1 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):

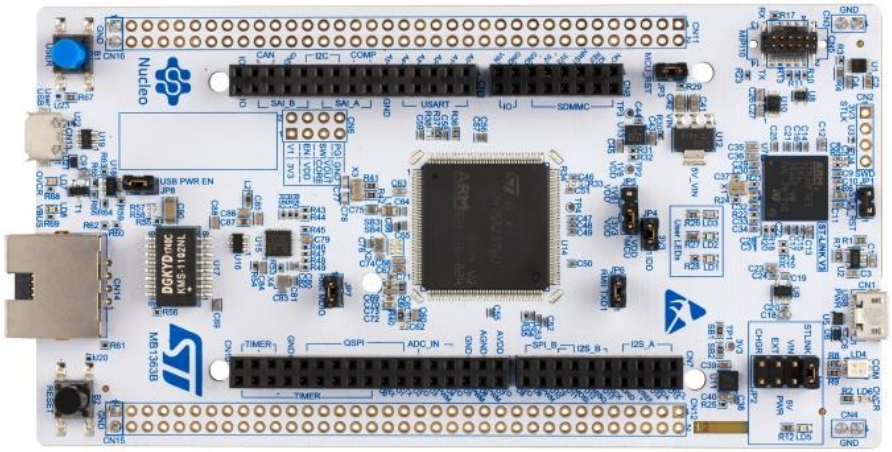
- zdolność do komunikacji z układem hosta poprzez port szeregowy UART i interfejs RS422,
    - odczyt zadanej trajektorii w postaci pozycji, prędkości i przyspieszenia kątowego dla każdego z sześciu napędów
    - odczyt tablic zawierających współczynniki kompensacji tarcia lub odczyt komendy użycia domyślnych 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.

2 Wymagany hardware i software

2.1 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.

