# Założenia ogólne

Celem pracy JTC (Joint Trajectory Controller) jest realizacja zadanej trajektorii manipulatora o otwartym łańcuchu kinematycznym o sześciu stopniach swobody. Trajektoria do realizacji jest określona w przestrzeni współrzędnych konfiguracyjnych jako zestaw pozycji kątowych, prędkości kątowych i przyspieszeń kątowych dla każdego napędu. Sterowanie napędami odbywa się w sposób siłowy z uwzględnieniem wpływu regulatorów PID na kontrolę pozycji kątowej. Na podstawie zadanych wartości JTC wyznacza momenty sił, jakie napędy mają wypracować z uwzględnieniem wpływu regulatorów i współczynników kompensacji tarcia wewnątrz napędów.

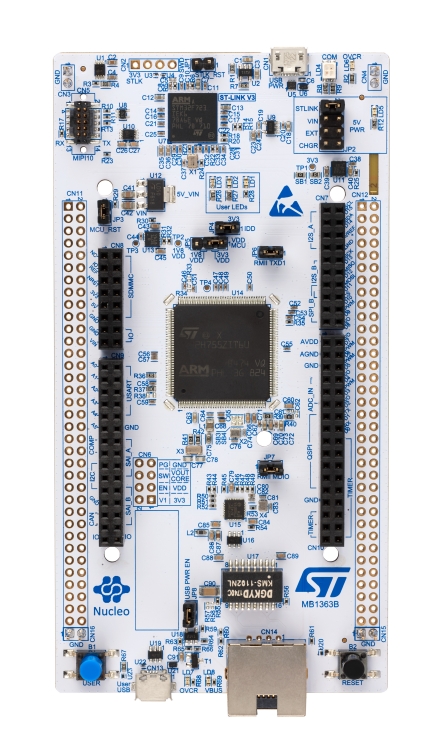
Podstawowe założenia co do działania JTC (Joint Trajectory Controller):

* zdolność do komunikacji z układem hosta poprzez port szeregowy UART i interfejs RS422,
  + odczyt zadanej trajektorii w postaci pozycji, prędkości i przyspieszenia kątowego dla każdego z sześciu napędów
  + odczyt tablic zawierających współczynniki kompensacji tarcia lub odczyt komendy użycia domyślnych tablic współczynników kompensacji tarcia dla każdego z sześciu napędów
  + odczyt nastaw regulatorów PID lub odczyt komendy użycia domyślnych nastaw regulatorów PID dla każdego z sześciu napędów
  + odczyt parametrów kinematycznych i dynamicznych manipulatora lub odczyt komendy użycia domyślnych wartości parametrów kinematycznych i dynamicznych manipulatora
  + odczyt komend sterujących pozwalających na uruchomienie, wstrzymanie lub zatrzymanie wykonywania trajektorii
  + odczyt komend sterujących pozwalających na czyszczenie błędów
  + zapis do układu hosta danych telemetrycznych oraz potwierdzeń
* realizacja trajektorii w przestrzeni konfiguracyjnej dla manipulatora o otwartym łańcuchu kinematycznym o sześciu stopniach swobody,
  + interpolacja trajektorii w celu zwiększenia częstotliwości punktów do 1kHz,
  + rozwiązanie zadania odwrotnego dynamiki
  + realizacja algorytmów regulatorów PID
  + interpolacja tablic zawierających współczynniki kompensacji tarcia
* zdolność do komunikacji z sześcioma sterownikami napędów poprzez magistralę FDCAN,
  + cykliczna transmisja (1kHz) danych do sterowników napędów, zawierających informacje o wartości zadanego momentu siły oraz o zadanym stanie pracy sterownika
  + cykliczny odbiór (1kHz) danych telemetrycznych ze sterowników napędów zawierających informacje o aktualnej pozycji, prędkości, momencie siły, temperaturze, stanie maszyny, błędach i ostrzeżeniach.

# Wymagany hardware i software

## Hardware

JTC bazuje na mikrokontrolerze z serii STM32H7. Jest kompatybilny z mikrokontrolerem STM32H745ZI i płytką testową NUCLEO-H745ZI-Q. Może być również łatwo dostosowany do mikrokontrolera STM32H743ZI i płytki testowej NUCLEO-H743ZI2. Poniżej przedstawiono wygląd modułu NUCLEO-H745ZI-Q2.

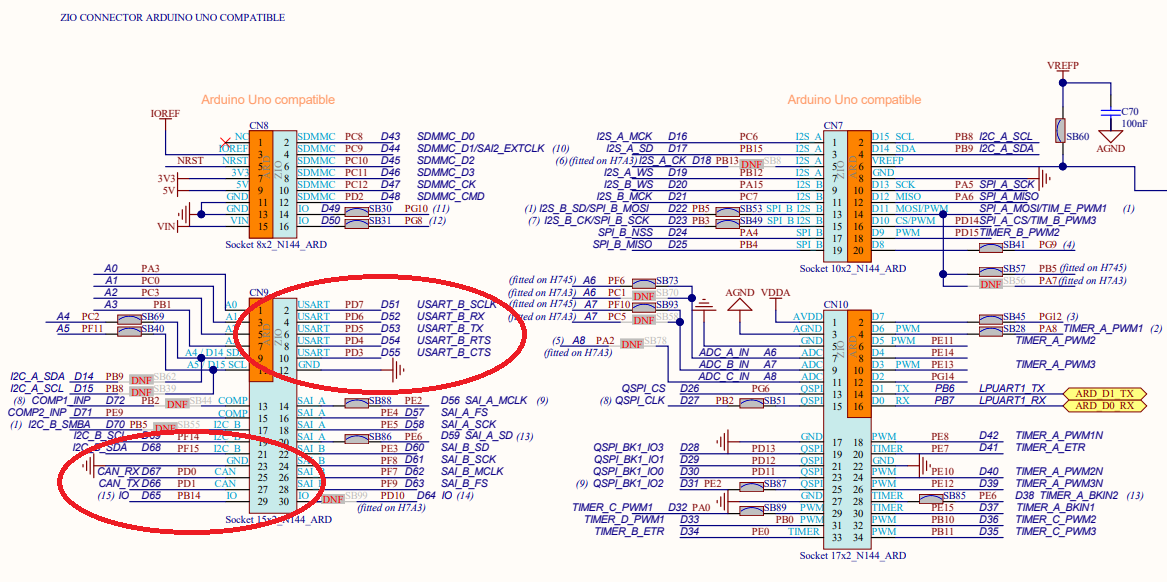


#### Rys. . Moduł NUCLEO-H745ZI-Q2

Wszystkie wyprowadzenie mikrokontrolera wykorzystywane do realizacji funkcji JTC dostępne są w formie złącz kompatybilnych z Arduino lub poprzez gniazdo USB debuggera / programatora ST-Link. Poniżej zestawiono wykorzystywane wyprowadzenia mikrokontrolera:

#### Tabela 2.1 JTC pinout description

|  |  |  |
| --- | --- | --- |
| **GPIO number** | **Function** | **Description** |
| GPIOD.0 | FDCAN RX | FDCAN bus receiving line |
| GPIOD.1 | FDCAN TX | FDCAN bus transmitting line |
| GPIOD.3 | Safety Input | Emergency input: 0 – error, 1 – no error |
| GPIOD.4 | Safety Output | Emergency output: 0 – error, 1 – no error |
| GPIOD.5 | UART2 TX | UART transmitting line – target uart for RS422 on arduino connector |
| GPIOD.6 | UART2 RX | UART receiving line – target uart for RS422 on arduino connector |
| GPIOD.8 | UART3 TX | UART transmitting line – temporary USB uart via ST-Link |
| GPIOD.9 | UART3 RX | UART receiving line – temporary USB uart via ST-Link |



#### Rys. . JTC pinout

## Software

Oprogramowanie układowe JTC zostało opracowane w środowisku programistycznym Keil uVision (MDK Arm <https://www.keil.com/>).

W przypadku domyślnym, gdy używany jest mikrokontroler STM32H745ZI projekt musi zostać przygotowany na obydwa rdzenie mikrokontrolera (Cortex-M4 i Cortex-M7). W pierwszej kolejności należy jednorazowo załadować plik wykonywalny na rdzeń Cortex-M4. Następnie należy załadować plik wykonywalny na rdzeń Cortex-M7. W przypadku modyfikacji projektu nie ma potrzeby ponownego ładowania pliku wykonywalnego na rdzeń Cortex-M4. Wszystkie obliczenia odbywają się na rdzeniu Cortex-M7.

