Oświetlenie pojazdu elektrycznego przy użyciu magistrali CAN

Sergiusz Grześkowiak
Michał Kowalewski
Łukasz Jaskulski

2018

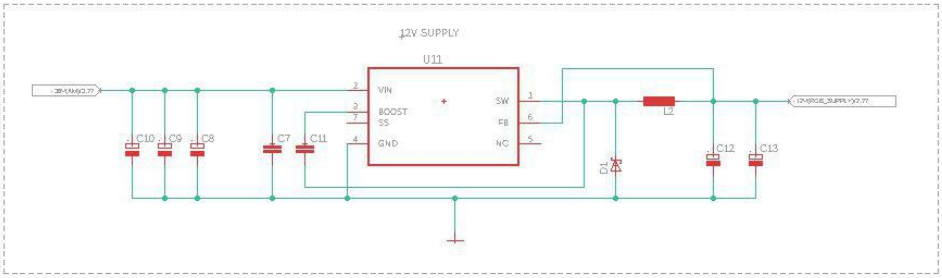
Stworzono przy użyciu LaTeX

Spis treści

1 Założenia i schematy

1.1 Kierunkowskazy

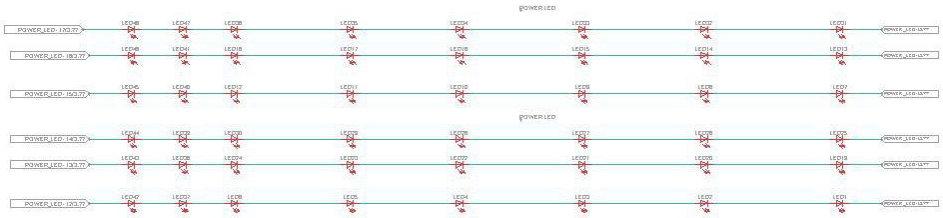
Projektując kierunkowskazy do pojazdu elektrycznego wzorowaliśmy się na dizajnie marki Audi. Gdzie w nowych modelach stosowane są dynamicznie płynące światła, jest to według nas najciekawsze rozwiązanie stawiające przed nami nowe wyzwania i możliwości. Problemem w sterowaniu tymi światłami jest to że każda pojedyncza dioda każdego kierunkowskazu musi posiadać osobne sterowanie barwą i natężeniem światła (w celu uzyskania płynnego efektu migania kierunkowskazu), ponad to światła te muszą spełniać rolę świateł do jazdy dziennej. Problemy te rozwiązaliśmy stosując paski ledowe oparte na diodach WS2811, które to diody zawierają wbudowany prosty układ scalony który umożliwia sterowanie wszystkimi diodami za pomocą jednego wyjścia mikrokontrolera.



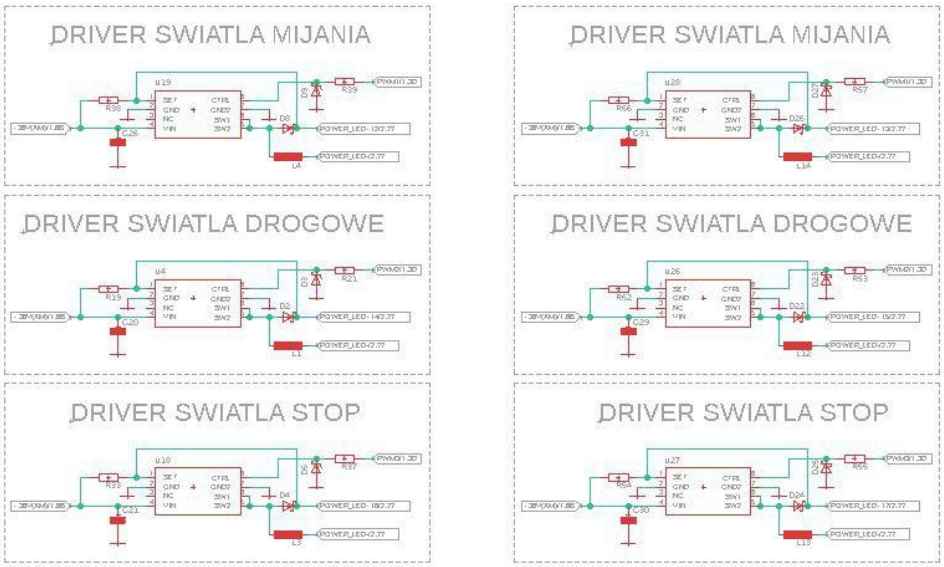
Rysunek 1: Zasilanie +12V DC programowalnych diod rgb

1.2 Światła mijania, drogowe, stopu

Do sterowania diodami dużej mocy zastosowaliśmy układ AP8801 500mA LED STEP-DOWN CONVERTER oprócz wspólnego sterowania dla każdego typu świateł każdy zestaw świateł zawierać będzie zestaw 2 pasków led (pojedynczy pasek na lewą i prawą stronę) po 9 diod, na każdy jeden pasek przypadać będzie jeden układ AP8801 (ze względu na jego wewnętrzne ograniczenia). W sumie samych układów step down będzie 6 każdy będzie odpowiedzialny za jedno światło(mijania lewe, mijania prawe, drogowe lewe, drogowe prawe, stop/mijania tylne lewy, stop/mijania tylne prawy), umożliwi to proste sterowanie i wykrywanie awarii oświetlenia.

