

Cel Projektu:

Obsługiwać dwa serwomechanizmy wykorzystując dwa kanały timera działające w trybie PWM out z DMA, aby zapewnić najwyższą możliwą rozdzielczość sygnału wyjściowego.

Obsługiwać czujnik ultradźwiękowy z wyzwaniem w pętli głównej i odbiorem za pomocą timera pracującego w trybie input capture, pobierać dane w zadanym odstępie czasu w milisekundach i zapisywać 500 pozycji w buforze kołowym, umożliwiać przeglądanie danych bieżących i archiwalnych.

Protokół ramki HDLC oraz znaki ucieczki.

nazwa	Liczba bajtów	opis	Sposób zapisu
Flaga początkowa	1	Początek ramki	0x7E
Adres nadawcy	1	Identyfikuje nadawcę	Uint_8 (bajt)
Adres odbiorcy	1	Identyfikuje odbiorcę	Uint_8 (bajt)
Długość danych	2	Określa długość danych, czyli liczbę bajtów, które będą przesyłane w sekcji danych. Zawiera rozmiar danych w ramce.	Uint_16 MSB
Dane	3500	Zawiera właściwe dane do transmisji, które są przesyłane między urządzeniami. To może być np. komenda, wartości sensorów, itp	0-3499 bajty
CRC-8	1	suma kontrolna CRC-8 obliczona na podstawie danych, zapewniająca integralność transmisji	Uint_8 (bajt)
Flaga końcowa	1	Koniec ramki	0x7E

ZNAKI UCIECZKI

Ramka zaczyna się i kończy znakiem 0x7E (w binarnym 01111110), który wskazuje, gdzie ramka się rozpoczyna i kończy.

Problem z 0x7E w danych: Jeśli 0x7E pojawi się wewnątrz danych, odbiornik mógłby błędnie uznać, że to koniec ramki.

Rozwiązanie

znak ucieczki 0x7D

Jeśli w danych wystąpi 0x7E, to wysyłamy zamiast niego:

najpierw 0x7D (znak ucieczki),

potem 0x5E (0x7E z odwróconym 5. bitem).

Jeśli pojawia się 0x7D(sam znak ucieczki), wysyłamy:

0x7D 0x5D (zamaskowany 0x7D).

OBLICZANIE CRC

Wielomian dla CRC-8

CRC-8 (ATM): $x^8 + x^2 + x + 1$ (wielomian 0x07) jako “wielomian sprawdzający”
zaczynamy od 0x00

Jako dane wysłane, traktuje dane jako wielomian (ciąg bitów), przed wysłaniem
liczone jest CRC-8 w ten sposób :

Przykład danych wejściowych 0x25 = 0010 0101 jest brane jako:

$$0 \cdot x^7 + 0 \cdot x^6 + 1 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0.$$

Urządzenie wysyłające oblicza CRC poprzez dodanie do danych odpowiedniej liczby zer. Dane są następnie przetwarzane przy użyciu matematyki binarnej przed wysłaniem ich kanałem komunikacyjnym. Ta metoda to implementacja „dodanych zer”

Najpierw do pełnego pakietu danych dodawane są zera. Dopasuj wielomian do pierwszej cyfry 1 w danych. Dane są poddawane XOR z wielomianem. Rezultatem są nowe dane. Ponownie wyrównaj wielomian z cyfrą znajdującą się najbardziej na lewo w danych. Proces trwa do momentu, aż dane będą mniejsze od wielomianu. To CRC jest dodawane do strumienia danych przed transmisją.

Zasady podczas liczenia:

1. Jeśli bit po lewej to 1, wykonuj XOR z wielomianem
2. Jeśli bit po lewej to 0, przesun się dalej
3. Kontynuuj aż przetworzysz wszystkie bity
4. Ostatnie 8 bitów to twój CRC

OGÓLNE ZASADY KOMEND

Rozpoznanie i wykonanie:

- Dla każdej komendy:
 1. **Znane komendy:** Wykonaj odpowiednie działanie (np. SET1[080] (ustawienie wartości dla serva)).
 2. **Nieznane komendy:** Wyślij komunikat zwrotny: **ERROR**
 3. **“Uszkodzone” komendy:** Jeżeli komenda jest źle napisana (np. Ua[1000[), bądź zadany zakres jest niepoprawny traktowana jest również **jak w punkcie 2.**
 4. W przypadku wystąpienia kilku komend w jednej ramce, konieczne jest odzielenie ich średnikiem. **Nie rozdzielone komendy będą traktowane jak w punkcie 2. Ostatnia komenda nie musi kończyć się średnikiem**

Wartości są wysyłane są jako surowe bajty

Wartości podawane dla komend z parametrami w kwadratowych nawiasach są decymalne.

