_PARITY_NONE;
486   huart2.Init.Mode = UART_MODE_TX_RX;
487   huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
488   huart2.Init.OverSampling = UART_OVERSAMPLING_16;
489   huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
490   huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
491   if (HAL_UART_Init(&huart2) != HAL_OK)
492   {
493     Error_Handler();
494   }
495   /* USER CODE BEGIN USART2_Init 2 */
496   /* USER CODE END USART2_Init 2 */
497
498 }
499
500 /**
501   * Enable DMA controller clock
502   */
503 static void MX_DMA_Init(void)
504 {
505
506   /* DMA controller clock enable */
507   __HAL_RCC_DMA1_CLK_ENABLE();
508
509   /* DMA interrupt init */
510   /* DMA1_Channel1_IRQn interrupt configuration */
511   HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 0, 0);
512   HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);
513
514 }
515
516 /**
517   * @brief GPIO Initialization Function
518   * @param None
519   * @retval None
520   */
521 static void MX_GPIO_Init(void)
```

```
522 {
523   GPIO_InitTypeDef GPIO_InitStruct = {0};
524 /* USER CODE BEGIN MX_GPIO_Init_1 */
525 /* USER CODE END MX_GPIO_Init_1 */
526
527   /* GPIO Ports Clock Enable */
528   __HAL_RCC_GPIOC_CLK_ENABLE();
529   __HAL_RCC_GPIOA_CLK_ENABLE();
530   __HAL_RCC_GPIOB_CLK_ENABLE();
531
532   /*Configure GPIO pin Output Level */
533   HAL_GPIO_WritePin(GPIOA,
      SW4_Pin|Pulse_Out_Pin|GPIO_PIN_8|Outswitch2_Pin
534                         |GPIO_PIN_10|OutSwitch1_Pin,
                             GPIO_PIN_RESET);
535
536   /*Configure GPIO pin Output Level */
537   HAL_GPIO_WritePin(GPIOB, SW1_Pin|LD3_Pin|SW3_Pin|CHRG_EN_Pin
538                         |SW2_Pin, GPIO_PIN_RESET);
539
540   /*Configure GPIO pins : SW4_Pin Pulse_Out_Pin PA8 Outswitch2_Pin
541                            PA10 OutSwitch1_Pin */
542   GPIO_InitStruct.Pin =
      SW4_Pin|Pulse_Out_Pin|GPIO_PIN_8|Outswitch2_Pin
543                         |GPIO_PIN_10|OutSwitch1_Pin;
544   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
545   GPIO_InitStruct.Pull = GPIO_NOPULL;
546   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
547   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
548
549   /*Configure GPIO pins : SW1_Pin LD3_Pin SW3_Pin CHRG_EN_Pin
550                            SW2_Pin */
551   GPIO_InitStruct.Pin = SW1_Pin|LD3_Pin|SW3_Pin|CHRG_EN_Pin
552                         |SW2_Pin;
553   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
554   GPIO_InitStruct.Pull = GPIO_NOPULL;
555   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
556   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
557
558   /*Configure GPIO pin : PA11 */
559   GPIO_InitStruct.Pin = GPIO_PIN_11;
560   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
561   GPIO_InitStruct.Pull = GPIO_NOPULL;
562   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
563   GPIO_InitStruct.Alternate = GPIO_AF12_COMP1;
564   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
565
566 /* USER CODE BEGIN MX_GPIO_Init_2 */
567 /* USER CODE END MX_GPIO_Init_2 */
```

```
568 }
569
570 /* USER CODE BEGIN 4 */
571 //Called when first half of buffer is filled
572 void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc){
573     if (hadc->Instance == ADC1) {
574         HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_10);
575       }
576 }
577
578
579
580 //Called when buffer is completely filled
581 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
582     if (hadc->Instance == ADC1) {
583         HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_10);
584     }
585 }
586
587 void HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef* hadc)
588 {
589      /* Set variable to report analog watchdog out of window
           status to main      */
590      /* program.

           */
591      //HarvestVoltage = HAL_ADC_GetValue(&hadc2);
592      ubAnalogWatchdogStatus = SET;
593 }
594
595 /* USER CODE END 4 */
596
597 /**
598   * @brief  This function is executed in case of error occurrence.
599   * @retval None
600   */
601 void Error_Handler(void)
602 {
603   /* USER CODE BEGIN Error_Handler_Debug */
604   /* USER CODE END Error_Handler_Debug */
605 }
606
607 #ifdef  USE_FULL_ASSERT
608 /**
609   * @brief  Reports the name of the source file and the source
        line number
610   *         where the assert_param error has occurred.
611   * @param  file: pointer to the source file name
612   * @param  line: assert_param error line source number
```

```
613   * @retval None
614   */
615 void assert_failed(uint8_t *file, uint32_t line)
616 {
617   /* USER CODE BEGIN 6 */
618   /* USER CODE END 6 */
619 }
620 #endif /* USE_FULL_ASSERT */
```

Listing A.1: HAL C code written for STM32L412KB NucleoBoard for ElectroStatic Energy Harvesting