W przypadku zmiany platformy na mikrokontroler STM32H743ZI utworzony projekt zawiera oprogramowanie układowe tylko dla rdzenia Cortex-M7. Aby dostosować pliki źródłowe do nowej platformy należy zmienić definicję dołączonego pliku nagłówkowego z:

##### #include <stm32h7xx.h>

##### #include <stm32h745xx.h>

na definicję:

##### #include <stm32h7xx.h>

##### #include <stm32h743xx.h>

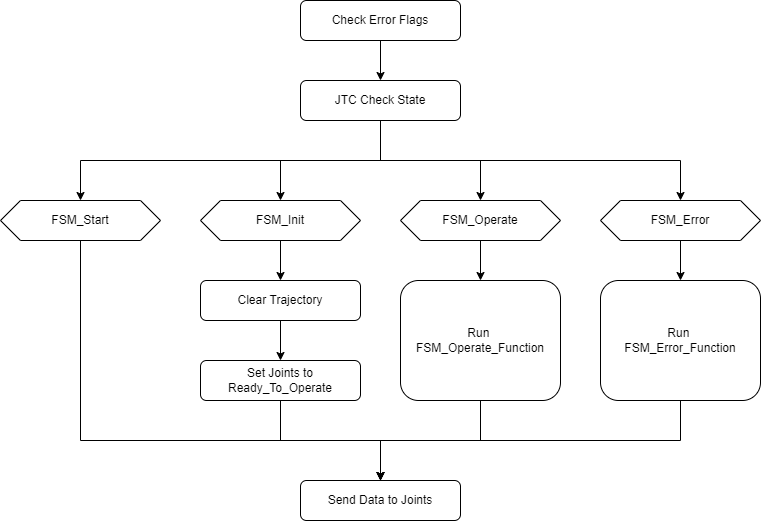
# Logika działania

## Ogólny schemat sterowania

JTC (Joint Trajectory Controller) może pracować w jednym z czterech trybów:

* FSM\_Start – etap zaraz po resecie procesora, inicjalizacja systemu,
* FSM\_Init – inicjalizacja napędów, inicjalizacja danych z układu hosta,
* FSM\_Operate – normalna praca urządzenia
* FSM\_Error – obsługa błędów.

Na poniższym rysunku przedstawiono ogólny schemat działania głównej funkcji programu. W pierwszej kolejności oprogramowanie sprawdza flagi błędów. Następnie sprawdza aktualny stan urządzenia oraz docelowy stan urządzenia i dokonuje jego ewentualnej zmiany. Dalej. realizowana jest obsługa jednego z czterech trybów pracy. Po ich wykonaniu realizowana jest komunikacja z jointami. Komunikacja z układem hosta odbywa się w sposób niezależny od głównej funkcji programu.



#### Rys. . JTC General scheme of operation

Poniżej przedstawiono główną funkcję programu.

##### static void Control\_JtcAct(void)

##### {

##### LED1\_OFF; LED2\_OFF; LED3\_OFF;

##### Control\_CheckErrorFlags();

##### Control\_JtcCheckState();

##### if(pC->Jtc.currentFsm == JTC\_FSM\_Start)

##### {

##### return;

##### }

##### else if(pC->Jtc.currentFsm == JTC\_FSM\_Init)

##### {

##### LED1\_ON;

##### Control\_JtcInit();

##### }

##### else if(pC->Jtc.currentFsm == JTC\_FSM\_Operate)

##### {

##### LED2\_ON;

##### Control\_JtcOperate();

##### }

##### else if(pC->Jtc.currentFsm == JTC\_FSM\_Error)

##### {

##### LED3\_ON;

##### Control\_JtcError();

##### }

##### Control\_SendDataToJoints();

##### }

## FSM\_Start

Zaraz po resecie procesora JTC znajduje się w trybie FSM\_Start. W tym trybie nie jest wykonywana jeszcze główna funkcja programu. W tym trybie uruchamiane są wewnętrzne układy mikrokontrolera:

* zmiana częstotliwości taktowania rdzenia na 480MHz,
* uruchomienie licznika systemowego z przerwaniem o częstotliwości 1kHz,
* uruchomienie magistral zegarowych,
* konfiguracja LED
* konfiguracja zmiennych globalnych
* konfiguracja portów szeregowych do komunikacji z układem hosta,
* konfiguracja magistrali FDCAN do komunikacji z napędami,
* konfiguracja parametrów kinematycznych i dynamicznych manipulatora do obliczeń dynamiki odwrotnej,
* konfiguracja wejścia i wyjścia bezpieczeństwa,
* konfiguracja i uruchomienie licznika z przerwaniem 1kHz do obsługi głównej funkcji programu,

Po wykonaniu powyższej procedury urządzenie przechodzi automatycznie w tryb FSM\_Init.

## FSM\_Init

W tym trybie JTC dokonuje inicjalizacji napędów i inicjalizacji zmiennych dotyczących działania urządzenia. Inicjalizacja napędów polega na wysyłaniu poprzez FDCAN do napędów komendy Joint\_FSM\_Init i oczekiwaniu na odebranie od każdego napędu komendy Joint\_FSM\_ReadyToOperate. Inicjalizacja danych z układu hosta polega na oczekiwaniu na odebranie z informacji o wartościach tablic współczynników tarcia, wartościach nastaw regulatorów PID i wartościach parametrów kinematycznych i dynamicznych manipulatora. Układ hosta może przesłać wartości tych parametrów lub wydać komendę użycia wartości domyślnych. Jeżeli w trakcie pracy którykolwiek z napędów powróci do stanu Joint\_FSM\_Init JTC również powróci do fazy inicjalizacji.

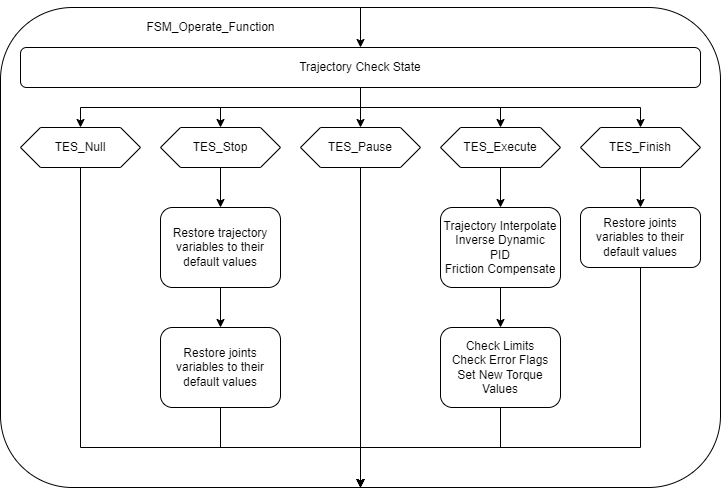
## FSM\_Operate

JTC znajduje się w trybie FSM\_Operate tylko wtedy gdy: wszystkie napędy zostały poprawnie zainicjalizowane, wartości zmiennych z układu hosta zostały poprawnie zainicjalizowane, nie wystąpił żaden błąd. W tym trybie możliwa jest realizacja trajektorii.

Trajektoria może znajdować się w jednym z pięciu podstawowych trybów realizacji:

* TES\_Null – brak odebranej trajektorii
* TES\_Stop – trajektoria odebrana, realizacja zatrzymana,
* TES\_Pause – trajektoria odebrana, realizacja wstrzymana,
* TES\_Execute – trajektoria odebrana, w realizacji,
* TES\_Finish – trajektoria zakończona.

W trybie FSM\_Operate JTC w pierwszej kolejności sprawdza aktualny i zadany status realizacji trajektorii. Następnie dokonuje jego przełączenia. Dalej na podstawie aktualnego status realizacji trajektorii wykonywana jej realizacja.



#### Rys. . FSM\_Operate scheme of operation

Jeżeli status realizacji wynosi TES\_Null, oznacza to, że trajektoria nie została odebrana z układu hosta lub została wyczyszczona przez JTC, na przykład z powodu wcześniejszych błędów. W tej sytuacji należy przesłać trajektorię ponownie. W tym trybie napędy znajdują się w stanie Joint\_FSM\_ReadyToOperate.

Jeżeli status realizacji wynosi TES\_Stop oznacza to, że trajektoria została odebrana, lecz jej realizacja jeszcze nie została rozpoczęta. Zmienne dotyczące realizacji trajektorii są wyczyszczone i przywrócone do wartości startowych. Należy uruchomić realizację trajektorii wysyłając stosowne polecenie. W tym trybie napędy znajdują się w stanie Joint\_FSM\_ReadyToOperate.

Jeżeli status realizacji wynosi TES\_Pause oznacza to, że trajektoria została odebrana, lecz jej realizacja została wstrzymana. Należy wznowić lub zatrzymać realizację trajektorii wysyłając stosowne polecenie. W tym trybie napędy znajdują się w stanie Joint\_FSM\_ReadyToOperate.

Jeżeli status realizacji wynosi TES\_Execute oznacza to, że trajektoria została odebrana, i jest realizowana. Realizację można wstrzymać lub zatrzymać wysyłając stosowne polecenie. Realizacja poszczególnych punktów trajektorii odbywa się następująco. W pierwszej kolejności sprawdzana jest wartość licznika punktów trajektorii i sprawdzana jest możliwość osiągnięcia końca trajektorii. Następnie JTC interpoluje trajektorię aby uzyskać krok czasowy równy 1ms. Dalej rozwiązywane jest zadanie odwrotne dynamiki. Następnie wyznaczane są wartości korekcyjnych momentów sił z regulatorów PID. Dalej wyznaczane są wartości współczynników kompensacji tarcia (interpolacja dwuwymiarowa tablic). Na podstawie powyższych wartości wyznaczany jest docelowy moment siły dla każdego napędu. Następnie sprawdzane są limity dla poszczególnych wartości (moment siły, pozycja kątowa, prędkość kątowa, przyspieszenie kątowe, uchyb pozycji). Dalej sprawdzane są flagi błędów aby uwzględnić powyższe limity wartości. Jeżeli błędy nie występują wyliczone wartości momentów zadanych sa przekazywane do dalsze wysłania do napędów. W tym trybie napędy znajdują się w stanie Joint\_FSM\_OperationEnable.