KOMENDY

2.1 Komendy Sterowania Serwomechanizmami

SET1[xxx] - Sterowanie Serwem 1

- **Opis:** Ustawia kąt dla serwomechanizmu 1
- **Format:** SET1[xxx], gdzie xxx to kąt w stopniach
- **Długość:** 9 bajtów (stała)
- **Zakres wartości:** 000-180 (z wiodącymi zerami)
- **Odpowiedzi:**
 - Sukces: "S1 SET"
 - Błąd: "S1 INVALID FORMAT"
- **Przykład:** SET1[090] - ustawienie serwa 1 na 90 stopni

SET2[xxx] - Sterowanie Serwem 2

- **Opis:** Ustawia kąt dla serwomechanizmu 2
- **Format:** SET2[xxx], gdzie xxx to kąt w stopniach
- **Długość:** 9 bajtów (stała)
- **Zakres wartości:** 000-180 (z wiodącymi zerami)
- **Odpowiedzi:**
 - Sukces: "S2 SET"
 - Błąd: "S2 INVALID FORMAT"
- **Przykład:** SET2[045] - ustawienie serwa 2 na 45 stopni

ADDDMA[xxx] - Dodawanie Wartości do DMA

- **Opis:** Dodaje wartość kąta do tablicy DMA
- **Format:** ADDDMA[xxx], gdzie xxx to kąt w stopniach
- **Długość:** 11 bajtów (stała)
- **Zakres wartości:** 000-180 (z wiodącymi zerami)
- **Odpowiedzi:**
 - Sukces: "DMA ADDED"
 - Błąd: "DMA INVALID FORMAT"
- **Przykład:** ADDDMA[120] - dodanie kąta 120 stopni do tablicy DMA

SAUTO[v] - Sterowanie Automatyczne Kąty pobierane z DMA

- **Opis:** Włącza/wyłącza automatyczną sekwencję ruchów
- **Format:** SAUTO[v], gdzie v to 0 lub 1

- **Długość:** 8 bajtów (stała)
- **Odpowiedzi:**
 - Włączenie: "SAUTO START"
 - Wyłączenie: "SAUTO STOP"
 - Błąd: "SAUTO INVALID FORMAT"
- **Przykład:** SAUTO[1] - włączenie trybu automatycznego

2.2 Komendy Czujnika Ultradźwiękowego

UA[XXXX] - Ustawienie Interwału

- **Opis:** Konfiguruje interwał automatycznego odczytu
- **Format:** UA[XXXX], gdzie XXXX to czas w milisekundach
- **Długość:** 8 bajtów (stała)
- **Zakres wartości:** 0010-1000 (z wiodącymi zerami)
- **Odpowiedzi:**
 - Sukces: "UA SET"
 - Błąd: "UA INVALID FORMAT"
- **Przykład:** UA[0500] - ustawienie interwału na 500ms

UA? - Odczyt Interwału

- **Opis:** Zwraca aktualnie ustawiony interwał odczytu
- **Format:** UA?
- **Długość:** 3 bajty (stała)
- **Odpowiedzi:**
 - Sukces: "UA <wartość>" (np. "UA 500")
 - Błąd: "ERROR"

2.3 Komendy Bufora Danych

BUF - Odczyt Ostatnich Pomiarów

- **Opis:** Pobiera 20 ostatnich pomiarów z bufora
- **Format:** BUF
- **Długość:** 3 bajty (stała)
- **Odpowiedzi:**
 - Sukces: Lista pomiarów (np. "12.00 15.30 18.00 ...")
 - Brak danych: "NODATA"

BUFALL - Odczyt Całego Bufora

- **Opis:** Pobiera wszystkie pomiary z bufora (do 500)

- **Format:** BUFALL
- **Długość:** 6 bajtów (stała)
- **Odpowiedzi:**
 - Sukces: Lista pomiarów (np. "12.00 15.30 18.00 ...")
 - Brak danych: "NODATA"

BUFN[start,end] - Odczyt Zakresu

- **Opis:** Pobiera pomiary z określonego zakresu
- **Format:** BUFN[xxx,xxx], gdzie xxx to indeksy
- **Długość:** 13 bajtów (stała)
- **Zakres wartości:** 000-500 dla obu parametrów
- **Warunki:** start \leq end
- **Odpowiedzi:**
 - Sukces: Lista pomiarów z zakresu
 - Błędy:
 - "BUFN INVALID FORMAT" - nieprawidłowy format
 - "BUFN INVALID START" - nieprawidłowa wartość początkowa
 - "BUFN INVALID END" - nieprawidłowa wartość końcowa
 - "BUFN INVALID RANGE" - nieprawidłowy zakres
- **Przykład:** BUFN[000,020] - pobranie pierwszych 21 pomiarów

Nieznana komenda zwróci ERROR (długość 5 bajtów)

FUNKCJA NA PARSOWANIE WIELU KOMEND - OMÓWIENIE

```
void processMultipleCommands(uint8_t *cmdData, uint16_t totalLength) {
    uint16_t pos = 0;
    while (pos < totalLength) {
        // Pomijam bajty separatora (0x3B to ';' w ASCII)
        while (pos < totalLength && cmdData[pos] == 0x3B) {
            pos++;
        }
        if (pos >= totalLength)
            break;

        uint16_t start = pos;
        // Szuka separatora lub końca bufora
        while (pos < totalLength && cmdData[pos] != 0x3B) {
            pos++;
        }
        uint16_t cmdLength = pos - start;
        if (cmdLength > 0) {
            processCommand(&cmdData[start], cmdLength); // Przekazuje surowe bajty
        }
    }
}
```

Funkcja przyjmuje dwa parametry:

- cmdData: wskaźnik na bufor zawierający ciąg komend
- totalLength: całkowita długość danych w buforze

Zmienna pos służy jako wskaźnik pozycji w buforze. Na początku jest ustawiona na 0 i będzie przesuwana w miarę analizy danych.