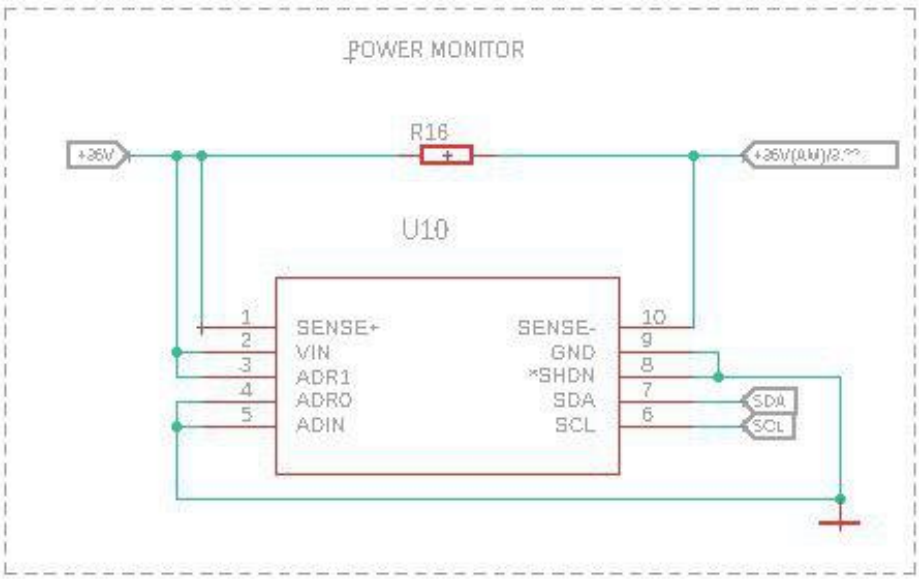


Rysunek 2: Paski 9 diod świateł mijania/drogowych/stop



Rysunek 3: Podział Power LED draiverów na poszczególne segmenty

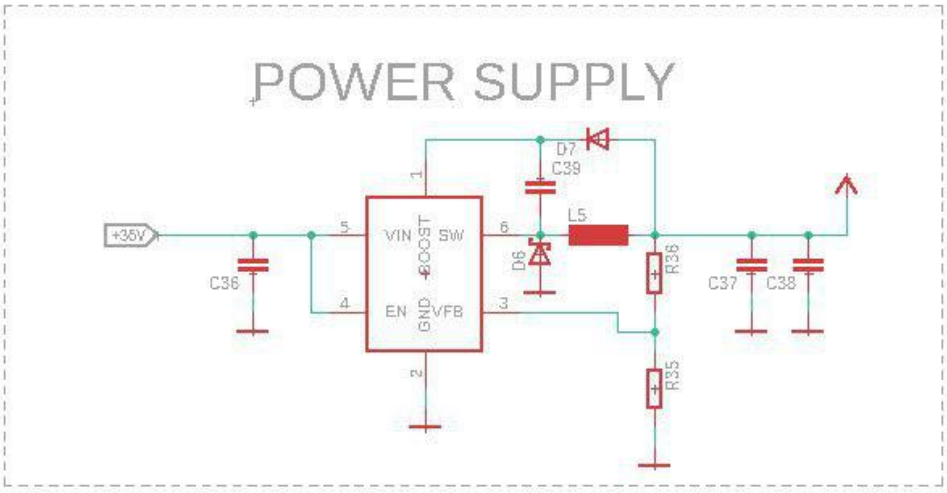
Awarie świateł wykrywamy aktywnie przy uruchomieniu pojazdu mierzony jest wówczas prąd pobierany na cały układ i porównywany z wartością znamionową. Ze względu na użycie jednego układu AP8801 na jedno światło w momencie przepalenia jednej diody prąd pobierany spadnie o jedną szóstą.



Rysunek 4: Wysoko napięciowy monitor prądu i napięcia z magiistralą I2C

1.3 Zasilanie

Zgodnie z założeniami cały projekt działa na zasilaniu +36 V DC. Ze względu na wykorzystanie mikrokontrolera STM 32, oraz programowalnych diod led, stosujemy 2 układy obniżania napięcia z 36V na 3,3V (do mikrokontrolera) oraz 12V (do programowalnych diod).

