Projekt

Sterowniki robotów

I kamień milowy

Sterownik lotu drona sterowanego wektorem ciągu „Goose”

TVCG

Skład grupy:
Eryk Możdżeń, 259375

Termin: wtTN19

Prowadzący:
dr inż. Wojciech Domski

27 marca 2023

Spis treści

1 Zakres wykonanych prac **2**

2 Protokół komunikacyjny **3**

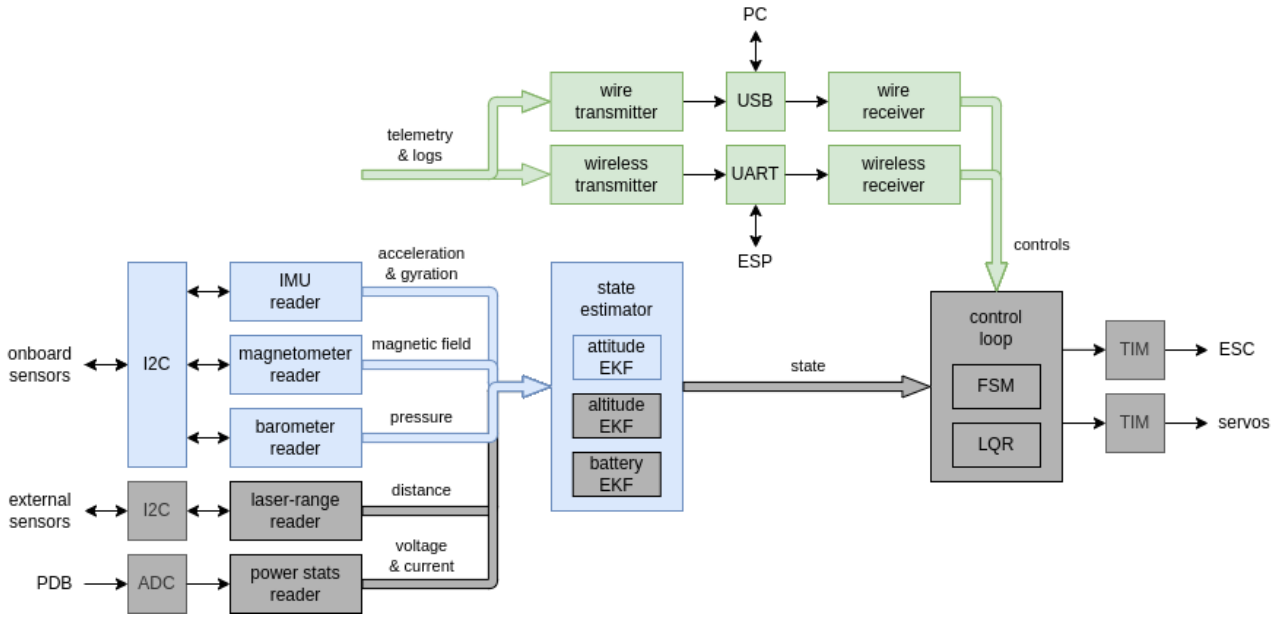
3 Komunikacja bezprzewodowa **6**

4 Komunikacja przewodowa **8**

1 Zakres wykonanych prac

Wszelkie prace prowadzone i dokumentowane były na repozytorium github[2].