Główna pętla while działa dopóki nie przeanalizujemy wszystkich danych (pos < totalLength). W każdej iteracji:

Pierwsza zagnieżdżona pętla while pomija wszystkie kolejne średniki, zabezpiecza to przed wpisanymi głupotami np ;;;SET1[120].

Następnie zapisujemy początkową pozycję komendy w zmiennej start:

Druga zagnieżdżona pętla while szuka końca aktualnej komendy kolejnego średnika lub końca bufora

Jeśli komenda ma niezerową długość, jest przekazywana do funkcji processCommand do przetworzenia.

RAMKA JEŻELI NATRAFI NA NP. (urposzczone dla lepszego rozumowania)

0x7e aaa;SET1[120];kolejnysmieć 0x7e

ramka zostanie przyjęta jednak zostanie zwrócone:

ERROR;S1 SET;ERROR

poprawne komendy (o ile parametry się zgadzają) zostaną wykonane.

KONWERTOWANIE NA DEC

Za przetworzenie surowych bajtów na decymalne wartości odpowiada funkcja:

```
value = value * 10 + (cmdData[i] - '0');
```

cmdData[i] - '0' konwertuje znak cyfry do wartości liczbowej:

np:

$$'1' - '0' \rightarrow 49 - 48 = 1$$

$$'2' - '0' \rightarrow 50 - 48 = 2$$

$$'3' - '0' \rightarrow 51 - 48 = 3$$

$value * 10 + (cmdData[i] - '0')$ buduje liczbę całkowitą poprzez przesunięcie wartości w miejscu dziesiętnym:

Jeśli value zaczyna się od 0 i cmdData zawiera "123", operacje będą przebiegać tak:

$$value = 0 * 10 + (1) = 1$$

$$value = 1 * 10 + (2) = 12$$

$$value = 12 * 10 + (3) = 123$$

Ostatecznie value będzie zawierało liczbę 123.

FUNKCJA NA ODRZUCANIE RAMKI – DANE SAME ERROR

```
// funkcja sprawdzająca, czy bufor odpowiedzi zawiera wyłącznie komunikaty "ERROR "
uint8_t allResponsesAreError(void) {
    if (bufIndex == 0) return 0; // Bufor pusty - nie traktujemy jako same błędy

    // długość nie jest wielokrotnością 6, nie są same bloki "ERROR "
    if (bufIndex % 6 != 0) return 0;

    uint16_t numBlocks = bufIndex / 6;

    for (uint16_t i = 0; i < numBlocks; i++) {
        //ERROR plus spacja
        if (memcmp(&tempbufanswer[i * 6], "ERROR ", 6) != 0)
            return 0; // Znaleziono blok inny niż "ERROR "
    }

    return 1; // Wszystkie bloki to "ERROR "
}
```

PĘTLA GŁÓWNA

Pętla główna sprawdza cały czas czy są dane do odebrania oraz pobiera dane z czujnika co zadany interwał czasowy.

```
while (1)
{
    //ServoWiper(); // na testowanie serva

    HDLC_ProcessInput();

    //czujnik
    // żeby działał to trzeba zrobić trigger na high przez minimum 10 us
    GetUltrasonicDistance();

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

```

int8_t HDLC_ProcessInput(void)
{
    while (USART_kbhit())
    {
        uint8_t rxByte = (uint8_t)USART_getchar();

        // sprawdzenie czy bufor jest pełny
        if (hdlcInPos >= HDLC_MAX_FRAME_SIZE)
        {
            hdlcRxState = WAITING_FOR_FLAG;
            hdlcInPos = 0;
            continue;
        }

        switch (hdlcRxState)
        {
            case WAITING_FOR_FLAG:
                if (rxByte == HDLC_FLAG)
                {
                    // Flaga oznacza początek ramki
                    hdlcInPos = 0;
                    hdlcRxState = READING_FRAME;
                }
                break;

            case READING_FRAME:
                if (rxByte == HDLC_FLAG)
                {
                    // Otrzymano flagę - traktujemy ją jako koniec bieżącej ramki
                    if (hdlcInPos > 0)
                    {
                        HDLC_ParseFrame(hdlcInBuf, hdlcInPos);
                    }
                    // Resetuj bufor, ale ustaw stan na READING_FRAME,
                    // bo ten sam bajt 0x7E ma być traktowany jako początek nowej ramki
                    hdlcInPos = 0;
                    hdlcRxState = READING_FRAME;
                }
                else if (rxByte == HDLC_ESCAPE)
                {
                    hdlcRxState = ESCAPING_BYTE;
                }
                else
                {
                    hdlcInBuf[hdlcInPos++] = rxByte;
                }
                break;

            case ESCAPING_BYTE:
                {
                    uint8_t decodedByte;
                    if (rxByte == HDLC_ESCAPE_7E)
                        decodedByte = HDLC_FLAG;
                    else if (rxByte == HDLC_ESCAPE_7D)
                        decodedByte = HDLC_ESCAPE;
                    else
                    {
                        // Błędna sekwencja escape - odrzucamy ramkę
                        hdlcRxState = WAITING_FOR_FLAG;
                        hdlcInPos = 0;
                        break;
                    }
                    hdlcInBuf[hdlcInPos++] = decodedByte;
                    //powracamy do stanu odbioru ramki
                    hdlcRxState = READING_FRAME;
                }
                break;