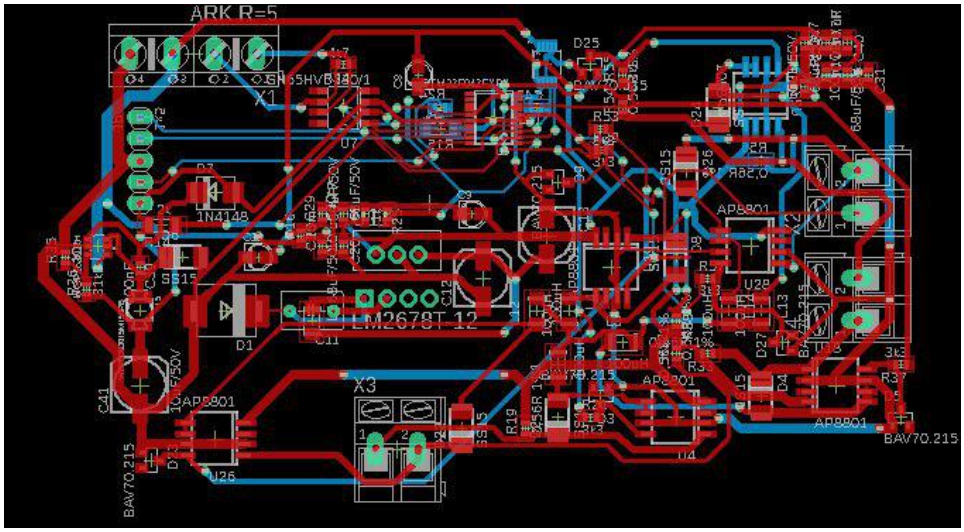


Rysunek 5: Stepdown na 3V3 dla mikrokontrolera

1.4 BRD

Schemat brd został zaprojektowany zachowując odpowiednie szerokości ścieżek proporcjonalnie do płynących w nich sygnałów. Kondensatory odsprzęgające znajdują się możliwie najbliżej układów scalonych. Uniknięto zamknięcia pętli masy.

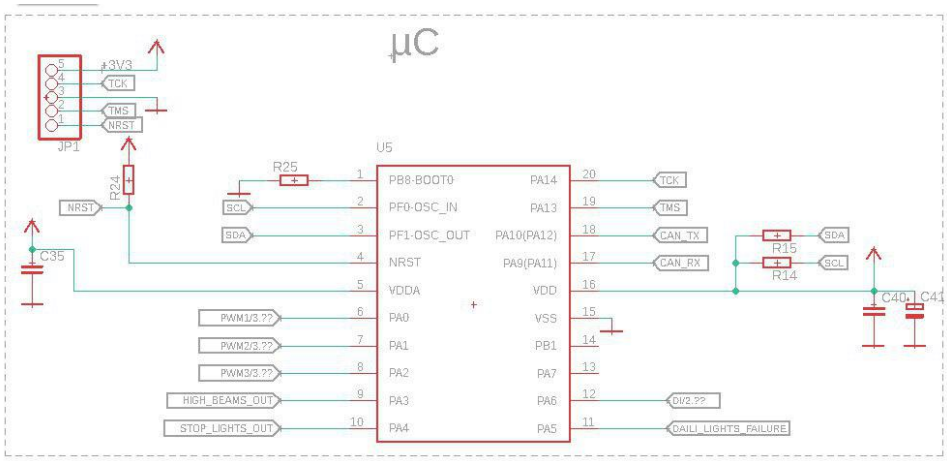
Poprzez duże nagromadzenie elementów na jednej płytce. Możliwie redukujemy koszty wykonania gotowych modułów . Ograniczamy zbędne miejsce, które pojawiłoby się w przypadku rozdzielenia modułów. Ponadto zwężamy koszt o zamówienie tylko jednego projektu, co przekłada się na lepsze warunki podczas masowej produkcji.



Rysunek 6: Kolorem czerwonym zaznaczone są ścieżki warstwy górnej, natomiast niebieskim warstwy dolnej

1.5 Mikrokontroler

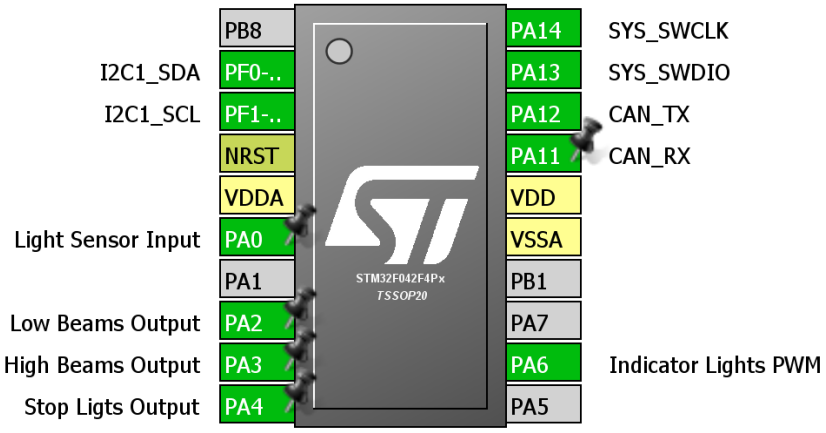
Zastosowanym mikrokontrolerem jest STM32F042F6P6. Zgodnie z założeniami projektu wybraliśmy mikrokontroler z rodziny STM32. Jest to najtańszy mikrokontroler z rodziny STM32 obsługując magristralę CAN, a jednocześnie w zupełności zaspokajający potrzeby projektu.