Jeżeli status realizacji wynosi TES\_Finish oznacza to, że trajektoria została odebrana, a jej realizacja została zakończona. Należy przesłać nową trajektorię, ponownie uruchomić aktualną. W tym trybie napędy znajdują się w stanie Joint\_FSM\_ReadyToOperate.

## FSM\_Error

JTC przechodzi w FSM\_Error w przypadku wystąpienia dowolnego błędu. Błędy podzielone są na dwie główne kategorie: błędy zewnętrzne i błędy wewnętrzne.

Do błędów zewnętrznych należą:

* wykrycie stanu niskiego na wejściu bezpieczeństwa
* odebranie z dowolnego napędu informacji o błędzie poprzez magistralę FDCAN

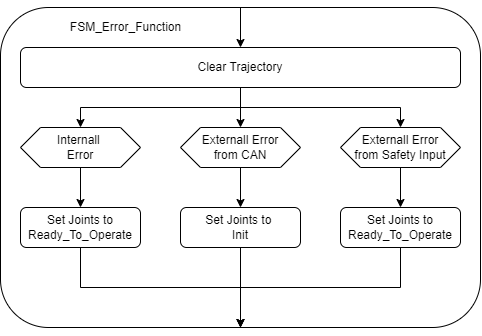
Do błędów wewnętrznych należą:

* błędy w komunikacji z napędami poprzez magistralę FDCAN (timeout),
* błąd w komunikacji z układem hosta (timeout – aktualnie nie obsługiwany),
* błąd dotyczący przekroczenia dopuszczalnych zakresów dla wyliczonych parametrów napędów (dotyczy momentu siły, pozycji kątowej, prędkości kątowej, przyspieszenia kątowego, uchybu pozycji, braku odpowiedniej wartości współczynnika kompensacji tarcia w tablicy).

W przypadku wykrycia błędu wewnętrznego uruchamiane jest wyjście bezpieczeństwa. W tej sytuacji napędy przechodzą w stan Joint\_FSM\_ReadyToOperate.

W przypadku wykrycia błędu zewnętrznego spowodowanego odebraniem poprzez FDCAN komunikatu o błędzie z dowolnego napędu, JTC ponownie inicjalizuje dany napęd poleceniem Joint\_FSM\_Init. Pozostałe napędy przechodzą w stan Joint\_FSM\_ReadyToOperate. W przypadku wykrycia błędu zewnętrznego spowodowanego stanem niskim na wejściu bezpieczeństwa napędy przechodzą w stan Joint\_FSM\_ReadyToOperate.

JTC w czasie rzeczywistym przesyła do układu hosta informacje o wszystkich wykrytych błędach.



#### Rys. . FSM\_Error scheme of operation

# Komunikacja JTC – Joints

JTC jest przystosowany do prowadzenia komunikacji z sześcioma napędami poprzez magistralę FDCAN. Częstotliwość pracy rdzenia FDCAN w JTC wynosi 85MHz. Prędkość w trybie nominalnym wynosi 1Mbps, natomiast prędkość w trybie danych wynosi 5Mbps. Magistrala pracuje w trybie FD z włączoną zmienną prędkością komunikacji. Parametry konfiguracyjne FDCAN przedstawiono w tabeli.

#### Tabela 4.1 Konfiguracja magistrali FDCAN

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| FDCAN core frequency | 85 MHz |
| Baudrate in nominal mode | 1 Mbps |
| Baudrate in data mode | 5 Mbps |
| Nominal Prescaler | 4 + 1 |
| Nominal Sync Jump Width | 1 + 1 |
| Nominal Time Seg1 | 11 + 1 |
| Nominal Time Seg2 | 3 + 1 |
| Data Prescaler | 0 + 1 |
| Data Sync Jump Width | 1 + 1 |
| Data Time Seg1 | 11 + 1 |
| Data Time Seg2 | 3 + 1 |

Funkcja konfiguracyjna FDCAN:

##### static void Can\_FdcanConf(void)

##### {

##### // pin configuration

##### GPIOD->MODER &= ~GPIO\_MODER\_MODE0 & ~GPIO\_MODER\_MODE1;

##### GPIOD->MODER |= GPIO\_MODER\_MODE0\_1 | GPIO\_MODER\_MODE1\_1;

##### GPIOD->AFR[0] |= 0x00000099;

##### 

##### // FDCAN setup begins

##### FDCAN1->CCCR |= FDCAN\_CCCR\_INIT | FDCAN\_CCCR\_CCE;

##### // Frame in CANFD format with variable speed (BRSE bit)

##### FDCAN1->CCCR |= FDCAN\_CCCR\_BRSE | FDCAN\_CCCR\_FDOE | FDCAN\_CCCR\_TXP;

##### 

##### // transmission baudrate in nominal mode

##### FDCAN1->NBTP = (0x01 << FDCAN\_NBTP\_NSJW\_Pos) | (0x04 << FDCAN\_NBTP\_NBRP\_Pos) | (0x0B << FDCAN\_NBTP\_NTSEG1\_Pos) | (0x03 << FDCAN\_NBTP\_NTSEG2\_Pos);

##### // transmission baudrate in data mode

##### FDCAN1->DBTP = (0x01 << FDCAN\_DBTP\_DSJW\_Pos) | (0x00 << FDCAN\_DBTP\_DBRP\_Pos) | (0x0B << FDCAN\_DBTP\_DTSEG1\_Pos) | (0x03 << FDCAN\_DBTP\_DTSEG2\_Pos);

##### 

##### // all remote and non-filtered frames are discarded

##### FDCAN1->GFC = (0x03 << FDCAN\_GFC\_ANFS\_Pos) | (0x03 << FDCAN\_GFC\_ANFE\_Pos) | FDCAN\_GFC\_RRFS | FDCAN\_GFC\_RRFE;

##### // CAN\_FILTERS\_MAX of standard filters and the address of the filters

##### FDCAN1->SIDFC = (CAN\_FILTERS\_MAX << FDCAN\_SIDFC\_LSS\_Pos) | (pC->Can.filterAddrOffset << 0);

##### 

##### // CAN\_TXBUF\_MAX send buffers and address of the first send buffer

##### FDCAN1->TXBC = (CAN\_TXBUF\_MAX << FDCAN\_TXBC\_NDTB\_Pos) | (pC->Can.txBufAddrOffset << 0);

##### // transmit buffers with a size of 20 bytes

##### FDCAN1->TXESC = (CAN\_TXBUFSIZE\_CODE << FDCAN\_TXESC\_TBDS\_Pos);

##### 

##### // 12-byte receiving buffers, RXFIFO 1 and RXFIFO 0 elements with a size of 12 bytes

##### FDCAN1->RXESC = (CAN\_RXBUFSIZE\_CODE << FDCAN\_RXESC\_RBDS\_Pos) | (CAN\_RXBUFSIZE\_CODE << FDCAN\_RXESC\_F1DS\_Pos) | (CAN\_RXBUFSIZE\_CODE << FDCAN\_RXESC\_F0DS\_Pos);

##### // first receive buffer address offset

##### FDCAN1->RXBC = (pC->Can.rxBufAddrOffset << 0);

##### // CAN\_RXBUFF\_MAX receive buffers, CAN\_RXFIFO0\_MAX fifo0 and address of first fifo0

##### FDCAN1->RXF0C = (CAN\_RXFIFO0\_MAX << FDCAN\_RXF0C\_F0S\_Pos) | (pC->Can.rxFifo0AddrOffset << 0);

##### // abort from receive to buffer, these interrupts are directed to EINT0

##### FDCAN1->IE = FDCAN\_IE\_TCE | FDCAN\_IE\_DRXE;

##### // Enable interrupts from transfer complete individually for each send buffer

##### for(int i=0;i<CAN\_TXBUF\_MAX;i++)

##### FDCAN1->TXBTIE |= (1 << i);

##### // The error handling interrupt, these interrupts are directed to EINT1

##### FDCAN1->IE |= FDCAN\_IE\_ARAE | FDCAN\_IE\_PEDE | FDCAN\_IE\_PEAE | FDCAN\_IE\_WDIE | FDCAN\_IE\_BOE | FDCAN\_IE\_EWE | FDCAN\_IE\_EPE | FDCAN\_IE\_ELOE;

##### FDCAN1->ILS = FDCAN\_ILS\_ARAE | FDCAN\_ILS\_PEDE | FDCAN\_ILS\_PEAE | FDCAN\_ILS\_WDIE | FDCAN\_ILS\_BOE | FDCAN\_ILS\_EWE | FDCAN\_ILS\_EPE | FDCAN\_ILS\_ELOE;

##### // turning on the interrupt line

##### FDCAN1->ILE = FDCAN\_ILE\_EINT0 | FDCAN\_ILE\_EINT1;

##### NVIC\_EnableIRQ(FDCAN1\_IT0\_IRQn);

##### NVIC\_EnableIRQ(FDCAN1\_IT1\_IRQn);

##### }

Komunikacja odbywa się w sposób cykliczny z częstotliwością 1kHz. W każdym cyklu komunikacji JTC wysyła ramkę rozgłoszeniową, która odbierana jest przez wszystkie napędy. Ramka ma długość 20 bajtów. Ramka ma postać:

#### Tabela 4.2 JTC to Joints broadcast frame

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value / Range** |
| ID | Header [B0] | uint8\_t | 0xAA |
| 0 - 1 | Joint 0 Torque [B1 – B0] | int16\_t | - |
| 2 | Joint 0 FSM [B0] | uint8\_t | 0x01 – FSM\_Init  0x02 – FSM\_ReadyToOperate  0x03 – FSM\_OperationEnable |
| 3 - 4 | Joint 1 Torque [B1 – B0] | int16\_t |  |
| … | … | … | … |
| 17 | Joint 5 FSM [B0] | uint8\_t | 0x01 – FSM\_Init  0x02 – FSM\_ReadyToOperate  0x03 – FSM\_OperationEnable |
| 18 | - | uint8\_t | 0x00 |
| 19 | - | uint8\_t | 0x00 |

Następnie w odpowiedzi każdy napęd odsyła ramkę telemetryczną o długości 12 bajtów. Ramki te mają postać:

#### Tabela 4.3 Joint n to JTC frame

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value / Range** |
| ID | Header [B0] | uint8\_t | Joint 0 - 0xA0, Joint 1 - 0xB0, Joint 2 - 0xC0,  Joint 3 - 0xD0, Joint 4 - 0xE0, Joint 5 - 0xF0 |
| 0 - 1 | Position [B1 – B0] | int16\_t | - |
| 2 - 3 | Velocity [B1 – B0] | int16\_t | - |
| 4 - 5 | Torque [B1 – B0] | int16\_t | - |
| 6 | Temperature [B0] | unt8\_t | - |
| 7 | FSM [B0] | uint8\_t | 0x00 – FSM\_Start  0x01 – FSM\_Init  0x02 – FSM\_ReadyToOperate  0x03 – FSM\_OperationEnable  0x0A – FSM\_TransStartToInit  0x0B – FSM\_TransInitToReadyToOperate  0x0C – FSM\_TransReadyToOperateToOperationEnable  0x0D – FSM\_TransOperationEnableToReadyToOperate  0x0E – FSM\_TransFaulyReactionActiveToFault  0x0F – FSM\_TransFaultToReadyToOperate  0xFE – FSM\_ReactionActive  0xFF – FSM\_Fault |
| 8 | MC Current Error [B0] | uint8\_t | 0x00 – MC\_NO\_ERRORS / MC\_NO\_FAULTS  0x01 – MC\_FOC\_DURATION  0x02 – MC\_OVER\_VOLT  0x04 – MC\_UNDER\_VOLT  0x08 – MC\_OVER\_TEMP  0x10 – MC\_START\_UP  0x20 – MC\_SPEED\_FDBK  0x40 – MC\_BREAK\_IN  0x80 – MC\_SW\_ERROR |
| 9 | MC Occured Error [B0] | uint8\_t | 0x00 – MC\_NO\_ERRORS / MC\_NO\_FAULTS  0x01 – MC\_FOC\_DURATION  0x02 – MC\_OVER\_VOLT  0x04 – MC\_UNDER\_VOLT  0x08 – MC\_OVER\_TEMP  0x10 – MC\_START\_UP  0x20 – MC\_SPEED\_FDBK  0x40 – MC\_BREAK\_IN  0x80 – MC\_SW\_ERROR |
| 10 | Current Error [B0] | uint8\_t | 0x00 – JOINT\_NO\_ERROR  0x01 – JOINT\_POSITION\_ENCODER\_FAILED  0x02 – JOINT\_MC\_FAILED |
| 11 | Current Warning | uint8\_t | 0x00 – JOINT\_NO\_WARNING  0x01 – JOINT\_POSITION\_NOT\_ACCURATE  0x02 – JOINT\_OUTSIDE\_WORKING\_AREA |

# Komunikacja JTC – Host

## Wstęp

Komunikacja pomiędzy JTC a układem hosta odbywa się za pomocą protokołu UART z wykorzystaniem interfejsu RS422. Komunikacja odbywa się w trybie asynchronicznym, pełny dupleks, point-to-point. Format ramki 115200 8N1, (prędkość 115200 bps, 8 bitów danych, 1 bit stopu, bez kontroli parzystości, bez sprzętowej kontroli przepływu). Układ hosta powinien zarezerwować osobny port COM dla każdego podłączonego JTC.

#### Tabela 5.1 Komunikacja JTC – Host: parametry komunikacji

|  |  |
| --- | --- |
| **Parametr** | **Value** |
| Tryb | Asynchroniczny, pełny duplex |
| Prędkość | 115200 bps (max. 15Mbps) |
| Liczba bitów danych | 8 |
| Liczba bitów stopu | 1 |
| Kontrola parzystości | Brak |
| Sprzętowa kontrola przepływu | Brak |

## Format danych i kontrola błędów

Dane przesyłane są w sposób binarny w formie ramek. Ramka wysyłana do JTC musi być transmitowana w sposób spójny, bez przerw czasowych. Koniec ramki wykrywany jest sprzętowo poprzez wykrycie stanu bezczynności na linii odbiorczej przez czas trwania transmisji dwóch bitów. Urządzenie wysyłające jest odpowiedzialne za zapewnienie spójności transmisji. Jeżeli ramka będzie transmitowana z  przerwami, zostanie przez JTC potraktowana jako kilka osobnych pakietów. W tej sytuacji JTC ma zaimplementowany mechanizm scalania z timeout. Jeżeli przerwa pomiędzy pakietami będzie mniejsza niż 5ms (do ustalenia – wartość zależna od prędkości i sposobu komunikacji – w urządzeniach RT przerwy nie powinny w ogóle występować), pakiety zostaną scalone do jednej ramki. Mechanizm scalania uwzględnia dane w ramce, w szczególności oczekiwaną długość ramki (pole n – drugi i trzeci bajt). Jeżeli timeout zostanie przekroczony, JTC odeśle stosowny komunikat. JTC zawsze transmituje ramki w sposób spójny, bez przerw czasowych. Ogólny schemat ramki przedstawiono w tabela 5.2.

#### Tabela 5.2 Ogólny schemat ramki komunikacyjnej JTC - Host

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value / Range** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x00 – 0xff | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 – 0xEA60 | Length frame in bytes |
| 3 | n [B0] |
| 4 | Data 0 | uint8\_t | Depends on the variable | Example data |
| 5 | Data1 [B1] | uint16\_t  int16\_t | Depends on the variable | Example data |
| 6 | Data1 [B0] |
| 7 | Data2 [B3] | uint32\_t  int32\_t  float | Depends on the variable | Example data |
| 8 | Data2 [B2] |
| 9 | Data2 [B1] |
| 10 | Data2 [B0] |
| … | … |  |  | … |
| n-2 | CRC [B1] | uint16\_t | 0x0000 – 0xFFFF | CRC16 checksum |
| n-1 | CRC [B0] |

Ramka zawsze zaczyna się od bajtu Header. Następnie wysyłany jest bajt Frame oznaczający rodzaj ramki. Dalej transmitowana jest długość ramki, która może wynosić od 6 do 60000 bajtów. Dane wielobajtowe transmitowane są począwszy od najstarszego bajtu. Liczby typu float mają rozmiar 4 bajtów i transmitowane sa począwszy od najstarszego bajtu. Ramka kończy się sumą kontrolną CRC16 obliczoną z bajtów o numerach od 0 do n-3 (wszystkie bajty poza bajtami sumy kontrolnej). Poniżej przedstawiono funkcję obliczająca sumę kontrolną.

##### uint16\_t Com\_Crc16(uint8\_t\* packet, uint32\_t nBytes)

##### {

##### uint16\_t crc = 0;

##### for(uint32\_t byte = 0; byte < nBytes; byte++)

##### {

##### crc = crc ^ ((uint16\_t)packet[byte] << 8);

##### for (uint8\_t bit = 0; bit < 8; bit++)

##### if(crc & 0x8000) crc = (crc << 1) ^ 0x1021;

##### else crc = crc << 1;

##### }

##### return crc;

##### }

## Ramki komunikacyjne JTC – Host

### Rodzaje ramek

W poniższej tabeli przedstawiono podstawowe informacje o wszystkich przesyłanych ramkach, takie jak: nazwa, numer, znaczenie, długość, kierunek przesyłania.