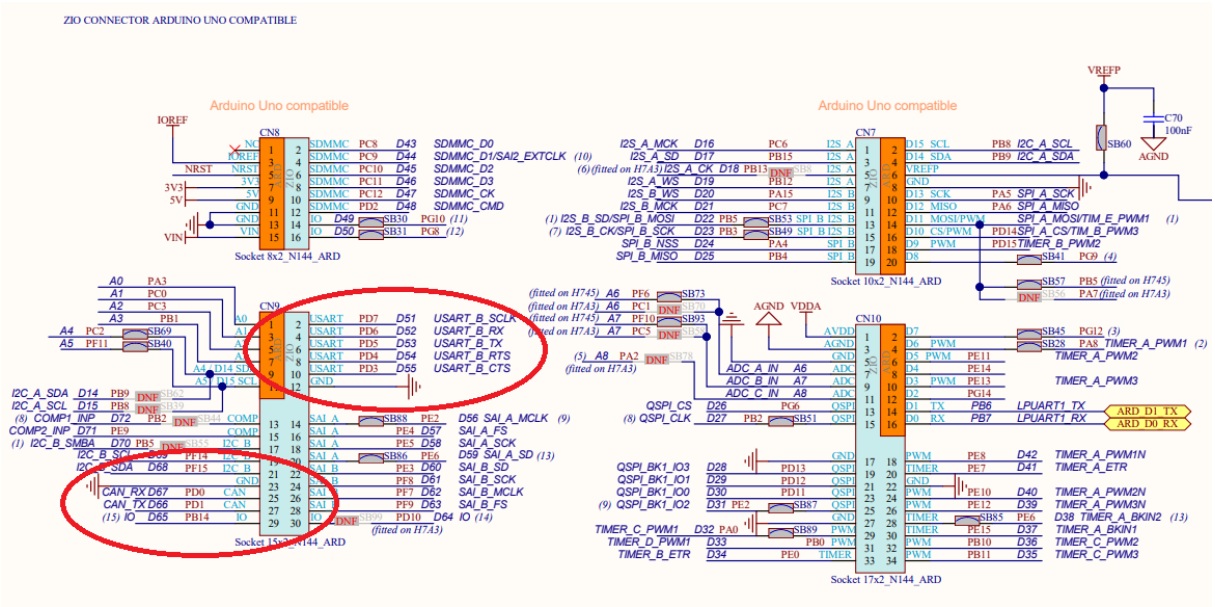


Rys. 2.1 Moduł NUCLEO-H745ZI-Q2

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. 2.2 JTC pinout

2.2 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>
```

3 Logika działania

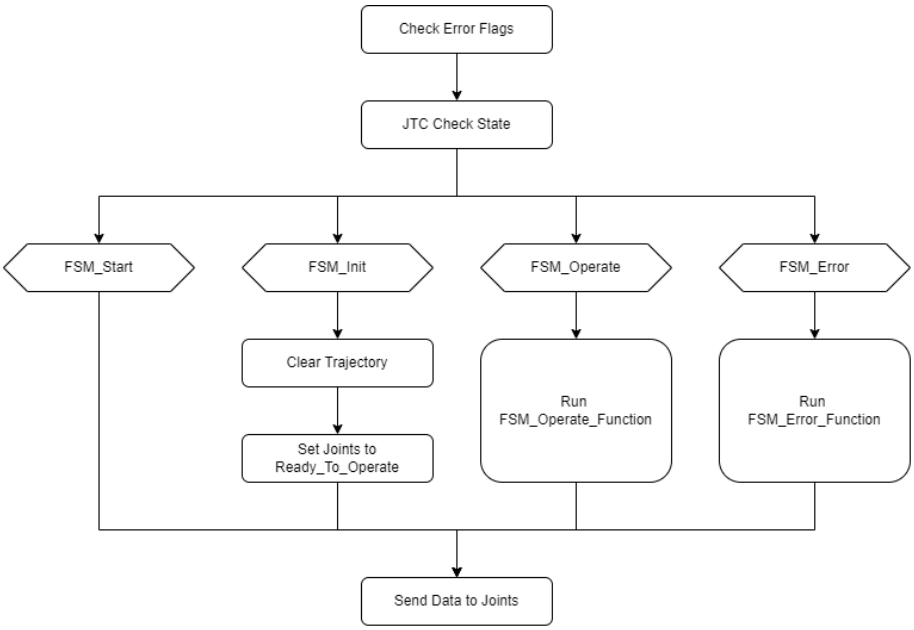
3.1 Ogólny schemat sterowania

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. 3.1 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();
}
```

3.2 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.

3.3 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.

3.4 FSM_Operate

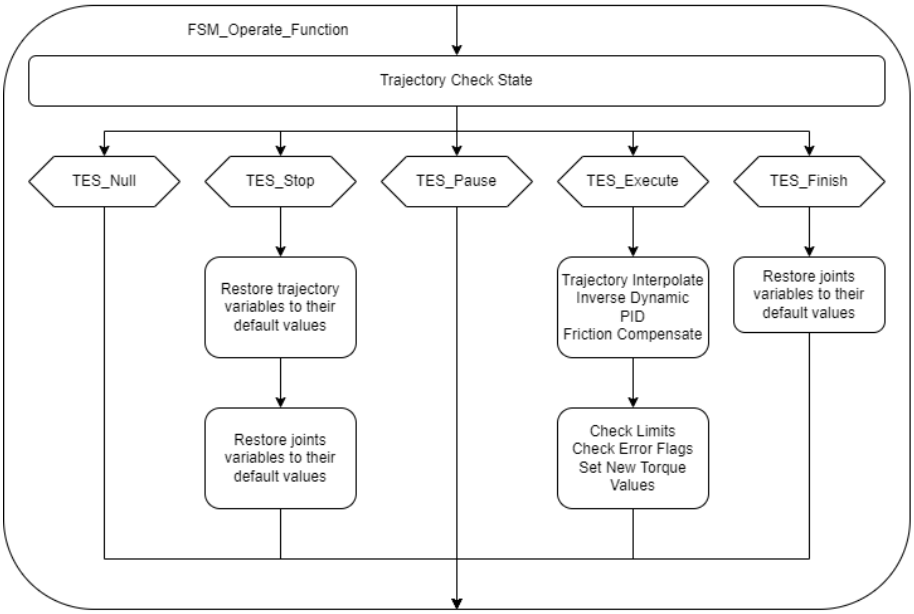
JTC znajduje się w trybie FSM_Operate tylko wtedy gdy: wszystkie napędy zostały poprawnie zainicjalizowane, wartości zmiennych z układu hosta zostały poprawnie zainicjalizowane, nie wystąpił żaden błąd. W tym trybie możliwa jest realizacja trajektorii.