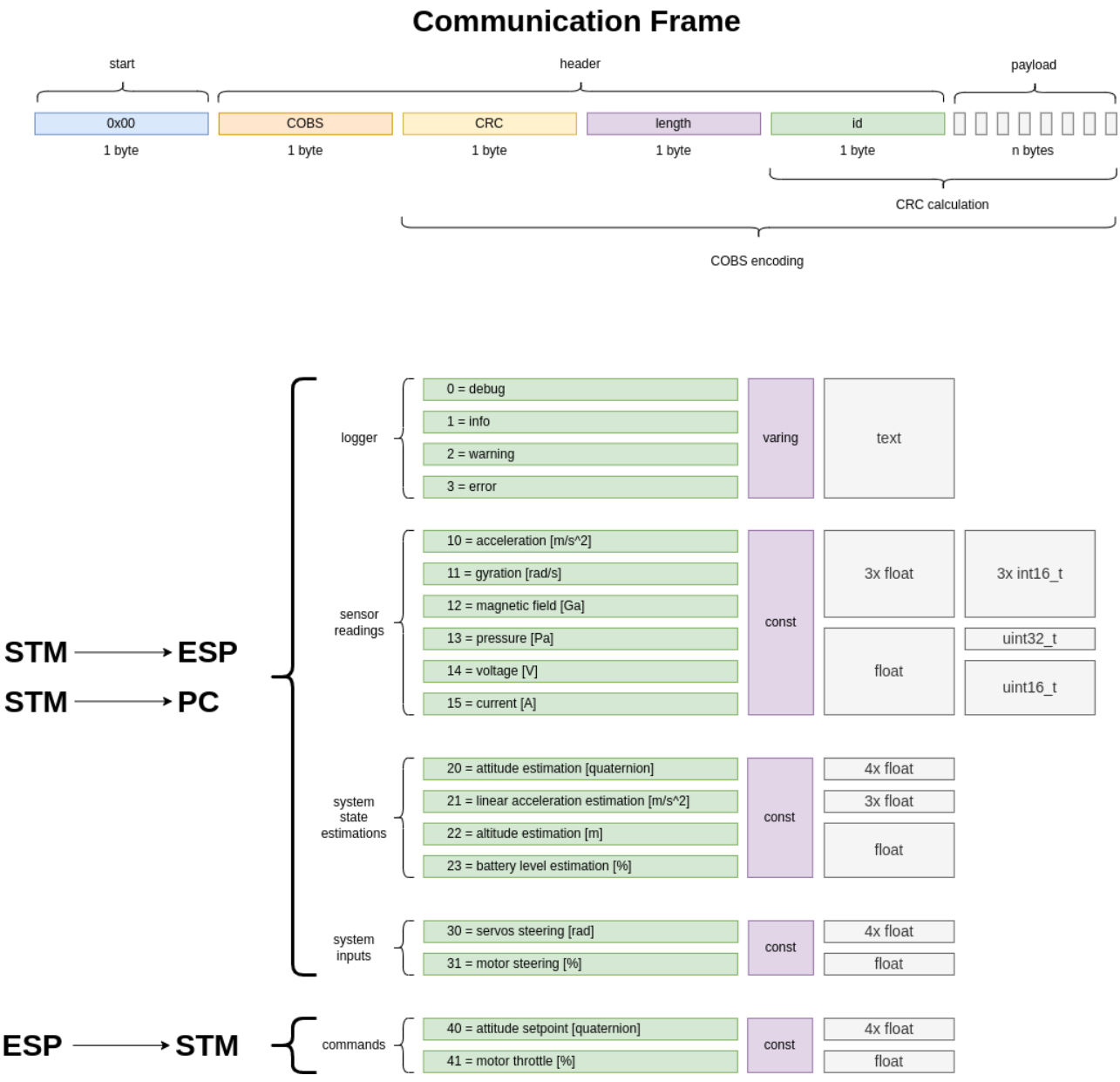
- implementacja protokołu oraz funkcji kodujących i dekodujących
- definicja danych wysyłanych w wiadomościach
- integracja protokołu kominikacji do wcześniej istenijacej architektury
- konfiguracja UART z użyciem DMA
- stworzenie tasków odbierających i nadawających po UART
- implementacja biblioteki USB oraz tasków ją obsługujące



Rysunek 1: Schemat architektury programu,
niebieski – wykonany przed oddaniem I etapu,
zielony – elementy komunikacji wykonane w ramach I kamienia milowego,
szary – planowane na następne kamienie milowe

2 Protokół komunikacyjny

Został zaimplementowany protokół komunikacyjny opisany w etapie I. Wykorzystuje on algorytmy takie jak COBS [3] oraz sumę konkrolną CRC8[4]. Niżej wypisano wszystkie ramki wiadomości, które będą używane do transmisji danych wraz z przydzielonym im identyfikatorami. Kod źródłowy klasy Transfer dzielony jest w programie mikrokontrolera oraz stacji pilota na komupterze PC.



Rysunek 2: Ramka danych oraz definicje wiadomości wraz z identyfikatorami

Kodowanie wiadomości

W pierwszej kolejności sprawdzany jest rozmiar wysyłanych danych. Zastosowano uproszczoną metodę kodowania COBS, która zawiera tylko jeden bit nadmiarowy. Ogranicza to długość jednej wiadomości do 255 bajtów, co odejmując narzut związany z obecnością nagłówka pozostawia 250 bajtów danych na treść wiadomości.