#### Tabela 5.3 Types of frames

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nazwa** | **Numer (Frame)** | **Znaczenie** | **Długość**  **(n)** | **Kierunek transmisji** |
| Host\_FT\_null | 0x00 | Ramka pusta, nie wysyłać | - | - |
| Host\_FT\_ClearCurrentErrors | 0x01 | Komenda kasowania bieżących błędów w JTC | 6 | Host -> JTC |
| Host\_FT\_ClearOccuredErrors | 0x02 | Komenda kasowania wszystkich błędów w JTC | 6 | Host -> JTC |
| Host\_FT\_JtcStatus | 0x03 | Dane telemetryczne z JTC | 164 | JTC -> Host |
| Host\_FT\_Trajectory | 0x04 | Nowa trajektoria dla JTC | 14 + 36 \* x  x – liczba punktów w trajektorii | Host -> JTC |
| Host\_FT\_FrictionTable | 0x05 | Nowe tablice wsp. tarcia dla 6 jointów | 10566 | Host -> JTC |
| Host\_FT\_FrictionTableUseDefault | 0x06 | Komenda użycia domyślnych tablic wsp. tarcia dla 6 jointów | 6 | Host -> JTC |
| Host\_FT\_PidParam | 0x07 | Nowe nastawy PID dla 6 jointów | 126 | Host -> JTC |
| Host\_FT\_PidParamUseDefault | 0x08 | Komenda użycia domyślnych nastaw PID dla 6 jointów | 6 | Host -> JTC |
| Host\_FT\_ArmModel | 0x09 | Nowe wartości parametrów dynamicznych i kinematycznych manipulatora | 538 | Host -> JTC |
| Host\_FT\_ArmModelUseDefault | 0x0A | Komenda użycia domyślnych parametrów dynamicznych i kinematycznych manipulatora | 6 | Host -> JTC |
| Host\_FT\_TrajSetExecStatus | 0x0B | Komenda zmiany stanu trajektorii (Stop, Pause, Execute) | 7 | Host -> JTC |

### Host\_FT\_ClearCurrentErrors

Ramka komendy wysyłana z układu hosta do JTC. Długość 6 bajtów. Używana do wyczyszczenia flag bieżących błędów w JTC.

#### Tabela 5.4 Host\_FT\_ClearCurrentErrors

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x01 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 | Length frame in bytes |
| 3 | n [B0] |
| 4 | CRC [B1] | uint16\_t | 0x8F76 | CRC16 checksum |
| 5 | CRC [B0] |

### Host\_FT\_ClearOccuredErrors

Ramka komendy wysyłana z układu hosta do JTC. Używana do wyczyszczenia flag wszystkich błędów w JTC.

#### Tabela 5.5 Host\_FT\_ClearOccuredErrors

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x02 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 | Length frame in bytes |
| 3 | n [B0] |
| 4 | CRC [B1] | uint16\_t | 0xD626 | CRC16 checksum |
| 5 | CRC [B0] |

### Host\_FT\_JtcStatus

Ramka wysyłana z JTC do układu hosta. Ramka zawiera potwierdzenie dotyczące ostatnio odebranej komendy z hosta oraz dane telemetryczne. Pierwsze 5 bajtów to odpowiedź JTC na ostatnio odebraną z hosta ramkę. Odpowiedź ta zawiera cztery pola:

* Frame Type – numer odebranej przez JTC ramki, której dotyczy odpowiedź,
* Status – status poprawności ramki odebranej przez JTC ramki, której dotyczy odpowiedź,
* Data Status – status poprawności danych w ramce odebranej przez JTC ramki, której dotyczy odpowiedź,
* Length – długość odebranej przez JTC ramki, której dotyczy odpowiedź.

Następnie wysyłane są dane telemetryczne JTC:

* JTC Current FSM – aktualny status całego JTC,
* JTC Current Errors – aktualne flagi błędów JTC: 0 – brak błędu, 1 – błąd,
* JTC Occured Errors – flagi błędów JTC, które wystąpiły: 0 – brak błędu, 1 – błąd,
* JTC Init Status – flagi inicjalizacji JTC: 0 – zainicjalizowany, 1 - niezainicjalizowany
* Joints Init Status – flagi inicjalizacji jontów: 0 – zainicjalizowany, 1 – niezainicjalizowany,
* Traj Execution Status – aktualny status realizacji trajektorii,
* Traj Num Current Point – numer aktualnie realizowanego punktu trajektorii (dotyczy trajektorii z krokiem 1ms).

Następnie wysyłane są dane telemetryczne dotyczące komunikacji poprzez CAN:

* CAN Status – aktualny status CAN,
* CAN Current Errors – aktualne flagi błędów CAN: 0 – brak błędu, 1 – błąd,
* CAN Occured Errors – aktualne flagi błędów CAN: 0 – brak błędu, 1 – błąd,

Następnie wysyłane są dane telemetryczne dotyczące napędów:

* Joint Current FSM – aktualny status napędu,
* Joint mcCurrentError – dane odebrane z napędu poprzez CAN, szczegóły w opisie pracy napędu,
* Joint mcOccuredError – dane odebrane z napędu poprzez CAN, szczegóły w opisie pracy napędu,
* Joint currentError – dane odebrane z napędu poprzez CAN, szczegóły w opisie pracy napędu,
* Joint currentWarning – dane odebrane z napędu poprzez CAN, szczegóły w opisie pracy napędu,
* Joint Internall Current Errors – aktualne błędy pracy napędu powstały w JTC,
* Joint Internall Occured Errors –błędy pracy napędu powstały w JTC,
* Joint Current Position – dane odebrane z napędu poprzez CAN,
* Joint Current Velocity – dane odebrane z napędu poprzez CAN,
* Joint Current Torque – dane odebrane z napędu poprzez CAN,
* Joint Current Temp – dane odebrane z napędu poprzez CAN,

#### Tabela 5.6 Host\_FT\_JtcStatus

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** |
| 0 | Header | uint8\_t | 0x9B |
| 1 | Frame | uint8\_t | 0x03 |
| 2 | n [B1] | uint16\_t | 0x00A4 |
| 3 | n [B0] |
| 4 | Response Frame Type [B0] | uint8\_t | 0x00 – 0x0B |
| 5 | Response Status [B0] | uint8\_t | 0x00 – Idle  0x01 – No Error  0x02 – Incorrect Header  0x03 – Incorrect FrameType  0x04 – Incorrect CRC  0x05 – Discontinuous Frame |
| 6 | Response Data Status [B0] | uint8\_t | 0x00 – Idle  0x01 – No Error  0x02 – Traj Too Many Points  0x03 – Traj Too Many Segs  0x04 – Traj Incorrect Seg Order  0x05 – Traj Incorrect Step Time |
| 7 - 8 | Response Length [B1 - B0] | uint16\_t | 0x0006 – 0xEA60 |
| 9 | JTC Current FSM [B0] | uint8\_t | 0x00 – JTC\_FSM\_Start  0x01 – JTC\_FSM\_Init  0x02 – JTC\_FSM\_Operate  0x03 – JTC\_FSM\_Error |
| 10 – 11 | JTC Current Errors [B1 – B0] | uint16\_t | B0.0 – Emergency Input Status  B0.1 – Emergency Output Status  B0.2 – Internall Error  B0.3 – Externall Error  B0.4 – Internall Joints Error  B0.5 – Internall CAN Error  B0.6 – Internall COM Error  B0.7 – Externall Joints Error |
| 12 – 13 | JTC Occured Errors [B1 – B0] | uint16\_t | B0.0 – Emergency Input Status  B0.1 – Emergency Output Status  B0.2 – Internall Error  B0.3 – Externall Error  B0.4 – Internall Joints Error  B0.5 – Internall CAN Error  B0.6 – Internall COM Error  B0.7 – Externall Joints Error |
| 14 | JTC Init Status [B0] | uint8\_t | B0.0 – Friction Table Init Status  B0.1 – Pid Parameters Init Status  B0.2 – Arm Model Init Status |
| 15 | Joints Init Status [B0] | uint8\_t | B0.0 – Joint 0 Init Status  B0.1 – Joint 1 Init Status  B0.2 – Joint 2 Init Status  B0.3 – Joint 3 Init Status  B0.4 – Joint 4 Init Status  B0.5 – Joint 5 Init Status |
| 16 | Traj Execution Status [B0] | uint8\_t | 0x00 – TES\_Null  0x01 – TES\_Stop  0x02 – TES\_Pause  0x03 – TES\_Execute  0x04 – TES\_Finish  0x05 – TES\_TransNullToStop |
| 17 – 20 | Traj Num Curr. Point [B3 – B0] | Uint32\_t | 0x00000000 – 0x0001D4C0 |
| 21 | CAN Current Status [B0] | uint8\_t | 0x00 – CAN\_NoError  0x01 – CAN\_Error |
| 22 – 25 | CAN Current Errors [B3 – B0] | uint32\_t | B0.0 – Transmit Timeout  B1.0 – Joints 0 Receive Timeout  B1.1 – Joints 0 Receive Timeout  B1.2 – Joints 0 Receive Timeout  B1.3 – Joints 0 Receive Timeout  B1.4 – Joints 0 Receive Timeout  B1.5 – Joints 0 Receive Timeout |
| 26 – 29 | CAN Occured Errors [B3 – B0] | uint32\_t | B0.0 – Transmit Timeout  B1.0 – Joints 0 Receive Timeout  B1.1 – Joints 0 Receive Timeout  B1.2 – Joints 0 Receive Timeout  B1.3 – Joints 0 Receive Timeout  B1.4 – Joints 0 Receive Timeout  B1.5 – Joints 0 Receive Timeout |
| 30 | Joint 0 Current FSM [B0] | uint8\_t | 0x00 – FSM\_Start  0x01 – FSM\_Init  0x02 – FSM\_ReadyToOperate  0x03 – FSM\_OperationEnable |
| 31 | Joint 0 MC Current Errors [B0] | uint8\_t | Value from CAN. See joint description |
| 32 | Joint 0 MC Occured Errors [B0] | uint8\_t | Value from CAN. See joint description |
| 33 | Joint 0 Current Errors [B0] | uint8\_t | Value from CAN. See joint description |
| 34 | Joint 0 Current Warnings [B0] | uint8\_t | Value from CAN. See joint description |
| 35 – 36 | Joint 0 Internall Errors [B1 – B0] | uint16\_t | B0.0 – Calculated pos over limit  B0.1 – Calculated vel over limit  B0.2 – Calculated acc over limit  B0.3 – Calculated torque over limit  B0.4 – Position error over limit  B0.5 – Friction table value over limit |
| 37 – 38 | Joint 0 Internall Occured Errors [B1 – B0] | uint16\_t | B0.0 – Calculated pos over limit  B0.1 – Calculated vel over limit  B0.2 – Calculated acc over limit  B0.3 – Calculated torque over limit  B0.4 – Position error over limit  B0.5 – Fric table value over limit |
| 39 – 42 | Joint 0 Current Position [B3 – B0] | float | Value from CAN. See joint description |
| 43 – 46 | Joint 0 Current Velocity [B3 – B0] | float | Value from CAN. See joint description |
| 47 – 50 | Joint 0 Current Torque [B3 – B0] | float | Value from CAN. See joint description |
| 51 | Joint 0 Current Temperature [B0] | uint8\_t | Value from CAN. See joint description |
| 53 | Joint 1 Current FSM [B0] | uint8\_t | 0x00 – FSM\_Start  0x01 – FSM\_Init  0x02 – FSM\_ReadyToOperate  0x03 – FSM\_OperationEnable |
| … | … | … | … |
| 161 | Joint 5 Current Temperature [B0] | uint8\_t | Value from CAN. See joint description |
| 162 | CRC [B1] | uint16\_t | - |
| 163 | CRC [B0] |