Rysunek 7: Schemat STM32F042F6P6

2 Oprogramowanie

2.1 Konfiguracja wyprowadzeń mikrokontrolera



Rysunek 8: Wyprowadzenia mikrokontrolera

Zastosowanie innych rozwiązań niż poprzedni twórcy projektu [1] pozwoliło na wykorzystanie mniejszej ilości wyprowadzeń mikrokontrolera. W połączeniu z konfiguracją *Debug Serial Wire* na pinach *PA14* oraz *PA13* mamy możliwość przeprogramowania mikrokontrolera w przyszłości, co może pozwolić na dodanie dodatkowych funkcji i wykorzystanie niezagospodarowanych wyprowadzeń.

Pin Number TSSOP20	Pin Name (function after reset)	Pin Type	Alternate Function(s)	Label
2	PF0-OSC_IN	I/O	I2C1_SDA	
3	PF1-OSC_OUT	I/O	I2C1_SCL	
4	NRST	Reset		
5	VDDA	Power		
6	PA0 *	I/O	GPIO_Input	Light Sensor Input
8	PA2 *	I/O	GPIO_Output	Low Beams Output
9	PA3 *	I/O	GPIO_Output	High Beams Output
10	PA4 *	I/O	GPIO_Output	Stop Ligts Output
12	PA6	I/O	TIM3_CH1	Indicator Lights PWM
15	VSSA	Power		
16	VDD	Power		
17	PA11	I/O	CAN_RX	
18	PA12	I/O	CAN_TX	
19	PA13	I/O	SYS_SWDIO	
20	PA14	I/O	SYS_SWCLK	

Rysunek 9: Raport konfiguracja wyprowadzeń

2.1.1 I2C

Wyprowadzenia *PF0-OSC_IN* oraz *PF1-OSC_OUT* zostały wykorzystane do skonfigurowania magistrali I^2C. Mikrokontroler komunikuje się tą magistralą z monitorem prądu *LTC4151*, który służy do wykrywania awarii układu [3].

```
/* I2C1 init function */
static void MX_I2C1_Init(void)
{

  hi2c1.Instance = I2C1;
  hi2c1.Init.Timing = 0x2000090E;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Configure Analogue filter
    */
  if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Configure Digital filter
    */
  if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

Rysunek 10: Inicjalizacja komunikacji po magistrali I^2C rzy użyciu HAL

Konfiguracja I^2C uniemożliwiła wykorzystanie zewnętrznego oscylatora kwarcowego [2], przez co do taktowania procesora wykorzystujemy wbudowany oscylator 48MHz, który dla naszych zastosowań jest bardziej niż wystarczający.

2.1.2 Piny wejścia/wyjścia

Do sterowania oświetlenie wykorzystane są wyprowadzenia *PA2, PA3, PA4* oraz *PA6*. Pin *PA0* służy do monitorowania stanu czujnika światła.

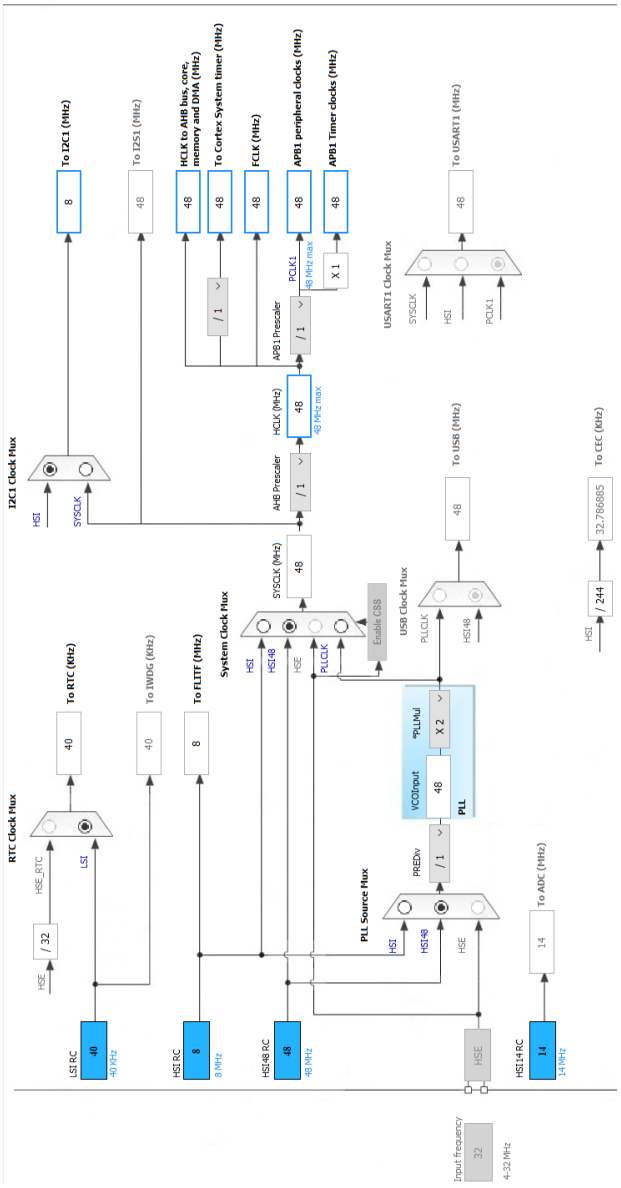
```
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, Indicator_Lights_PWM_Pin|Daily_Lights_Output_Pin|High_Beams_Output_Pin|Stop_Ligts_Output_Pin, GPIO_PIN_RESET);

