```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2021 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/*
 * Timers:
 *
 * htim2 1kHz burst repetition period
 * htim3 1kHz energy income simulator
 * htim4 1MHz auxiliary timer
 * htim6 10MHz OOK modulation generator
 *
 * Missing parts:
 * Timer initialization, ARR to fit the right frequency/period
 * Clean the UART_TX with predefined strings/chars
 */

/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

#include <stdlib.h>

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
typedef enum { false, true } bool;

typedef enum
{
    WAIT = 0x00,
    DATA_TX = 0x03,
    DATA_RX = 0x04
}TX_Status;

typedef enum
{
    BURST_WAIT = 0x00, //0
    BURST_TX = 0x01, //1
    BURST_RX = 0x02, //2
}MOD_Status;

/* USER CODE END PTD */
```

```
62
63 /* Private define ------------------------------------------------------------*/
64 /* USER CODE BEGIN PD */
65 #define NODE                2       // node ID
66
67 #define ENERGY_STORAGE      100     // energy storage capacity
68 #define ENERGY_RX           70      // energy consumption for data TX, energy in RX is a
   half
69 #define ENERGY_UPDATE       500     // energy update period in ms
70 #define ENERGY_CHANGE       50      // energy variation parameter
71 #define ENERGY_INCREMENT    5       // energy maximum increment
72
73 #define DATA_TX_TIME        50000   // data transmission time in us (max 65535 aka 65ms)
74
75 #define BURST_REP           1000    // burst repetition period in ms
76 #define OOK_FREQ            30      // OOK modulation frequency in kHz
77 #define SHORT_BURST         64      // 64 pulses for short burst (2.2ms)
78 #define MIDDLE_BURST        128     // 128 pulses for middle burst (4.3ms)
79 #define LONG_BURST          256     // 256 pulses for long burst (8.6ms)
80 #define BURST_GUARD         40      // maximum lost pulses in burst rx
81 #define TIMEOUT             100     // timeout for the burst rx pulses (us)
82
83 #define T_BURST_TX          110     // message to send when data tx starts
84 #define T_BURST_RX          111     // message to send when burst rx starts
85 #define T_DATA_TX_START     112     // message to send when data tx starts
86 #define T_DATA_RX_OK        113     // message to send when data tx starts
87 #define T_DATA_ERROR        114     // message to send when data tx starts
88 #define T_DATA_ABORT        115     // message to send when the data tx is aborted
89 #define T_CONTROL           200     // control for debug
90
91 /* USER CODE END PD */
92
93 /* Private macro -------------------------------------------------------------*/
94 /* USER CODE BEGIN PM */
95
96 /* USER CODE END PM */
97
98 /* Private variables ---------------------------------------------------------*/
99 TIM_HandleTypeDef htim2;
100 TIM_HandleTypeDef htim3;
101 TIM_HandleTypeDef htim4;
102 TIM_HandleTypeDef htim6;
103
104 UART_HandleTypeDef huart2;
105
106 /* USER CODE BEGIN PV */
107 uint8_t energy_level = 0;
108
109 TX_Status stat = WAIT;
110 MOD_Status mod = WAIT;
111
112 uint16_t count = 0;
113 uint16_t pulses = 0;
114 uint16_t burst_length = 0;
115 uint16_t energy_count = 0;
116 uint16_t energy_count_limit = 0;
117 uint8_t energy_increment = 0;
118
119 /* USER CODE END PV */
120
121 /* Private function prototypes -----------------------------------------------*/
```

```c
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM6_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM4_Init(void);
/* USER CODE BEGIN PFP */
void uart_tx(uint8_t);

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM2_Init();
  MX_USART2_UART_Init();
  MX_TIM6_Init();
  MX_TIM3_Init();
  MX_TIM4_Init();
  /* USER CODE BEGIN 2 */

  HAL_Delay(100);

  srandom(NODE);

  energy_count_limit = (ENERGY_CHANGE/2) + (random() % (ENERGY_CHANGE/2 + 1));
  energy_increment = random() % (ENERGY_INCREMENT + 1);

  // Initiate ARR for timers

```

```
183   burst_length = SHORT_BURST*2;
184
185   TIM6->ARR = 10000 /(2*OOK_FREQ); // Timer @ 10MHz -> OOK_FREQ in kHz
186
187   TIM2->ARR = BURST_REP;
188
189   TIM3->ARR = ENERGY_UPDATE;
190
191   while(HAL_GPIO_ReadPin(GPIOC, B1_Pin));
192
193   HAL_TIM_Base_Start_IT(&htim2);
194   HAL_TIM_Base_Start_IT(&htim3);
195
196   /* USER CODE END 2 */
197
198   /* Infinite loop */
199   /* USER CODE BEGIN WHILE */
200   while (1)
201   {
202       if(energy_level == ENERGY_STORAGE)
203       {
204           burst_length = LONG_BURST*2;
205       }
206       else if(energy_level >= ENERGY_RX)
207       {
208           burst_length = MIDDLE_BURST*2;
209       }
210       else if(energy_level < ENERGY_RX)
211       {
212           burst_length = SHORT_BURST*2;
213       }
214
215       if (energy_count >= energy_count_limit){
216           energy_count_limit = (ENERGY_CHANGE/2) + (random() % (ENERGY_CHANGE/2 + 1));
217           energy_increment = random() % (ENERGY_INCREMENT + 1);
218           energy_count = 0;
219       }
220
221
222