Funkcja wysyłająca dane z JTC:

##### static void Host\_ComPrepareFrameJtcStatus(void)

##### {

##### uint8\_t \*buf = Com.txFrames[Host\_TxFN\_JtcStatus].frame;

##### uint8\_t idx = 0;

##### buf[idx++] = (uint8\_t)Host\_FT\_Header;

##### buf[idx++] = (uint8\_t)Host\_FT\_JtcStatus;

##### buf[idx++] = (uint8\_t)(0 >> 8); // Space for the number of bytes in the frame

##### buf[idx++] = (uint8\_t)(0 >> 0); // Space for the number of bytes in the frame

##### buf[idx++] = (uint8\_t)Com.rxFrame.frameType;

##### buf[idx++] = (uint8\_t)Com.rxFrame.status;

##### buf[idx++] = (uint8\_t)Com.rxFrame.dataStatus;

##### buf[idx++] = (uint8\_t)(Com.rxFrame.receivedLength>>8);

##### buf[idx++] = (uint8\_t)(Com.rxFrame.receivedLength>>0);

##### 

##### buf[idx++] = (uint8\_t)pC->Jtc.currentFsm;

##### buf[idx++] = (uint8\_t)(pC->Jtc.errors >> 8);

##### buf[idx++] = (uint8\_t)(pC->Jtc.errors >> 0);

##### buf[idx++] = (uint8\_t)(pC->Jtc.occuredErrors >> 8);

##### buf[idx++] = (uint8\_t)(pC->Jtc.occuredErrors >> 0);

##### buf[idx++] = (uint8\_t)pC->Jtc.jtcInitStatus;

##### buf[idx++] = (uint8\_t)pC->Jtc.jointsInitStatus;

##### buf[idx++] = (uint8\_t)Traj.currentTES;

##### buf[idx++] = (uint8\_t)(Traj.numInterPoint >> 8);

##### buf[idx++] = (uint8\_t)(Traj.numInterPoint >> 0);

##### 

##### buf[idx++] = (uint8\_t)pC->Can.statusId;

##### buf[idx++] = (uint8\_t)(pC->Can.statusFlags >> 24);

##### buf[idx++] = (uint8\_t)(pC->Can.statusFlags >> 16);

##### buf[idx++] = (uint8\_t)(pC->Can.statusFlags >> 8);

##### buf[idx++] = (uint8\_t)(pC->Can.statusFlags >> 0);

##### buf[idx++] = (uint8\_t)(pC->Can.statusOccurredFlags >> 24);

##### buf[idx++] = (uint8\_t)(pC->Can.statusOccurredFlags >> 16);

##### buf[idx++] = (uint8\_t)(pC->Can.statusOccurredFlags >> 8);

##### buf[idx++] = (uint8\_t)(pC->Can.statusOccurredFlags >> 0);

##### 

##### union conv32 x;

##### for(int num=0;num<JOINTS\_MAX;num++)

##### {

##### buf[idx++] = (uint8\_t)(pC->Joints[num].currentFsm >> 0);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].mcCurrentError >> 0);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].mcOccuredError >> 0);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].currentError >> 0);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].currentWarning >> 0);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].internallErrors >> 8);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].internallErrors >> 0);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].internallOccuredErrors >> 8);

##### buf[idx++] = (uint8\_t)(pC->Joints[num].internallOccuredErrors >> 0);

##### 

##### x.f32 = pC->Joints[num].currentPos;

##### buf[idx++] = (uint8\_t)(x.u32 >> 24);

##### buf[idx++] = (uint8\_t)(x.u32 >> 16);

##### buf[idx++] = (uint8\_t)(x.u32 >> 8);

##### buf[idx++] = (uint8\_t)(x.u32 >> 0);

##### 

##### x.f32 = pC->Joints[num].currentVel;

##### buf[idx++] = (uint8\_t)(x.u32 >> 24);

##### buf[idx++] = (uint8\_t)(x.u32 >> 16);

##### buf[idx++] = (uint8\_t)(x.u32 >> 8);

##### buf[idx++] = (uint8\_t)(x.u32 >> 0);

##### 

##### x.f32 = pC->Joints[num].currentTorque;

##### buf[idx++] = (uint8\_t)(x.u32 >> 24);

##### buf[idx++] = (uint8\_t)(x.u32 >> 16);

##### buf[idx++] = (uint8\_t)(x.u32 >> 8);

##### buf[idx++] = (uint8\_t)(x.u32 >> 0);

##### 

##### buf[idx++] = (uint8\_t)pC->Joints[num].currentTemp;

##### }

##### // Number of bytes in frame and CRC

##### buf[2] = (uint8\_t)((idx + 2) >> 8);

##### buf[3] = (uint8\_t)((idx + 2) >> 0);

##### uint16\_t crc = Com\_Crc16(buf, idx);

##### buf[idx++] = (uint8\_t)(crc >> 8); // CRC

##### buf[idx++] = (uint8\_t)(crc >> 0); // CRC

##### Com.txFrames[Host\_TxFN\_JtcStatus].status = Host\_TxFS\_ReadyToSend;

##### Com.txFrames[Host\_TxFN\_JtcStatus].len = idx;

##### Com.txFrames[Host\_TxFN\_JtcStatus].active = false;

##### }

### Host\_FT\_Trajectory

Ramka wysyłana z układu hosta do JTC. Używana do wysłania nowej trajektorii. Maksymalna długość trajektorii to 12000 punktów. Każdy punkt składa się z sześciu wartości pozycji kątowej, sześciu wartości prędkości kątowej i sześciu wartości przyspieszenia kątowego (razem 18 liczb typu int16\_t – 36 bajtów). Dane przesyłane są w formie liczb typu int16\_t. Wyznaczenie wartości w formacie int16\_t należy wykonać według poniższych wzorów:

* pos(int16\_t) = pos(float) / 3.141592 \* 32767.0
* vel(int16\_t) = vel(float) / 6.283185 \* 32767.0
* acc(int16\_t) = acc(float) / 12.566370 \* 32767.0

Trajektoria musi być przesyłana w segmentach nie dłuższych niż 1500 punktów. Każdy segment wysyłany jest jako osobna ramka. JTC każdorazowo potwierdza odebranie ramki danych wysyłając informację o jej poprawności lub błędach. Trajektoria musi się składać z co najmniej jednego segmentu. Liczba segmentów nie może przekraczać 100. Segmenty muszą być wysyłane w kolejności od segmentu numer 0 do segmentu numer x-1 (x = liczba segmentów). Poprawne przesłanie segmentu numer 0 powoduje wyczyszczenie bieżącej trajektorii i przejście JTC w status TES\_Null. Po odebraniu wszystkich segmentów JTC przechodzi w status TES\_Stop. Następnie można uruchomić trajektorię. W przypadku błędnego przesłania danego segmentu należy ponowić jego wysyłanie. W przypadku przesyłania segmentów w błędnej kolejności należy rozpocząć ponownie przesyłanie całej trajektorii od segmentu numer 0. Host może w każdej chwili przerwać przesyłanie trajektorii i rozpocząć jej transmisje od początku. Poniższa tabela przedstawia ramkę przesyłająca jeden segment.