Po przekopiowaniu danych do bufora oraz uzupełnieniu pól identyfikatora oraz długości ładunku następuję obliczenie sumy kontrolnej na danych jak i id. Następnie suma kontrolna, długość, id oraz dane kodowane są przez COBS w celu usnięcia bajtów o wartości 0 z wiadomości. Na początku wpisywane jest 0.

```
template<typename T>
static FrameTX encode(const T &payload, const ID id) {
    static_assert(sizeof(T)<=max_length, "payload too big to encode");

    FrameTX frame;

    memcpy(&frame.buffer[5], &payload, sizeof(T));
    frame.buffer[4] = id;
    frame.buffer[3] = sizeof(T);
    frame.buffer[2] = calculate_CRC(&frame.buffer[4], 1 + sizeof(T));
    frame.buffer[1] = encode_COBS(&frame.buffer[2], 3 + sizeof(T));
    frame.buffer[0] = start_byte;

    frame.length = sizeof(T) + 5;

    return frame;
}
```

Listing 1: statyczna metoda kodująca w klasie Transfer

Dekodowanie wiadomości

W celu zdekodowania strumienia bajtów dochodzących do drona/wizalizacji wymagany jest obiekt klasy Transfer. Jej zadaniem jest akumulowanie kolejnych bajtów danych metodą consume() do czasu aż niezostanie zebrana pełna poprawna ramka danych.

```
void Transfer::consume(const uint8_t byte) {
  if(counter==0 && byte!=start_byte) {
    return;
  }

  if(counter>0 && byte==start_byte) {
    reset();
  }

  if(counter>=max_length+5) {
    reset();
  }

  frame_buffer[counter] = byte;
  counter++;
}
```

Listing 2: metoda akumulująca klasy Transfer

Aby sprawdzić czy już zostały otrzymane wszystkie bajty tworzące ramkę należy wywołać metodę receive(). Sprawdza ona czy obecnie zakumulowane dane, które spełniają sumę kontrolną oraz oczekiwaną ilość danych.

Początkowo sprawdzane są warunki tego, że pierwszym bajtem jest bajt startu, oraz że oprócz nagłówka zostały odebrane jakiekolwiek dane. Następnie przekopiowanie danych i odwrócenie kolejności ich odkodowywania względem ich zakodowywania.

```
1  bool Transfer::receive(FrameRX &frame) const {
2    if(counter<5) {
3      return false;
4    }
5
6    if(frame_buffer[0]!=start_byte) {
7      return false;
8    }
9
10   uint8_t copy[max_length + 5];
11   memcpy(copy, frame_buffer, counter);
12
13   const uint8_t cobs = copy[1];
14
15   decode_COBS(&copy[2], counter - 2, cobs);
16
17   frame.length = copy[3];
18   const size_t length_real = counter - 5;
19
20   if(frame.length!=length_real) {
21     return false;
22   }
23
24   const uint8_t crc = copy[2];
25   const uint8_t crc_real = calculate_CRC(&copy[4], frame.length + 1);
26
27   if(crc!=crc_real) {
28     return false;
29   }
30
31   frame.id = static_cast<ID>(copy[4]);
32
33   memcpy(frame.payload, &copy[5], frame.length);
34
35   return true;
36 }
```

Listing 3: metoda sprawdzająca klasy Transfer

3 Komunikacja bezprzewodowa

Główną drogą komunikacji docelowo będzie bezprzewodowa wymiana danych. W tym celu został użyty projekt BlackMagic–ESPIDF [1], z którego na potrzeby tego projektu wykorzystana będzie komunikacja po usłudze Telnet. Tak zaprogramowany mikrokontroler ESP8288 automatycznie łączy się na stałe zadaną siecią, i stanowi rolę pośrednika w wymianie Telnet – UART.

Do transmisji jak i odbioru danych używane jest DMA. Konfiguracja UART jest zgodna z opisaną w założeniach projektowych. Metody zostały napisane tak, aby task je wywołujacy przeszedłw tryb uśpienia podczas oczekiwania na zakończenie transmisji lub odczytu danych z użyciem notyfikacji z FreeRTOS. Całość kodu obsługująca UART została zamknięta w singletonie CommunicationBus, która została odpowiednio zabezpieczona przed dostępem z wielu tasków. Wielkość bufora została arbitralnie dobrana. Im większy zostanie dobrany bufor tym bardziej odciążany jest proceor, lecz tym żadziej reaguje na dane.

```
1  void CommunicationBus::write(const uint8_t *src, const uint32_t len) {
2    lock_tx.take(portMAX_DELAY);
3
4    com_bus_task_tx = xTaskGetCurrentTaskHandle();
5
6    HAL_UART_Transmit_DMA(&huart1, src, len);
7
8    if(!ulTaskNotifyTakeIndexed(2, pdTRUE, 100)) {
9      Logger::getInstance().log(Logger::ERROR, "cbus: tx timeout");
10   }
11
12   lock_tx.give();
13 }
14
15 void CommunicationBus::read(uint8_t *dest, const uint32_t len) {
16   lock_rx.take(portMAX_DELAY);
17
18   com_bus_task_rx = xTaskGetCurrentTaskHandle();
19
20   HAL_UART_Receive_DMA(&huart1, dest, len);
21
22   ulTaskNotifyTakeIndexed(3, pdTRUE, portMAX_DELAY);
23
24   lock_rx.give();
25 }
```