223       // With TRAP data TX only if the received burst is MIDDLE_BURST
224
225           if((energy_level == ENERGY_STORAGE) && (stat != DATA_RX) && (count >=
MIDDLE_BURST))
226           {
227
228               TIM4->CNT = 0;
229               TIM4->ARR = DATA_TX_TIME;
230               HAL_TIM_Base_Start_IT(&htim4);
231
232               HAL_GPIO_WritePin(TX_Line_GPIO_Port, TX_Line_Pin, GPIO_PIN_SET);
233
234               energy_level = 0;
235               count = 0;
236
237               stat = DATA_TX;
238               mod = BURST_WAIT;
239
240               uart_tx(T_DATA_TX_START);
241
242           }
```

```
243
244 /*
245
246       // Without TRAP the policy is simple, as soon as energy level == 100 start data
    transmission
247
248
249       if((energy_level == ENERGY_STORAGE) && (stat != DATA_RX))
250       {
251
252           TIM4->CNT = 0;
253           TIM4->ARR = DATA_TX_TIME;
254           HAL_TIM_Base_Start_IT(&htim4);
255
256           HAL_GPIO_WritePin(TX_Line_GPIO_Port, TX_Line_Pin, GPIO_PIN_SET);
257
258           energy_level = 0;
259           count = 0;
260
261           stat = DATA_TX;
262
263           uart_tx(T_DATA_TX_START);
264
265       }
266       */
267
268           /* USER CODE END WHILE */
269
270     /* USER CODE BEGIN 3 */
271   }
272   /* USER CODE END 3 */
273 }
274
275 /**
276   * @brief System Clock Configuration
277   * @retval None
278   */
279 void SystemClock_Config(void)
280 {
281   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
282   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
283   RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};
284
285   /** Initializes the RCC Oscillators according to the specified parameters
286   * in the RCC_OscInitTypeDef structure.
287   */
288   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
289   RCC_OscInitStruct.HSIState = RCC_HSI_ON;
290   RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
291   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
292   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
293   RCC_OscInitStruct.PLL.PLLM = 1;
294   RCC_OscInitStruct.PLL.PLLN = 10;
295   RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
296   RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
297   RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
298   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
299   {
300     Error_Handler();
301   }
302   /** Initializes the CPU, AHB and APB buses clocks
```

```
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
  {
    Error_Handler();
  }
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2;
  PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure the main internal regulator output voltage
  */
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @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};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 40000;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 4294967295;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
  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();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
```

```
364   if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
365   {
366     Error_Handler();
367   }
368   /* USER CODE BEGIN TIM2_Init 2 */
369
370   /* USER CODE END TIM2_Init 2 */
371
372 }
373
374 /**
375   * @brief TIM3 Initialization Function
376   * @param None
377   * @retval None
378   */
379 static void MX_TIM3_Init(void)
380 {
381
382   /* USER CODE BEGIN TIM3_Init 0 */
383
384   /* USER CODE END TIM3_Init 0 */
385
386   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
387   TIM_MasterConfigTypeDef sMasterConfig = {0};
388
389   /* USER CODE BEGIN TIM3_Init 1 */
390
391   /* USER CODE END TIM3_Init 1 */
392   htim3.Instance = TIM3;
393   htim3.Init.Prescaler = 40000;
394   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
395   htim3.Init.Period = 65535;
396   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
397   htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
398   if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
399   {
400     Error_Handler();
401   }
402   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
403   if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
404   {
405     Error_Handler();
406   }
407   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
408   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
409   if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
410   {
411     Error_Handler();
412   }
413   /* USER CODE BEGIN TIM3_Init 2 */
414
415   /* USER CODE END TIM3_Init 2 */
416
417 }
418
419 /**
420   * @brief TIM4 Initialization Function
421   * @param None
422   * @retval None
423   */
424 static void MX_TIM4_Init(void)
```

```
{

  /* USER CODE BEGIN TIM4_Init 0 */

  /* USER CODE END TIM4_Init 0 */

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

  /* USER CODE BEGIN TIM4_Init 1 */

  /* USER CODE END TIM4_Init 1 */