#### Tabela 5.7 Host\_FT\_Trajectory

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x04 | Type of frame |
| 2 | n [B1] | uint16\_t | - | Length frame in bytes |
| 3 | n [B0] |
| 4 - 5 | Trajectory number | uint16\_t | 0x00 | Data |
| 6 - 7 | Segment number | uint16\_t | 0x0000 – 0x0063 | Data |
| 8 - 9 | Number of segments | uint16\_t | 0x0001 – 0x0064 | Data |
| 10 - 11 | Step time | uint16\_t | 0x0001 – 0xFFFF | Data |
| 12 - 13 | Point 0 Joint 0 Pos [B1 – B0] | int16\_t | - | Data |
| … | … | … | … | … |
|  | Point 0 Joint 5 Pos [B1 – B0] | int16\_t | - | Data |
|  | Point 0 Joint 0 Vel [B1 – B0] | int16\_t | - | Data |
| … | … | … | … | … |
|  | Point 0 Joint 5 Vel [B1 – B0] | int16\_t | - | Data |
|  | Point 0 Joint 0 Acc [B1 – B0] | int16\_t | - | Data |
| … | … | … | … | … |
|  | Point 0 Joint 5 Acc [B1 – B0] | int16\_t | - | Data |
|  | Point 1 Joint 0 Pos [B1 – B0] | int16\_t | - | Data |
| … | … | … | … | … |
|  | Point k-1 Joint 5 Acc [B1 – B0] | int16\_t | - | Data |
|  | CRC [B1] | uint16\_t | - | CRC16 checksum |
|  | CRC [B0] |

Funkcja odbierająca dane w JTC:

##### static void Host\_ComReadFrameTrajectory(uint8\_t\* buf)

##### {

##### uint16\_t idx = 4, np;

##### uint16\_t nd = Com.rxFrame.expectedLength;

##### uint16\_t crc1 = Com\_Crc16(buf, nd-2);

##### uint16\_t crc2 = ((uint16\_t)buf[nd-2]<<8) + ((uint16\_t)buf[nd-1]<<0);

##### if(crc1 == crc2)

##### {

##### Com.timeout = 0;

##### uint16\_t trajNum = ((uint16\_t)buf[idx++]<<8);

##### trajNum += ((uint16\_t)buf[idx++]<<0);

##### uint16\_t segNum = ((uint16\_t)buf[idx++]<<8);

##### segNum += ((uint16\_t)buf[idx++]<<0);

##### uint16\_t segMax = ((uint16\_t)buf[idx++]<<8);

##### segMax += ((uint16\_t)buf[idx++]<<0);

##### uint16\_t stepTime = ((uint16\_t)buf[idx++]<<8);

##### stepTime += ((uint16\_t)buf[idx++]<<0);

##### 

##### // liczba punktow w odebranym segmencie

##### np = (nd - 14) / (3 \* JOINTS\_MAX \* 2);

##### // odebrano juz zbyt wiele punktów w tej trajektorii

##### if((Traj.numRecPoints + np) > TRAJ\_POINTSMAX)

##### {

##### Com.rxFrame.dataStatus = Host\_RxDS\_TrajTooManyPoints;

##### return;

##### }

##### // odebrano juz zbyt wiele segmentow w tej trajektorii

##### if(segNum > TRAJ\_SEGSSMAX || segMax > TRAJ\_SEGSSMAX)

##### {

##### Com.rxFrame.dataStatus = Host\_RxDS\_TrajTooManySegs;

##### return;

##### }

##### // odebrano segment o zlym numerze (np: zla kolejnosc transmitowanych segmentow)

##### if(segNum != 0 && (((int16\_t)(segNum - Traj.numSeg) != 1) || (segNum >= segMax)))

##### {

##### Com.rxFrame.dataStatus = Host\_RxDS\_TrajIncorrectSegOrder;

##### return;

##### }

##### // odebrano niepoprawny krok czasowy (np: stepTime = 0)

##### if(stepTime == 0)

##### {

##### Com.rxFrame.dataStatus = Host\_RxDS\_TrajIncorrectStepTime;

##### return;

##### }

##### // odebrane dane sa poprawne

##### Com.rxFrame.dataStatus = Host\_RxDS\_NoError;

##### // segment o numerze 0 oznacza nowa trajektorie, poprzednia jest czyszczona

##### if(segNum == 0)

##### Control\_TrajClear();

##### // sprawdzenie czy odebrany segment jest ostatnim w trajektorii

##### if(segNum == (segMax-1))

##### {

##### Traj.comStatus = TCS\_WasRead; // last segment

##### Traj.targetTES = TES\_Stop;

##### }

##### else

##### Traj.comStatus = TCS\_IsRead;

##### Traj.numTraj = trajNum;

##### Traj.numSeg = segNum;

##### Traj.maxSeg = segMax;

##### Traj.stepTime = stepTime;

##### Traj.flagReadSeg[Traj.numSeg] = true;

##### Traj.numPointsSeg[Traj.numSeg] = np;

##### for(uint16\_t i=Traj.numRecPoints;i<Traj.numRecPoints+np;i++)

##### {

##### for(uint16\_t j=0;j<JOINTS\_MAX;j++)

##### {

##### Traj.points[i].pos[j] = ((uint16\_t)buf[idx++]<<8);

##### Traj.points[i].pos[j] += ((uint16\_t)buf[idx++]<<0);

##### }

##### for(uint16\_t j=0;j<JOINTS\_MAX;j++)

##### {

##### Traj.points[i].vel[j] = ((uint16\_t)buf[idx++]<<8);

##### Traj.points[i].vel[j] += ((uint16\_t)buf[idx++]<<0);

##### }

##### for(uint16\_t j=0;j<JOINTS\_MAX;j++)

##### {

##### Traj.points[i].acc[j] = ((uint16\_t)buf[idx++]<<8);

##### Traj.points[i].acc[j] += ((uint16\_t)buf[idx++]<<0);

##### }

##### }

##### Traj.numRecPoints += np;

##### }

##### else

##### {

##### Com.rxFrame.status = Host\_RxFS\_ErrorIncorrectCrc;

##### }

##### }

### Host\_FT\_FrictionTable

Ramka wysyłana z układu hosta do JTC. Używana do ustawienia nowych wartości tablic współczynników kompensacji tarcia. Pole danych zawiera wartości dla sześciu napędów. Wartości wysyłane są począwszy od napędu numer 0. Rozmiar tablicy dla jednego napędu wynosi 22 wiersze i 20 kolumn. Pierwszy wiersz zawiera wartości prędkości w rad/s. Drugi wiersz zawiera wartości temperatury w stopniach Celsjusza. Następne wiersze zawierają wartości momentu tarcia dla danej prędkości i temperatury. Wszystkie dane w tablicy są zapisane w formacie float.

#### Tabela 5.8 Host\_FT\_FrictionTable

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x05 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x2946 | Length frame in bytes |
| 3 | n [B0] |
| 4 - 7 | Joint 0 Vel 0 [B3 – B0] | float | - | Data |
| … | … | … | … | … |
|  | Joint 0 Vel 19 [B3 – B0] | float | - | Data |
|  | Joint 0 Temp 0 [B3 – B0] | float | - | Data |
| … | … | … | … | … |
|  | Joint 0 Temp 19 [B3 – B0] | float | - | Data |
|  | Joint 0 Fric 0,0 [B3 – B0] | float | - | Data |
|  | … | … | … | … |
|  | Joint 0 Fric 0,19 [B3 – B0] | float | - | Data |
|  | Joint 0 Fric 1,0 [B3 – B0] | float | - | Data |
|  | … | … | … | … |
|  | Joint 0 Fric 19,19 [B3 – B0] | float | - | Data |
|  | Joint 1 Vel 0 [B3 – B0] | float | - | Data |
| … | … | … | … | … |
| 10560 - 10563 | Joint 5 Fric 19,19 [B3 – B0] | float | - | Data |
| 10564 | CRC [B1] | uint16\_t | - | CRC16 checksum |
| 10565 | CRC [B0] |

Funkcja odbierająca dane w JTC:

##### static void Host\_ComReadFrameFricionTable(uint8\_t\* buf)

##### {

##### uint16\_t nd = Com.rxFrame.expectedLength;

##### uint16\_t crc1 = Com\_Crc16(buf, nd-2);

##### uint16\_t crc2 = ((uint16\_t)buf[nd-2]<<8) + ((uint16\_t)buf[nd-1]<<0);

##### uint16\_t idx = 4;

##### if(crc1 == crc2)

##### {

##### Com.timeout = 0;

##### // the received data is correct

##### Com.rxFrame.dataStatus = Host\_RxDS\_NoError;

##### union conv32 x;

##### for(int num=0;num<JOINTS\_MAX;num++)

##### {

##### for(int i=0;i<JOINTS\_FRICTABVELSIZE;i++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[num].fricTableVelIdx[i] = x.f32;

##### }

##### for(int i=0;i<JOINTS\_FRICTABTEMPSIZE;i++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[num].fricTableTempIdx[i] = x.f32;

##### }

##### for(int i=0;i<JOINTS\_FRICTABVELSIZE;i++)

##### {

##### for(int j=0;j<JOINTS\_FRICTABTEMPSIZE;j++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[num].fricTable[i][j] = x.f32;

##### }

##### }

##### }

##### Joints\_FindMinMaxVelTempInFrictionTabeIdx();

##### pC->Jtc.flagInitGetFrictionTable = false;

##### }

##### else

##### {

##### Com.rxFrame.status = Host\_RxFS\_ErrorIncorrectCrc;

##### }

##### }

### Host\_FT\_FrictionTableUseDefault

Ramka komendy wysyłana z układu hosta do JTC. Używana do przywrócenia domyślnych wartości tablic współczynników kompensacji tarcia.