            default:
                hdlcRxState = WAITING_FOR_FLAG;
                hdlcInPos = 0;
                break;
        }
    }
    return 0;
}

```

Funkcja `HDLC_ProcessInput` działa w pętli – dopóki dostępne są bajty do odczytu (sprawdzone przez `USART_kbhit`), pobiera kolejny bajt z portu USART (za pomocą `USART_getchar`) i przetwarza go w zależności od bieżącego stanu odbioru (`hdlcRxState`).

Na początku każdej iteracji funkcja sprawdza, czy indeks bufora (`hdlcInPos`) nie przekroczył maksymalnej dopuszczalnej długości ramki (`HDLC_MAX_FRAME_SIZE`). Jeśli bufor jest pełny, następuje reset: `hdlcRxState` ustawiany jest na `WAITING_FOR_FLAG`, a `hdlcInPos` zostaje wyzerowany, i pętla przechodzi do następnej iteracji – w ten sposób chronimy pamięć przed nadpisaniem.

W stanie `WAITING_FOR_FLAG` funkcja oczekuje na bajt równy `HDLC_FLAG` (np. `0x7E`), który sygnalizuje początek nowej ramki. Gdy taki bajt zostanie odebrany, bufor jest resetowany (`hdlcInPos = 0`), a stan zmieniany na `READING_FRAME`, co oznacza, że od tego momentu odbierane bajty będą częścią ramki.

W stanie `READING_FRAME` każdy odebrany bajt jest dodawany do bufora. Jeśli w tym stanie pojawi się kolejny bajt równy `HDLC_FLAG`, funkcja traktuje go jako znak zakończenia bieżącej ramki. Wówczas, jeżeli w buforze znajdują się jakieś dane (`hdlcInPos > 0`), wywoływana jest funkcja `HDLC_ParseFrame`, która przetwarza zbudowaną ramkę. Następnie bufor jest resetowany, a stan ustawiany na `READING_FRAME` – dzięki temu ten sam bajt `0x7E` (lub kolejny) natychmiast inicjuje nową ramkę.

W stanie `ESCAPING_BYTE` (aktywnym, gdy odebrano bajt `HDLC_ESCAPE`) funkcja dekoduje kolejny bajt. Jeśli bajt odpowiada `HDLC_ESCAPE_7E`, zostaje przekształcony na `HDLC_FLAG`; jeśli odpowiada `HDLC_ESCAPE_7D`, na `HDLC_ESCAPE`. Jeśli bajt nie pasuje do żadnej z oczekiwanych sekwencji, uznaje się, że nastąpił błąd – stan i bufor są resetowane, a ramka odrzucana. Po poprawnym dekodowaniu, zdekodowany bajt jest dodawany do bufora, a stan wraca do `READING_FRAME`, aby kontynuować odbiór danych ramki.

PARSOWANIE RAMKI

```
void HDLC_ParseFrame(const uint8_t* frame, uint16_t length) {
    if (length < HDLC_MIN_FRAME_SIZE)
        return;

    uint8_t addrSrc = frame[0];
    uint8_t addrDst = frame[1];
    uint16_t dataLen = (frame[2] << 8) | frame[3];

    if (dataLen + 5 > length) {
        uint8_t message[] = "LEN NOT MATCH";
        HDLC_SendFrame(addrSrc, addrDst, message, sizeof(message) - 1);
        return;
    }

    const uint8_t* dataPtr = &frame[4];
    uint8_t crcRecv = frame[4 + dataLen];

    // Przygotowanie bufora do obliczenia CRC
    uint8_t tempBuf[4 + dataLen];
    tempBuf[0] = addrSrc;
    tempBuf[1] = addrDst;
    tempBuf[2] = frame[2];
    tempBuf[3] = frame[3];
    memcpy(&tempBuf[4], dataPtr, dataLen);

    // Oblicz CRC
    uint8_t crcCalc = computeCRC8(tempBuf, 4 + dataLen);
    if (crcCalc != crcRecv) {
        uint8_t message[] = "INVALID CRC";
        HDLC_SendFrame(addrSrc, addrDst, message, sizeof(message) - 1);
        return;
    }

    static uint8_t cmdBuff[MAX_DATA_LEN];
    if (dataLen >= MAX_DATA_LEN) // Zabezpieczenie przed przepełnieniem
        return;

    uint16_t cmdLen = dataLen;
    memcpy(cmdBuff, dataPtr, cmdLen);

    // Przetwarzam wiele komend oddzielonych średnikiem
    processMultipleCommands(cmdBuff, cmdLen);

    // Jeśli bufor odpowiedzi zawiera same "ERROR", ramka jest odrzucana
    if (!allResponsesAreError()) {
        HDLC_SendFrame(addrSrc, addrDst, tempbufanswer, (uint16_t)bufIndex);

        // Reset bufora
        memset(tempbufanswer, 0, sizeof(tempbufanswer));
        bufIndex = 0;
    }
}
```

BIG-EDIAN