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. 3.2 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.

3.5 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

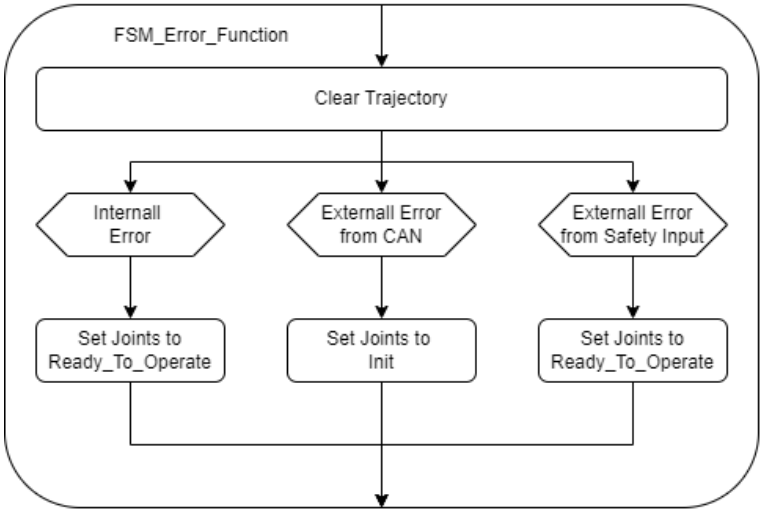
Do błędów wewnętrznych należą:

- 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. 3.3 FSM_Error scheme of operation

4 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

5 Komunikacja JTC – Host

5.1 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

5.2 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;
}
```

5.3 Ramki komunikacyjne JTC – Host

5.3.1 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	162	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

5.3.2 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]			

5.3.3 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]			

5.3.4 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	0x00A2
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 – 18	Traj Num Curr. Point [B1 – B0]	uint16_t	0x0000 – 0x2EDF
19	CAN Current Status [B0]	uint8_t	0x00 – CAN_NoError 0x01 – CAN_Error
20 – 23	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
24 – 27	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
28	Joint 0 Current FSM [B0]	uint8_t	0x00 – FSM_Start 0x01 – FSM_Init 0x02 – FSM_ReadyToOperate 0x03 – FSM_OperationEnable
29	Joint 0 MC Current Errors [B0]	uint8_t	Value from CAN. See joint description
30	Joint 0 MC Occured Errors [B0]	uint8_t	Value from CAN. See joint description
31	Joint 0 Current Errors [B0]	uint8_t	Value from CAN. See joint description
32	Joint 0 Current Warnings [B0]	uint8_t	Value from CAN. See joint description
33 – 34	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

35 – 36	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
37 – 40	Joint 0 Current Position [B3 – B0]	float	Value from CAN. See joint description
41 – 44	Joint 0 Current Velocity [B3 – B0]	float	Value from CAN. See joint description
45 – 48	Joint 0 Current Torque [B3 – B0]	float	Value from CAN. See joint description
49	Joint 0 Current Temperature [B0]	uint8_t	Value from CAN. See joint description
51	Joint 1 Current FSM [B0]	uint8_t	0x00 – FSM_Start 0x01 – FSM_Init 0x02 – FSM_ReadyToOperate 0x03 – FSM_OperationEnable
…	…	…	…
159	Joint 5 Current Temperature [B0]	uint8_t	Value from CAN. See joint description
160	CRC [B1]	uint16_t	-
161	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;
}
```

5.3.5 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;
        }
}
```

5.3.6 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;
        }
}
```

5.3.7 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]			

5.3.8 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;
        }
}
```

5.3.9 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]			

5.3.10 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;
        }
}
```

5.3.11 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]			

5.3.12 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]			