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

  /*Configure GPIO pins : Indicator_Lights_PWM_Pin Daily_Lights_Output_Pin High_Beams_Output_Pin Stop_Ligts_Output_Pin */
  GPIO_InitStruct.Pin = Indicator_Lights_PWM_Pin|Daily_Lights_Output_Pin|High_Beams_Output_Pin|Stop_Ligts_Output_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}
```

Rysunek 11: Konfiguracja pinów wejścia/wyjścia

2.1.3 CAN

Zgodnie z wymaganiami specyfikacji [2], do komunikacji po magistrali CAN wykorzystane zostały wyprowadzenia *PA11* oraz *PA12*. Szczegóły konfiguracji zostaną opisane w dalszej części raportu.

2.2 Timery



Rysunek 12: Konfiguracja zegarów mikroprocesora w IDE STM32CubeMX

W projekcie wykorzystywane są trzy wewnętrzne oscylatory:

- *HSI RC*, służy do taktowania magistrali I^2C z częstotliwością 8MHz
- *HSI48 RC*, taktuje procesor z częstotliwością 48MHz
- *LSI RC* wykorzystujemy do konfiguracji zegara czasu rzeczywistego.

Ponadto w projektcie używane są dwa liczniki: *Timer3* oraz *RTC*. *Timer3* służy do generowania sygnału PWM sterującego programowalną listwe LED. Wszystkie zegary, oprócz zegara RTC zostały skonfigurowane z pomocą powłoki HAL.

```
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;
  RCC_PeriphCLKInitTypeDef PeriphClkInit;

    /**Initializes the CPU, AHB and APB busses clocks
    */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI|RCC_OSCILLATORTYPE_HSI48;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
  RCC_OscInitStruct.HSICalibrationValue = 16;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Initializes the CPU, AHB and APB busses clocks
    */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI48;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_I2C1;
  PeriphClkInit.I2c1ClockSelection = RCC_I2C1CLKSOURCE_HSI;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Configure the Systick interrupt time
    */
  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

    /**Configure the Systick
    */
  HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

  /* SysTick_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
}
```

Rysunek 13: Funkcja konfigurująca główne zegary procesora

2.2.1 Real Time Clock

Do obsługi kierunkowskazów oraz świateł awaryjnych konieczne było skonfigurowanie zegara czasu rzeczywistego (RTC). Wykorzystuje on wewnętrzny

oscylator niskoczęstotliowściowy (LSI RC) o częstotliwości 40kHz. Odczytywanie czasu rzeczywistego od uruchomienia modułu zostało zaprogramowane w funkcji void RTC (*p_buf). Przez wywołanie odpowiedniej funkcji jesteśmy wstanie odczytać czas od uruchomienia w postaci godzin, minut, sekund i milisekund.

```
//Real Time Clock
void RTC(char *p_buf) {

    uint32_t TimeVar;
    TimeVar = RTC_GetCounter();
    TimeVar = TimeVar % 40000;

    s_TimeStructVar.HourHigh = (uint8_t)(TimeVar / 3600) / 10;
    s_TimeStructVar.HourLow = (uint8_t)(TimeVar / 3600) % 10;
    s_TimeStructVar.MinHigh = (uint8_t)((TimeVar % 3600) / 60) / 10;
    s_TimeStructVar.MinLow = (uint8_t)((TimeVar % 3600) / 60) % 10;
    s_TimeStructVar.SecHigh = (uint8_t)((TimeVar % 3600) % 60) / 10;
    s_TimeStructVar.SecLow = (uint8_t)((TimeVar % 3600) % 60) % 10;
    s_TimeStructVar.MilliesHigh = (uint8_t)((TimeVar % 3600) / 60) / 1000;
    s_TimeStructVar.MilliesLow = (uint8_t)((TimeVar % 3600) / 60) % 1000;
}
```

Rysunek 14: Funkcja konfigurująca zegar czasu rzeczywistego

W momencie włączenia kierunkowskazu lub świateł awaryjnych generowany jest sygnał PWM włączający odpowiednie diody oraz zapisywana jest do zmiennej wartość z zegara czasu rzeczywistego. Przy następnym cyklu procesora zapisana wartość porównywana jest z nową wartością RTC i następuje sprawdzenie, czy różnica czasu jest większa lub równa wartości stałej Indicator_interval.