```
uint16_t dataLen = (frame[2] << 8) | frame[3];
```

Big-endian jest tu używany, aby zachować zgodność ze standardami formatu danych, które wymagają określonej kolejności bajtów. Dzięki temu liczba dataLen jest zawsze interpretowana poprawnie, niezależnie od architektury procesora.

OGÓLNE USTAWIENIA TIMERÓW

SERVO-MECHANIZMY

1. Timer 1 – generowanie sygnału PWM do sterowania serwami

Dla serwomechanizmów generuje się sygnał 20 ms (50 Hz). Wypełnienie jest zmienne od 1 ms do 2 ms (co odpowiada zakresowi kąta około 0–180°).

wzór na obliczenie częstotliwości impulsu

$$\text{Frequency} = \text{ClockFreq} / ((\text{PSC} + 1) * (\text{ARR} + 1))$$

Aby uzyskać jak największą rozdzielczość sygnału PWM, należy minimalizować wartość preskalera przy zachowaniu wymaganego okresu (20 ms dla 50 Hz). Im niższy PSC, tym timer pracuje z większą częstotliwością, a wartość auto-reload staje się wyższa. Wyższa wartość ARR oznacza więcej dyskretnych ticków w jednym okresie, co pozwala na precyzyjniejsze ustawienie szerokości impulsu czyli dokładniejsze sterowanie kątem serwa zatem dla 16 bitowego timera zastosowałem takie ustawienia.

Prescaler = 54, Period = 65453: wartości te dobrano tak, by uzyskać częstotliwość w okolicach 50 Hz

2. DMA – obsługa sterowania serwomechanizmu (PWM) w trybie automatycznym

Kod DMA inicjalizowany jest przez `MX_DMA_Init()`, a w pliku głównym wykorzystuje się:

```
DMA_HandleTypeDef hdma_tim1_ch1;
```

```
DMA_HandleTypeDef hdma_tim1_ch2;
```

oraz funkcje:

- `PWM_DMA_Init(&huart2, &htim1)` – przygotowuje strukturę do obsługi automatycznych ruchów serwa.
- `PWM_DMA_AddValue(angle)` – dodaje kąt serwa (0–180°) do tablicy, z której DMA będzie pobierać wartości wypełnienia (tzw. `pwm_values[]`).
- `PWM_DMA_Start()` – uruchamia generowanie sygnału PWM na kanale 1 w pętli (DMA w trybie CIRCULAR).
- `PWM_DMA_Stop()` – zatrzymuje tę sekwencję.

W przerwaniu DMA (`HAL_TIM_PWM_PulseFinishedCallback`) zwiększany jest indeks aktualnej wartości PWM (sterowanie sekwencją ruchów serwa).

Obsługa serwomechanizmów – PWM i DMA

W moim projekcie DMA obsługuje tylko serwomechanizm podłączony do timera 1 i kanału 1.

Funkcje do sterowania serwem

- `__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, pulse_ticks);` – bezpośrednio ustawienie wypełnienia sygnału PWM na danym kanale (serwo 1).

- `__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, pulse_ticks);` – analogicznie dla serwo 2.

`pulse_ticks` obliczamy np. jako:

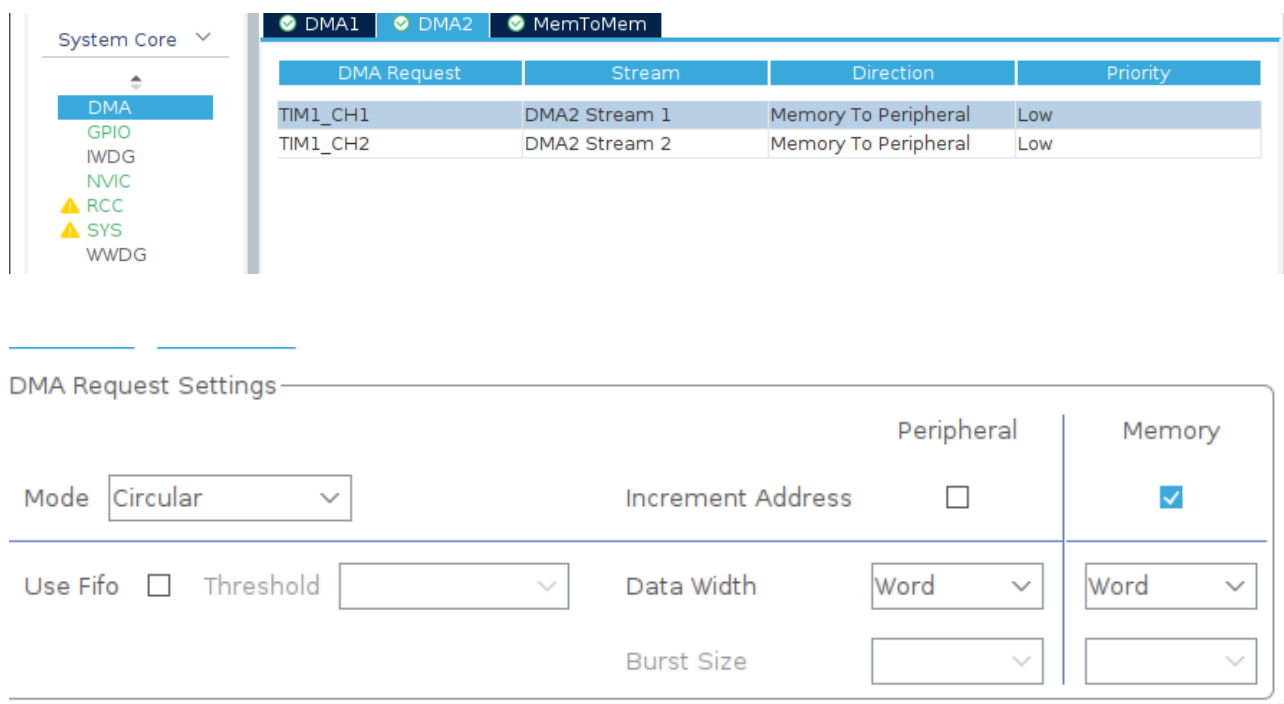
`pulse_ticks = SERVO_MIN_TICKS + (value * (SERVO_MAX_TICKS - SERVO_MIN_TICKS) / 180);`

gdzie:

- `SERVO_MIN_TICKS` – liczba cykli timera odpowiadająca ~1 ms.
- `SERVO_MAX_TICKS` – liczba cykli timera odpowiadająca ~2 ms.
- `value` – kąt w stopniach (0–180).

Jeżeli chcemy zrealizować automatyczny ruch (np. seria różnych kątów) – wykorzystujemy DMA. Dodajemy kolejne kąty do tablicy przez `PWM_DMA_AddValue()`, a następnie uruchamiamy `PWM_DMA_Start()`.

KONFIGURACJA DMA DLA TIM1



The screenshot displays the STM32CubeMX configuration interface for DMA. The top section shows the 'DMA' tab selected, with a table listing DMA requests for TIM1_CH1 and TIM1_CH2, both using DMA2 streams and configured for 'Memory To Peripheral' transfer with 'Low' priority. The bottom section shows the 'DMA Request Settings' for TIM1_CH1, configured in 'Circular' mode with 'Word' data width and 'Word' burst size.

DMA Request	Stream	Direction	Priority
TIM1_CH1	DMA2 Stream 1	Memory To Peripheral	Low
TIM1_CH2	DMA2 Stream 2	Memory To Peripheral	Low

DMA Request Settings

Peripheral		Memory
Mode: Circular	Increment Address: <input type="checkbox"/>	<input checked="" type="checkbox"/>
Use Fifo: <input type="checkbox"/> Threshold: <input type="text"/>	Data Width: Word	Word
	Burst Size: <input type="text"/>	<input type="text"/>

Jak już wcześniej napisałem, DMA zostało skonfigurowane w trybie circular.

W trybie circular kontrolera DMA po zakończeniu transferu określonej liczby danych, transfer automatycznie rozpoczyna się od nowa, bez interwencji procesora. Oznacza to, że po przesłaniu całego bufora danych, DMA wraca do początku bufora i kontynuuje transfer. Działanie w tym trybie **nie kończy się samoistnie** transfer będzie trwał w sposób ciągły, dopóki nie zostanie zatrzymany przez procesor lub wystąpią inne warunki przerwania.

Funkcja PWM_DMA_Stop zatrzymuje działanie PWM z DMA uprzednio upewniając się czy DMA jest aktywne przy pomocy zmiennej is_running używając funkcji HALOWSKIEJ dla serva podłączonego do TIM1 i kanału 1.