#### Tabela 5.9 Host\_FT\_FrictionTableUseDefault

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x06 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 | Length frame in bytes |
| 3 | n [B0] |
| 4 | CRC [B1] | uint16\_t | 0x0AE6 | CRC16 checksum |
| 5 | CRC [B0] |

### Host\_FT\_PidParam

Ramka wysyłana z układu hosta do JTC. Używana do ustawienia nowych wartości nastaw regulatorów PID. Pole danych zawiera wartości dla sześciu napędów. Wartości wysyłane są począwszy od napędu numer 0. Dla każdego napędu przesyłanych jest 5 wartości liczbowych typu float (20 bajtów). Wartości to:

* wzmocnienie członu proporcjonalnego pidKp,
* wzmocnienie członu inercyjnego pidKi,
* wzmocnienie członu różniczkującego pidKd,
* saturacja całki uchybu – wartość minimalna pidErrorIntMin,
* saturacja całki uchybu – wartość maksymalna pidErrorIntMax

#### Tabela 5.10 Host\_FT\_PidParam

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x07 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x7E | Length frame in bytes |
| 3 | n [B0] |
| 4 - 7 | Joint 0 pidKp [B3 – B0] | float | - | Data |
| 8 - 11 | Joint 0 pidKi [B3 – B0] | float | - | Data |
| 12 - 15 | Joint 0 pidKd [B3 – B0] | float | - | Data |
| 16 - 19 | Joint 0 pidErrorIntMin [B3 – B0] | float | - | Data |
| 20 – 23 | Joint 0 pidErrorIntMax [B3 – B0] | float | - | Data |
| 24 – 27 | Joint 1 pidKp [B3 – B0] | float | - | Data |
| … | … | … | … | … |
| 120 - 123 | Joint 5 pidErrorIntMax [B3 – B0] | float | - | Data |
| 124 | CRC [B1] | uint16\_t | - | CRC16 checksum |
| 125 | CRC [B0] |

Funkcja odbierająca dane w JTC:

##### static void Host\_ComReadFramePidParam(uint8\_t\* buf)

##### {

##### uint16\_t nd = Com.rxFrame.expectedLength; //JOINTS\_MAX \* 7 \* 4 + 4;

##### uint16\_t crc1 = Com\_Crc16(buf, nd-2);

##### uint16\_t crc2 = ((uint16\_t)buf[nd-2]<<8) + ((uint16\_t)buf[nd-1]<<0);

##### uint16\_t idx = 4;

##### if(crc1 == crc2)

##### {

##### Com.timeout = 0;

##### // the received data is correct

##### Com.rxFrame.dataStatus = Host\_RxDS\_NoError;

##### Joints\_SetDefaultVariables();

##### union conv32 x;

##### for(int i=0;i<JOINTS\_MAX;i++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[i].pidKp = x.f32;

##### 

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[i].pidKi = x.f32;

##### 

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[i].pidKd = x.f32;

##### 

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[i].pidErrorIntMin = x.f32;

##### 

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Joints[i].pidErrorIntMax = x.f32;

##### }

##### 

##### pC->Jtc.flagInitGetPidParam = false;

##### }

##### else

##### {

##### Com.rxFrame.status = Host\_RxFS\_ErrorIncorrectCrc;

##### }

##### }

### Host\_FT\_PidParamUseDefault

Ramka komendy wysyłana z układu hosta do JTC. Używana do przywrócenia domyślnych wartości nastaw regulatorów PID.

#### Tabela 5.11 Host\_FT\_PidParamUseDefault

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x08 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 | Length frame in bytes |
| 3 | n [B0] |
| 4 | CRC [B1] | uint16\_t | 0x11E7 | CRC16 checksum |
| 5 | CRC [B0] |

### Host\_FT\_ArmModel

Ramka wysyłana z układu hosta do JTC. Używana do ustawienia nowych wartości parametrów kinematycznych i dynamicznych manipulatora. Pole danych zawiera wartości w formacie float (4 bajty). Na początek wysyłane są parametry układów współrzędnych dla jointów. Następnie wysyłane są parametry układów współrzędnych, momentów bezwładności i mas dla linków.

#### Tabela 5.12 Host\_FT\_ArmModel

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x09 | Type of frame |
| 2 | n [B1] | uint16\_t | 0x021A | Length frame |
| 3 | n [B0] |
| 4 - 7 | Joint 0 Origin X [B3 – B0] | float | - | Data |
| 8 - 11 | Joint 0 Origin Y [B3 – B0] | float | - | Data |
| 12 - 15 | Joint 0 Origin Z [B3 – B0] | float | - | Data |
| 16 - 19 | Joint 0 Origin Roll [B3 – B0] | float | - | Data |
| 20 – 23 | Joint 0 Origin Pitch [B3 – B0] | float | - | Data |
| 24 – 27 | Joint 0 Origin Yaw [B3 – B0] | float | - | Data |
| 28 – 31 | Joint 1 Origin X [B3 – B0] | float | - | Data |
| … | … | … | … | … |
|  | Joint 6 Origin Yaw [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Origin X [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Origin Y [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Origin Z [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Origin Roll [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Origin Pitch [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Origin Yaw [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Inertia Ixx [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Inertia Ixy [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Inertia Ixz [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Inertia Iyy [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Inertia Iyz [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Inertia Izz [B3 – B0] | float | - | Data |
|  | Link 0 Inertial Mass [B3 – B0] | float | - | Data |
|  | Link 1 Inertial Origin X [B3 – B0] | float | - | Data |
| … | … | … | … | … |
| 532 - 535 | Link 6 Inertial Mass [B3 – B0] | float | - | Data |
| 536 | CRC [B1] | uint16\_t | - | CRC16 checksum |
| 537 | CRC [B0] |

Funkcja odbierająca dane w JTC:

##### static void Host\_ComReadFrameArmModel(uint8\_t\* buf)

##### {

##### uint16\_t nd = Com.rxFrame.expectedLength;

##### uint16\_t crc1 = Com\_Crc16(buf, nd-2);

##### uint16\_t crc2 = ((uint16\_t)buf[nd-2]<<8) + ((uint16\_t)buf[nd-1]<<0);

##### uint16\_t idx = 4;

##### if(crc1 == crc2)

##### {

##### Com.timeout = 0;

##### // the received data is correct

##### Com.rxFrame.dataStatus = Host\_RxDS\_NoError;

##### union conv32 x;

##### for(int i=0;i<ARMMODEL\_DOF+1;i++)

##### {

##### for(int j=0;j<6;j++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Arm.Joints[i].origin[j] = x.f32;

##### }

##### }

##### for(int i=0;i<ARMMODEL\_DOF+1;i++)

##### {

##### for(int j=0;j<6;j++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Arm.Links[i].origin[j] = x.f32;

##### }

##### for(int j=0;j<6;j++)

##### {

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Arm.Links[i].innertia[j] = x.f32;

##### }

##### x.u32 = ((uint32\_t)buf[idx++]<<24);

##### x.u32 += ((uint32\_t)buf[idx++]<<16);

##### x.u32 += ((uint32\_t)buf[idx++]<<8);

##### x.u32 += ((uint32\_t)buf[idx++]<<0);

##### pC->Arm.Links[i].mass = x.f32;

##### }

##### pC->Jtc.flagInitGetArmModel = false;

##### }

##### else

##### {

##### Com.rxFrame.status = Host\_RxFS\_ErrorIncorrectCrc;

##### }

##### }

### Host\_FT\_ArmModelUseDefault

Ramka komendy wysyłana z układu hosta do JTC. Używana do przywrócenia domyślnych wartości parametrów kinematycznych i dynamicznych manipulatora.

#### Tabela 5.13 Host\_FT\_ArmModelUseDefault

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x0A | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 | Length frame in bytes |
| 3 | n [B0] |
| 4 | CRC [B1] | uint16\_t | 0x7F87 | CRC16 checksum |
| 5 | CRC [B0] |

### Host\_FT\_TrajSetExecStatus

Ramka komendy wysyłana z układu hosta do JTC. Używana do zmiany trybu realizacji trajektorii. Dopuszczalne tryby to: STOP, PAUSE, EXECUTE.

#### Tabela 5.14 Host\_FT\_TrajSetExecStatus

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Byte number** | **Name** | **Format** | **Value** | **Description** |
| 0 | Header | uint8\_t | 0x9B | Header |
| 1 | Frame | uint8\_t | 0x0A | Type of frame |
| 2 | n [B1] | uint16\_t | 0x0006 | Length frame in bytes |
| 3 | n [B0] |
| 4 | CMD | uint8\_t | 0x01 – STOP  0x02 – PAUSE  0x03 – EXECUTE | New trajectory mode |
| 5 | CRC [B1] | uint16\_t | CRC = 0x5DDC (for CMD = 0x01)  CRC = 0x 6DBF (for CMD = 0x02)  CRC = 0x 7D9E (for CMD = 0x03) | CRC16 checksum |
| 6 | CRC [B0] |