```
    if (i >= 2) {
     unsigned long currentMillis = s_TimeStructVar.millis();
      if (currentMillis - previousTime >= Indicator_interval) {
        previousTime = currentMillis;
}
```

Stała Indicator_Interval została zadekalrowana jako const static unsigned int, w celu obptymalizacji zużycia pamięci mikroprocesora, a jego wartość ustaliliśmy jako 500.

```
/* Private variables ---------------------------------

//Miganie kierunkowskazów
const static unsigned int Indicator_interval = 500;
unsigned long previousTime = 0;
uint8_t Indicator = off;
```

Oznacza to, że po upływie 500 milisekund następuje sekwencja włączania lub wyłączania odpowiednich diod. Daje nam to pulsacje o okresie 1 sekundy.

2.3 CAN

Podstawowym zadaniem przy pisaniu opragramowania było konfiguracja komunikacje po magistrali CAN. Pin *PA12* służy do wysyłania informacji po magistrali, natomiast *PA13* do odbierania.

```
/* CAN init function */
static void MX_CAN_Init(void)
{

  hcan.Instance = CAN;
  hcan.Init.Prescaler = 16;
  hcan.Init.Mode = CAN_MODE_NORMAL;
  hcan.Init.SJW = CAN_SJW_1TQ;
  hcan.Init.BS1 = CAN_BS1_3TQ;
  hcan.Init.BS2 = CAN_BS2_5TQ;
  hcan.Init.TTCM = DISABLE;
  hcan.Init.ABOM = DISABLE;
  hcan.Init.AWUM = DISABLE;
  hcan.Init.NART = DISABLE;
  hcan.Init.RFLM = DISABLE;
  hcan.Init.TXFP = DISABLE;
  if (HAL_CAN_Init(&hcan) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

Rysunek 15: Inicjalizacja CAN

Zgodnie z dokumentacją [2], skonfigurowane zostały trzy skrzynki odbiorcze, w których przechowywane są wiadomości, aż do czasu wykonania odpowiednich funkcji.

```
void CAN1_Tx(uint8_t data0, data1, data2) {

    CAN1->sTxMailBox[0].TDLR = data0;
    CAN1->sTxMailBox[0].TIR = 0;

    CAN1->sTxMailBox[1].TDLR = data1;
    CAN1->sTxMailBox[1].TIR = 0;

    CAN1->sTxMailBox[2].TDLR = data2;
    CAN1->sTxMailBox[2].TIR = 0;
}
```

Filtr magistrali został skonfigurowany na priorytet identyfikatora wiadomości. Oznacza to, że informacja z niższym numerem ID zostanie obsłużona jako pierwsza.

```
// Configure the CAN Filter
sFilterConfig.FilterNumber = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = 0;
sFilterConfig.FilterActivation = ENABLE;
sFilterConfig.BankNumber = 14;
```

W projekcie założyliśmy siedem modułów, od których może zostać odebrany sygnał: Przełącznik świateł długich, świateł drogowych, świateł awaryjnych, pedał stopu oraz obu kierunkowskazów.

```
/* USER CODE BEGIN 4 */
void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef *CanHandle)
{

    //Światła dzienne
    if ((CanHandle->pRxMsg->StdId == 0x255) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        Daily_Lights(CanHandle->pRxMsg->Data[0]);
    }

    //Światła mijania
    if ((CanHandle->pRxMsg->StdId == 0x150) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        Low_Beam(CanHandle->pRxMsg->Data[0]);
    }

    //Światła długie
    if ((CanHandle->pRxMsg->StdId == 0x100) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        High_Beam(CanHandle->pRxMsg->Data[0]);
        ubKeyNumber = CanHandle->pRxMsg->Data[0];
    }

    //Kierunkowskaz lewy
    if ((CanHandle->pRxMsg->StdId == 0x50) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        Indicator_Left(CanHandle->pRxMsg->Data[0]);
        ubKeyNumber = CanHandle->pRxMsg->Data[0];
    }

    //Kierunkowskaz prawy
    if ((CanHandle->pRxMsg->StdId == 0x51) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        Indicator_Right(CanHandle->pRxMsg->Data[0]);
        ubKeyNumber = CanHandle->pRxMsg->Data[0];
    }

    //Awaryjne
    if ((CanHandle->pRxMsg->StdId == 0x10) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        Awaryjne(CanHandle->pRxMsg->Data[0]);
        ubKeyNumber = CanHandle->pRxMsg->Data[0];
    }

    //Stop
    if ((CanHandle->pRxMsg->StdId == 0x00) && (CanHandle->pRxMsg->IDE == CAN_ID_STD) && (CanHandle->pRxMsg->DLC == 2))
    {
        Stop_Lights(CanHandle->pRxMsg->Data[0]);
        ubKeyNumber = CanHandle->pRxMsg->Data[0];
    }

    /* Receive */
    if (HAL_CAN_Receive_IT(CanHandle, CAN_FIFO0) != HAL_OK)
    {
        /* Reception Error */
        Error_Handler();
    }
```

Rysunek 16: Obsługa odbioru wiadomości od poszczególnych modułów

Przyjęliśmy, że najniższe ID (0x00) będzie posiadał komunikat o światłach stopu. Oznacza to, że informacja o wciśnięciu lub zwolnieniu hamulca będzie zawsze obsługiwana jako pierwsza. Następnie w kolejności: informacja o światłach arawyjnych, kierunkowskazach, światłach długich, światłach mijania oraz światłach dziennych.