  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 80;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 65535;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM4_Init 2 */

  /* USER CODE END TIM4_Init 2 */

}

/**
  * @brief TIM6 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM6_Init(void)
{

  /* USER CODE BEGIN TIM6_Init 0 */

  /* USER CODE END TIM6_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 7;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 65535;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
```

```
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */

  /* USER CODE END TIM6_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
```

```
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, MOD_OUT_Pin|TX_Line_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : MOD_OUT_Pin TX_Line_Pin */
  GPIO_InitStruct.Pin = MOD_OUT_Pin|TX_Line_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*Configure GPIO pins : RX_Line_Pin MOD_IN_Pin */
  GPIO_InitStruct.Pin = RX_Line_Pin|MOD_IN_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);

}

/* USER CODE BEGIN 4 */

void uart_tx(uint8_t tx)
{

    if(tx == T_BURST_TX)
    {
        HAL_UART_Transmit(&huart2, &energy_level, 1, 10);
    }
    else
    {
        HAL_UART_Transmit(&huart2, &tx, 1, 10);
    }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    // Interrupt handler for burst RX
```

```
608
609     if((GPIO_Pin == MOD_IN_Pin) && (energy_level == ENERGY_STORAGE) && (mod != BURST_TX) &&
    (stat == WAIT))
610     {
611         mod = BURST_RX;
612 
613         if (count == 0)
614         {
615             TIM4->ARR = TIMEOUT;
616             HAL_TIM_Base_Start_IT(&htim4);
617         }
618 
619         TIM4->CNT = 0;
620         count++;
621     }
622     else
623     {
624         count = 0;
625     }
626 
627     // Interrupt handler for data TX-RX
628 
629     if(GPIO_Pin == RX_Line_Pin)
630     {
631         if(energy_level >= ENERGY_RX)
632         {
633             TIM4->CNT = 0;
634             TIM4->ARR = DATA_TX_TIME;
635 
636             HAL_TIM_Base_Start_IT(&htim4);
637 
638             stat = DATA_RX;
639             energy_level = energy_level - ENERGY_RX;
640         }
641         else
642         {
643             uart_tx(T_DATA_ERROR);
644 
645             stat = WAIT;
646 
647             energy_level = 0;
648         }
649     }
650 }
651 
652 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
653 {
654 
655     // Burst generation interrupt handler
656 
657     if(htim == &htim6)
658     {
659         HAL_GPIO_TogglePin(MOD_OUT_GPIO_Port, MOD_OUT_Pin);
660         pulses++;
661 
662         if (pulses == burst_length)
663         {
664             HAL_TIM_Base_Stop_IT(&htim6);
665             pulses = 0;
666             mod = BURST_WAIT;
667         }
```

```
668
669     }
670
671     // Burst repetition interrupt handler
672
673     if(htim == &htim2)
674     {
675         TIM6->CNT = 0;
676         pulses = 0;
677
678         mod = BURST_TX;
679
680         uart_tx(T_BURST_TX);
681
682         HAL_TIM_Base_Start_IT(&htim6);
683     }
684
685     // Energy update interrupt handler
686
687     if(htim == &htim3)
688     {
689         if(stat == WAIT)
690         {
691             uint8_t energy_step = random() % (energy_increment + 1);
692             energy_level = energy_level + energy_step;
693             if(energy_level >= ENERGY_STORAGE) energy_level = ENERGY_STORAGE;
694             energy_count++;
695
696             energy_step = 120 + energy_step;
697
698             HAL_UART_Transmit(&huart2, &energy_step, 1, 10);
699         }
700     }
701
702     // Auxiliary timer interrupt. BURST_RX TIMEOUT, DATA_TX, DATA_RX
703
704     if(htim == &htim4)
705     {
706
707         // Reached the timeout for burst reception
708
709         if(mod == BURST_RX)
710         {
711             mod = BURST_WAIT;
712             count = 0;
713             HAL_TIM_Base_Stop_IT(&htim4);
714         }
715
716         // Finish DATA_TX
717
718         if(stat == DATA_TX)
719         {
720             HAL_TIM_Base_Stop_IT(&htim4);
721             stat = WAIT;
722
723             HAL_GPIO_WritePin(TX_Line_GPIO_Port, TX_Line_Pin, GPIO_PIN_RESET);
724         }
725
726         // Finish DATA_RX
727
728         if(stat == DATA_RX)
```

```
        {
            stat = WAIT;
            uart_tx(T_DATA_RX_OK);
        }
    }
}


/*
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{

    if(Rx == T_DATA_TX_START)
    {
        if(energy_level < ENERGY_STORAGE/2)
        {
            uart_tx(T_DATA_ERROR);
        }
        else
        {
            HAL_TIM_Base_Start_IT(&htim4);
            TIM4->CNT = 0;
            TIM4->ARR = DATA_TX_TIME;
            stat = DATA_RX;
            energy_level = energy_level - (ENERGY_STORAGE/2);
        }
    }
}
*/

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
```

```
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```