```
void PWM_DMA_Stop(void) {  
    if (!pwm_handler.is_running) {  
        return;  
    }  
  
    HAL_TIM_PWM_Stop_DMA(pwm_handler.htim, TIM_CHANNEL_1);  
    pwm_handler.is_running = 0;  
}
```

STEROWANIE

1. Czym są ticki:

Tick timera to najmniejsza jednostka czasu, którą timer potrafi zmierzyć. Jego czas zależy od częstotliwości zegara timera i ustawionego preskalera. W tym przypadku obliczyłem, że jeden tick trwa około 0,305 μ s.

Tick timera obliczyłem według wzoru:

$$T_{\text{tick}} = (PSC + 1) / F_{\text{clock}}$$

Przyjmijmy, że zegar timera F_{clock} wynosi 180 MHz, a prescaler (PSC) jest ustawiony na 54. Wówczas:

- $PSC + 1 = 55$
- $T_{\text{tick}} = 55 / 180\,000\,000$ sekund

Obliczając to:

$T_{\text{tick}} \approx 0.0000003056 \text{ s} = 0.3056 \mu\text{s}$

Stąd jeden tick trwa około 0,305 μs .

2. Przeliczenie 1 ms i 2 ms na ticki:

Aby uzyskać impuls trwający 1 ms (1000 μs), należy podzielić 1000 μs przez 0,305 μs , co daje około 3278 ticków. Analogicznie, 2 ms (2000 μs) to około 6556 ticków. Dlatego ustawienia są jako define:

- `SERVO_MIN_TICKS = 3278` (dla 1 ms)
- `SERVO_MAX_TICKS = 6556` (dla 2 ms)

3. Cel obliczenia impulsu PWM:

Dla sterowania serwem chcemy, aby impuls PWM (czas stanu wysokiego) mieścił się w przedziale od 1 ms do 2 ms, co odpowiada kątowi od 0° do 180°. Gdy kąt wynosi 0°, impuls powinien trwać 1 ms, a gdy kąt wynosi 180°, impuls powinien trwać 2 ms.

4. Interpolacja liniowa:

Wzór:

$$\text{pulse_ticks} = \text{SERVO_MIN_TICKS} + (\text{value} * (\text{SERVO_MAX_TICKS} - \text{SERVO_MIN_TICKS}) / 180);$$

działa w ten sposób, że gdy wartość value wynosi 0 (0°), to `pulse_ticks` = 3278 (czyli 1 ms), a gdy value wynosi 180 (180°), `pulse_ticks` = $3278 + (180 * (6556 - 3278) / 180) = 3278 + (6556 - 3278) = 6556$ (czyli 2 ms). Dla wartości pośrednich wyliczany jest odpowiedni, liniowo interpolowany impuls.

CZUJNIK ULTRADŹWIĘKOWY

Timer 3 – Input Capture dla czujnika ultradźwiękowego

- Timer pracuje jako Input Capture na kanale 1 – reaguje na zbocze rosnące i opadające sygnału ECHO.

- Prescaler = 44, Period = 38000 – wartości te są dobrane tak, by móc precyzyjnie odmierzyć czas trwania impulsu ECHO.

Timer 6 – okresowe wyzwalanie pomiaru ultradźwiękowego

- Timer6 co pewien czas (domyślnie co 100 ms) generuje przerwanie.
- W przerwaniu (HAL_TIM_PeriodElapsedCallback) wykonywana jest funkcja TriggerUltrasonic(), która wystawia krótki impuls TRIG.
- Dzięki temu pomiary odbywają się cyklicznie (można zmienić wartość wypełnienia i prescaler, by dostosować częstotliwość pomiaru).

Timer 7 – precyzyjny impuls TRIG (10 μ s)