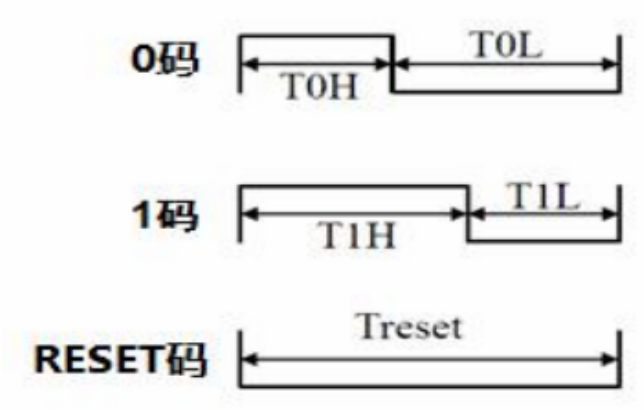
Jednakże z powodu, że instrukcje procesora są proste i jedynie programowalna listwa RGB wymaga od nas modulacji bardziej skomplikowanego sygnału PWM, nie spodziewamy się sytuacji przepełnienia skrzynki odbiorczej. Nawet sytuacja w której otrzymalibyśmy trzy informacje na raz nie powinna

wywołać widocznego opóźnienia reakcji.

2.4 Programowalna listwa LED

Wykorzystanie diod programowalnych typu WS2811 pozwoliło wyeliminować LED drivery zastosowane przez poprzednią grupę, oraz sterowanie czterena kierunkowskazami oraz światłami dziennymi jednym sygnałem PWM.

Sygnał dla WS2811 nie jest sygnałem typowym. Wartość niska kodowana jest przez ustawienie stanu wysokiego przez 220ns do 380ns [4], a następnie stanu niskiego trwającego od 580ns do 1,6us. Natomiast stan wysoki kodujemy ustawiając stan wysoki na 580ns do 1,6us, a następnie stan niski od 220ns do 420ns. Ustawienie stanu niskiego przez minimum 280us stanu niskiego oznacza reset.



Rysunek 17: Kodowanie bitu [4]

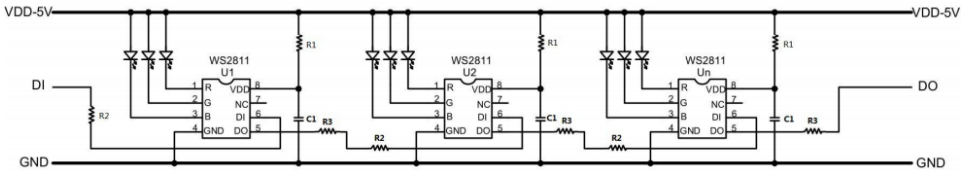
Pojedyncze słowo składa się 24 bitów. W słowie przypada 8 bitów na pojedynczą diodę: Czerwoną, Niebieską i zieloną. 8 bitów pozwala nam uzyskać 255 stopni regulacji intensywności świecenia dla każdej z 3 składowych diod.

Composition of 24bit Data

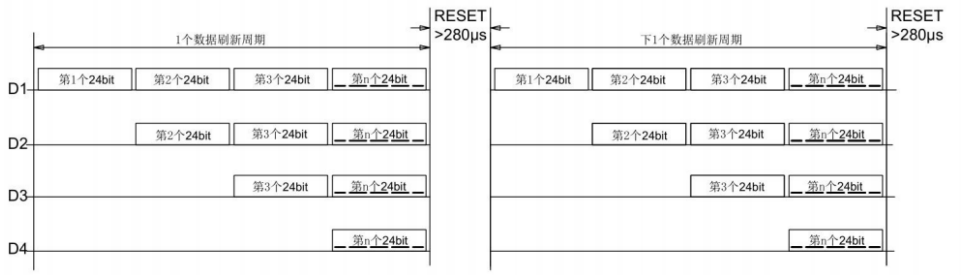
R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

Rysunek 18: Kompozycja jednego słowa [4]

Sterowniki diod połączone są ze sobą kaskadowo. Każdy sterownik interpretuje najmłodsze słowo sygnału, odcine je i przekazuje resztę sygnału dalej, tak jak pokazano to na rysunkach.



Rysunek 19: Kaskadowe połączenie sterowników [4]



Rysunek 20: Zasada przepływu sygnałów [4]

Oznacza to, że w pojedynczej sekwencji musimy nadać 180 bajtów. Licząc, że na nadanie jednego bitu potrzeba około 1us, to pojedyncza sekwencja będzie trwać 1440us + 280us bitu stopu. Rozważmy przypadek w którym obsługujemy kierunkowskaz i rozpoczynamy wysyłanie 60 słów do naszego paska ledowego. Jeśli w tym samym momencie użytkownik wyłączy kierunkowskaz, to dla ludzkiego oka reakcja będzie natychmiasotwa, mimo

konieczności nadania wszystkich słów.