Listing 4: metody wysyłające i odbierające

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
  if(huart->Instance==USART1) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveIndexedFromISR(com_bus_task_tx, 2, &
        xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
  }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  if(huart->Instance==USART1) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveIndexedFromISR(com_bus_task_rx, 3, &
        xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
  }
}
```

Listing 5: reakcja na przerwanie DMA

Odbieranie danych przebiega w osobnym tasku, którego zadaniem jest oczekiwanie na dane do skutku. Po ich otrzymaniu następuję odkodowanie danych i wysłanie zdekodowanej ramki danych do kolejki. Został

```
void WirelessReceiver::task() {

  Transfer transfer;

  uint8_t buffer[8];
  Transfer::FrameRX frame;

  while(1) {
    CommunicationBus::getInstance().read(buffer, sizeof(buffer));

    for(uint32_t i=0; i<sizeof(buffer); i++) {
      transfer.consume(buffer[i]);

      if(transfer.receive(frame)) {
        Transport::getInstance().frame_rx_queue.push(frame, 0);
      }
    }
  }
}
```

Listing 6: task odbierający UART

Task wysyłajcy zajmuje się wyłacznie opróżnianiem kolejki wiadomości do wysłania oraz rozpoczynaniem i oczekiwaniem na koniec transmisji.

```
void WirelessTransmitter::task() {
  wireless_transmitter_task = getTaskHandle();

  while(1) {
    Transfer::FrameTX tx;
    Transport::getInstance().wireless_tx_queue.pop(tx, portMAX_DELAY);

    CommunicationBus::getInstance().write(tx.buffer, tx.length);
  }
}
```

Listing 7: task wysyłajacy UART

4 Komunikacja przewodowa

Do przewodowej komunikacji wykorzystany będzie USB w roli urządzenia CDC. Podobnie jak wcześniej taski próbujące wywołać/odczytać dane z USB powinny być uśpione do momentu zakończenia transmisji lub braku danych wejściowych. W tym celu zostały zmodyfikowane funkcje callback, które umożliwiają reakcje na wydarzenia.

```
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len) {
  USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);

  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  for(int i=*Len-1; i>=0; i--) {
    Transport::getInstance().wire_rx_queue.push_ISR(Buf[i],
        xHigherPriorityTaskWoken);
  }

  USBD_CDC_ReceivePacket(&hUsbDeviceFS);
  return USBD_OK;
}

uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len) {
  uint8_t result = USBD_OK;

  USBD_CDC_HandleTypeDef *hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.
      pClassData;
  if(hcdc->TxState != 0){
    return USBD_BUSY;
  }
  USBD_CDC_SetTxBuffer(&hUsbDeviceFS, Buf, Len);
  result = USBD_CDC_TransmitPacket(&hUsbDeviceFS);

  return result;
}
```

Listing 8: reakcje na callbacki biblioteki USB CDC

W tym wypadku odbiór danych polega na umieszczeniu danych w kolejce. Dla USB także stworzono dwa taski (odpowiednio odbierający i wysyłający) w sposób analogiczny do UART.

Bibliografia

[1] *Blackmagic Wireless SWD Debug probe hosted on esp-idf SDK (for ESP8266) with UART on Telnet port and HTTP using xterm.js*. URL: https://github.com/walmis/blackmagic-espidf.

[2] Eryk Możdżeń. *UAV TVC Goose*. URL: https://github.com/Eryk-Mozdzen/uav-tvc-goose.git.

[3] Wikipedia. *Consistent Overhead Byte Stuffing*. URL: https://en.wikipedia.org/wiki/Consistent_Overhead_Byte_Stuffing.

[4] Wikipedia. *Cykliczny kod nadmiarowy*. URL: https://pl.wikipedia.org/wiki/Cykliczny_kod_nadmiarowy.