```
static void MX_TIM7_Init(void){  
    htim7.Init.Prescaler = 179;  
    htim7.Init.Period = 9;  
}
```

- Timer7 służy do uzyskania krótkiego impulsu na pinie TRIG.
- W TriggerUltrasonic() ustawiany jest pin TRIG w stan wysoki i uruchamia się Timer7, który po 10 mikrosekundach wywołuje przerwanie i tam pin TRIG zostaje wyzerowany.

GPIO – konfiguracja linii TRIG

W MX_GPIO_Init():

```
GPIO_InitStruct.Pin = TRIG_Pin_Pin;
```

```
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
```

...

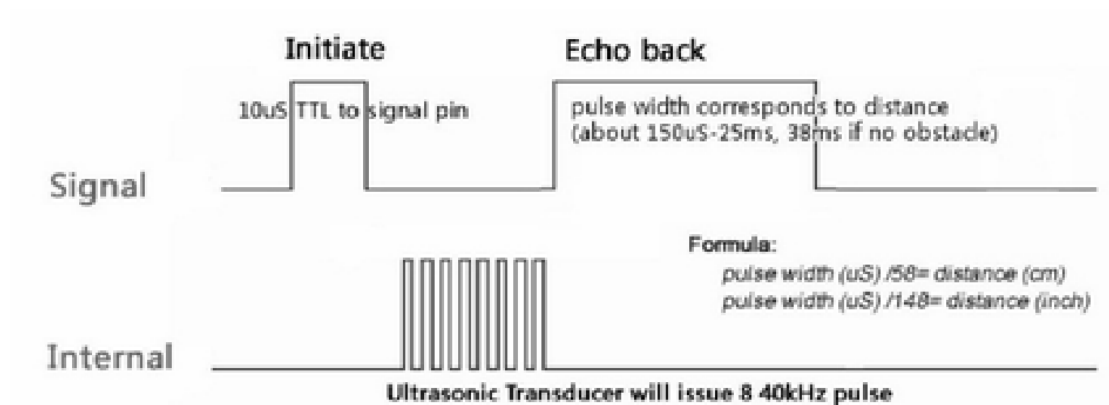
```
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Linia TRIG to pin, który musi zostać ustawiony na 10 μ s w stan wysoki, żeby wyzwolić pomiar czujnika ultradźwiękowego zgodnie z dokumentacją samego czujnika. Producent zaleca.

The timing diagram of HC-SR04 is shown. To start measurement, Trig of SR04 must receive a pulse of high (5V) for at least 10us, this will initiate the sensor will transmit out 8 cycle of ultrasonic burst at 40kHz and wait for the reflected ultrasonic burst. When the sensor detected ultrasonic from receiver, it will set the Echo pin to high (5V) and delay for a period (width) which proportion to distance. To obtain the distance, measure the width (Ton) of Echo pin.

Time = Width of Echo pulse, in uS (micro second)

- Distance in centimeters = Time / 58
- Distance in inches = Time / 148
- Or you can utilize the speed of sound, which is 340m/s



Note:

- Please connect the GND pin first before supplying power to VCC.
- Please make sure the surface of object to be detect should have at least 0.5 meter² for better performance.

USART2 – komunikacja szeregową i obsługa HDLC

- Wykorzystywane są funkcje przerwań do odbioru danych w sposób ciągły:
- HAL_UART_RxCpltCallback – automatycznie przenosi odebrane bajty do bufora USART_RxBuf.

- HAL_UART_TxCpltCallback – obsługuje stan bufora przy wysyłaniu danych.

W kodzie znajdują się również funkcje pomocnicze do odbioru i wysyłania danych (m.in. USART_getchar(), USART_fsend, HDLC_ProcessInput(), itp.).

Pomiar odległości – czujnik ultradźwiękowy

Zasada działania pomiaru

Ustawiamy pin TRIG w stan wysoki na 10 μ s.

Czujnik wystawia na pin ECHO stan wysoki przez czas zależny od odległości.

Mierzymy czas trwania stanu wysokiego (poprzez Timer3 w trybie Input Capture).