Tak jak było wspomniane, sygnał modulowany dla listwy ledowej jest dość nietypowy, przez co obsługa jje mogłaby przysporzyć wielu problemów. Na szczęście w powłoce HAL dostępna jest biblioteka znacznie ułatwiająca konfiguracje.

```
static const struct init_entry_ init_table[] =
{
    { &FLASH->ACR, FLASH_ACR_PRFTBE | 1 },
{ &RCC->CFGR, RCC_CFGR_PLLMULL10 | RCC_CFGR_SW_PLL },

{ &RCC->APB2ENR, RCC_APB2ENR_SPI1EN },
{ &RCC->AHBENR, RCC_AHBENR_GPIOAEN | RCC_AHBENR_SRAMEN },

{ &GPIOA->OSPEEDR, BF2(6, GPIO_OSPEEDR_MED) }, // MOSI - medium speed
{ &GPIOA->MODER, GPIOA_MODER_SWD | BF2(7, GPIO_MODER_AF) }, // MOSI as AF
                                                            // SPI setup
{ (__IO32p)&SPI1->CR2, SPI_CR2_DSIZE(12) | SPI_CR2_TXEIE },
{ (__IO32p)&SPI1->CR1, SPI_CR1_SSM | SPI_CR1_SSI | SPI_CR1_SPE | SPI_CR1_BRDIV16 | SPI_CR1_MSTR }, //
enable
//SysTick setup
{ &SCB->SHP[1], 0x80c00000 }, // PendSV lower priority, Systick higher
{ &SysTick->LOAD, SYSCLK_FREQ / SYSTICK_FREQ - 1 },
{ &SysTick->VAL, 0 },
{ &SysTick->CTRL, SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk },
// interrupts and sleep
{ &NVIC->ISER[0], 1 << SPI1_IRQn }, // enable interrupts
{ &SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk }, // sleep while not in handler
{ 0, 0 }
};
```

Dzięki temu byliśmy wstanie zaprogramować płynnięcie kierunkowskazu, wzrosując się na efekcie uzyskanym w samochodach marki Audi.

```
void SysTick_Handler(void)
{
    static uint8_t change_timer = CHANGE_PERIOD;
    static uint8_t newpix_timer = GEN_PERIOD;
    static uint8_t nxtcol = 0;
    if (--change_timer == 0)
    {
        uint32_t i;
        change_timer = CHANGE_PERIOD;
        for (i = NLEDS - 1; i > 0; i--)
            wsdata[i] = wsdata[i - 1];
        if (--newpix_timer == 0)
        {
            newpix_timer = GEN_PERIOD;
            wsdata[0] = colors[nxtcol];
            if (++nxtcol == sizeof(colors) / sizeof(struct wspix_))
                nxtcol = 0;
        }
        else
        {
            wsdata[0].green /= 2;
            wsdata[0].red /= 2;
            wsdata[0].blue /= 2;
        }
        WS2812_start(wsdata, NLEDS);
    }
}
```

2.5 Czujnik światła

Po uruchomieniu pojazdu, w zależności od sygnału z czujnika natężenia światła następuje włączenie świateł dziennych lub mijania. Oświetlenie jest przełączane w zależności od obecnych odczytów czujnika do czasu aż operator pojazdu nie wykona następujących operacji: Ręczne wyłączenie świateł dziennych lub mijania, ręczne włączenie świateł długich, dziennych lub mijania. Po wystąpieniu takiego zdarzenia kontroler nie będzie reagował na sygnały z czujnika natężenia światła, aż do następnego uruchomienia pojazdu.

```
while (1)
{

  if(Forced) {

  } else {
     if (Light_Sensor) {
         Daily_Lights();
     } else {
         Low_Beam();
     }
  }
```

2.6 Wykrywanie awarii

Wykrywanie awarii polega na włączeniu na dwie sekundy wszystkich świateł i pomiar łącznego poboru prądu przez monitor prądu LTC4151. Wartość ta zostaje wysłana magistralą I^2C do mikrokontrolera, gdzie zostaje porównana ze znamionowym poborem prądu. Jeśli wartości różnią się od siebie o 20%, mikrokontroler wysyła po magistrali CAN (w domyśle do głównego komputera pojazdu) informacje o błędzie.

```
bool error_checker () {
    bool error = false;
    test();
    uint8_t value;

    I2C1.LTC4151.read() = value;

    if (value >= 1.2 * current || value <= 0.8 * current) {
        error = true;
        CanHandle.pTxMsg->StdId = error;
    }
    return error;
}
```

Bibliografia

[1] Sławomir Smyczyński Arkadiusz Nawrot. *Projekt oświetlenia w pojeździe*. 2018.

[2] STM32. *STM32F042x4 STM32F042x6 Data Sheet*.

[3] Linear Technology. *LTC4151 datasheet*.

[4] Worldsemi. *WS2811 - Signal line 256 Gray level 3 channel Constant current LED driver IC datasheet*.