Przeliczamy:

zakładając, że prędkość dźwięku wynosi ~ 340 m/s

czas tam i z powrotem fali ultradźwiękowej

Konfiguracja Timer3 w trybie Input Capture

- Kanał TIM3_CH1 został skonfigurowany w trybie wychwytywania zboczy (Both Edges).
- W przerwaniu (HAL_TIM_IC_CaptureCallback) sprawdzam czy to zbocze narastające czy opadające.

```

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM3 && htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    {
        if (echo_captured == 0)
        {
            // Pierwsze zbocze narastające start impulsu
            echo_start = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            echo_captured = 1;

            // Zmień polaryzację na falling
            __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_ICPOLARITY_FALLING);
        }
        else if (echo_captured == 1)
        {
            // Drugie zbocze opadające koniec impulsu
            echo_end = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            echo_captured = 2; // pomiar zakończony

            // Przywróć polaryzację na rising (następny pomiar)
            __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_ICPOLARITY_RISING);
        }
    }
}

```

- Zmienna globalna echo_captured ustawia stan pomiaru (0 – czeka, 1 – start, 2 – koniec pomiaru).

Wyzwalanie TRIG i czas pomiędzy pomiarami

- Wyzwalanie TRIG następuje w TriggerUltrasonic(), która ustawia pin w stan wysoki i uruchamia Timer7.
- Timer7 po 10 μ s generuje przerwanie (HAL_TIM_PeriodElapsedCallback), w którym pin TRIG jest ustawiany w stan niski.
- Timer6 generuje przerwanie co 100 ms (domyślnie), w którym wywoływany jest TriggerUltrasonic().

W funkcji SetUltrasonicInterval jest funkcja HAL, wytłumaczenie jej działania.

HAL_RCC_GetPCLK1Freq():

Jest to funkcja HAL, która zwraca częstotliwość zegara magistrali APB1

Typowo dla STM32F1 jest to 72 MHz (72,000,000 Hz)

RCC = Reset and Clock Control

Obliczanie prescalera:

$$\text{prescaler} = (\text{timer_freq} / 1000) - 1$$

Obliczanie period:

$$\text{period} = \text{ms} - 1$$

Jeśli ms = 60 (minimalna przerwa dla HC-SR04):

$$\text{period} = 60 - 1 = 59$$

Częstotliwość timera po podzieleniu:

$$\text{Częstotliwość wyjściowa} = \text{timer_freq} / (\text{prescaler} + 1)$$
$$= 72,000,000 / 72,000$$
$$= 1,000 \text{ Hz (czyli 1 kHz)}$$

Czas przepełnienia timera:

Czas = (period + 1) / Częstotliwość wyjściowa
= 60 / 1000
= 0.06 sekundy (60 ms)

Podsumowując wszystko

TIM6 wywołuje `TriggerUltrasonic()`, aby rozpocząć nowy pomiar co określony czas i może być regulowany przy pomocy `SetUltrasonicInterval()`

TIM7 generuje impuls **TRIG** o długości 10 μ s

TIM3 przechwytuje sygnał **Echo** i zapisuje początek oraz koniec impulsu

Po zakończeniu pomiaru funkcja `GetUltrasonicDistance` oblicza odległość

Wynik jest zapisywany w buforze

Bufor kołowy na wyniki pomiarów

W strukturze `CircularBuffer` `cb`; przechowujemy ostatnie wyniki pomiarów odległości. Kluczowe funkcje:

- `CircularBuffer_Init()` – inicjalizacja bufora.
- `CircularBuffer_Put(&cb, dist);` – dodanie nowego pomiaru do bufora (jeśli pełny, nadpisuje najstarsze dane).
- `CircularBuffer_Get(&cb, &value);` – pobranie najstarszego elementu z bufora.
- `CircularBuffer_Size(&cb);` – ilość elementów w buforze.
- `CircularBuffer_Peek(&cb, pos, &value);` – podgląd elementu na danej pozycji, bez usuwania go.

Podsumowanie:

Ogólnie na koniec zauważyłem możliwy konflikt w momencie, gdy DMA chodzi a użytkownik chce dodać nową wartość do bufora z które właśnie korzysta DMA.

Konfliktem jest to, że zarówno procesor jak i DMA chcą mieć dostęp do tego samego bufora w tym samym czasie. Rozwiązaniem jakie zastosowałem jest sprawdzenie przed dodaniem wartości do bufora czy DMA jest aktywne, jeżeli tak to należy włączyć przerwanie (zastopować DMA) dodać wartość I aktywować ponownie DMA.

Źródła:

<https://forbot.pl/blog/kurs-stm32-f4-8-zaawansowane-funkcje-licznikow-id13473>

https://www.st.com/content/ccc/resource/technical/document/application_note/54/0f/67/eb/47/34/45/40/DM00042534.pdf/files/DM00042534.pdf/jcr:content/translations/en.DM00042534.pdf

STM32L4_WDG_TIMERS_GPTIM

STM32L4_WDG_TIMERS_GPTIM-1

<https://forbot.pl/blog/kurs-stm32-7-liczniki-timery-w-praktyce-pwm-id8459>

https://wiki.st.com/stm32mcu/wiki/Getting_started_with_DMA

<https://deepbluembedded.com/stm32-ultrasonic-sensor-input-capture-library/>

<https://controllerstech.com/servo-motor-with-stm32/>

<https://forbot.pl/blog/kurs-stm32-f1-hal-liczniki-timery-w-praktyce-pwm-id24334>

I wiele innych dziwnych stron oraz youtube