1 General setup

The purpose of the JTC (Joint Trajectory Controller) work is to implement the given trajectory of the manipulator with an open kinematic chain with six degrees of freedom. The trajectory to be realized is defined in the configuration coordinate space as the set of angular positions, angular velocities and angular accelerations for each drive. The drives are controlled by force, taking into account the influence of PID regulators on the control of the angular position. On the basis of the set values, the JTC determines the moments of force that the drives are to develop, taking into account the influence of the regulators and the coefficients of friction compensation inside the drives.

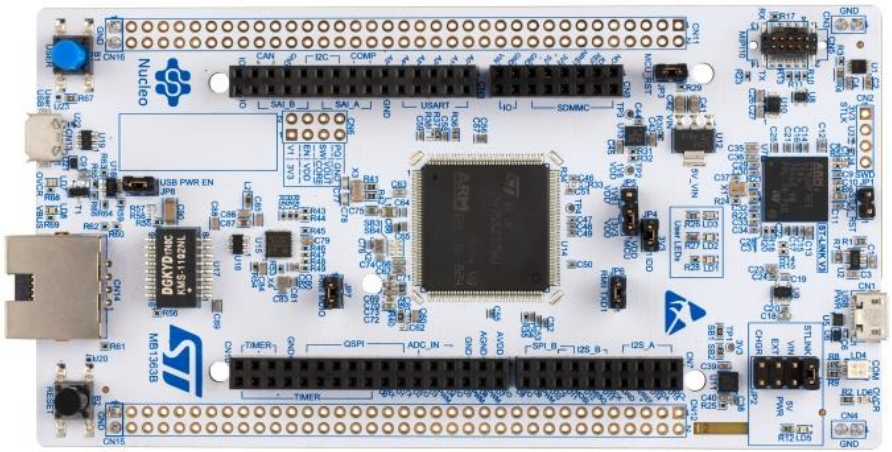
Basic assumptions about the operation of JTC (Joint Trajectory Controller):

- ability to communicate with the host chip via UART serial port and RS422 interface,
    - reading of the given trajectory in the form of position, velocity and angular acceleration for each of the six drives
    - reading the tables containing friction compensation coefficients or reading the command to use the default friction compensation factor tables for each of the six drives
    - reading the PID controller settings or reading the command to use the default PID controller settings for each of the six drives
    - reading the kinematic and dynamic parameters of the manipulator or reading the command to use the default values of kinematic and dynamic parameters of the manipulator
    - reading of control commands allowing to start, pause or stop trajectory execution
    - reading control commands allowing for error cleaning
    - writing telemetry data and confirmations to the host system
- implementation of a trajectory in a configuration space for a manipulator with an open kinematic chain with six degrees of freedom,
    - trajectory interpolation to increase the frequency of points to 1kHz,
    - solving the problem of inverse dynamics,
    - implementation of algorithms for PID controllers,
    - interpolation of tables containing friction compensation coefficients,
- ability to communicate with six drive controllers via the FDCAN bus,
    - cyclic transmission (1kHz) of data to drive controllers, containing information on the value of the set torque and the set state of the controller operation
    - cyclical reception (1kHz) of telemetry data from drive controllers containing information about the current position, speed, torque, temperature, machine condition, errors and warnings.

2 Required hardware and software

2.1 Hardware

JTC is based on the STM32H7 series microcontroller. It is compatible with the STM32H745ZI microcontroller and the NUCLEO-H745ZI-Q test board. It can also be easily adapted to the STM32H743ZI microcontroller and the NUCLEO-H743ZI2 test board. The NUCLEO-H745ZI-Q2 module view is shown below.

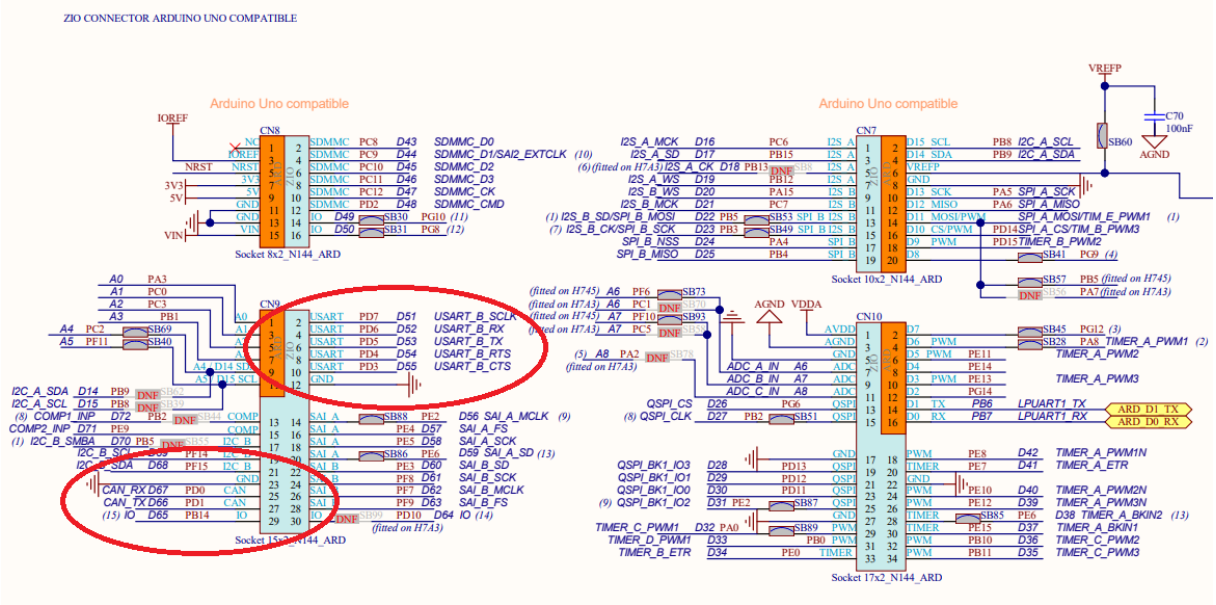


Rys. 2.1 Moduł NUCLEO-H745ZI-Q2

All microcontroller leads used to perform the JTC functions are available in the form of Arduino compatible connectors or via the USB socket of the ST-Link debugger / programmer. The microcontroller pins used are listed below:

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

The JTC firmware was developed in the Keil uVision development environment (MDK Arm https://www.keil.com/).

In the default case, when the STM32H745ZI microcontroller is used, the project must be prepared for both microcontroller cores (Cortex-M4 and Cortex-M7). First, you need to load the executable file on the Cortex-M4 core once. Then load the executable on the Cortex-M7 core. There is no need to reload the executable on the Cortex-M4 core when modifying the project. All calculations are done on the Cortex-M7 core.

When changing the platform to the STM32H743ZI microcontroller, the created project contains firmware only for the Cortex-M7 core. To adapt the source files to the new platform, change the definition of the attached header file from:

```
#include <stm32h7xx.h>
#include <stm32h745xx.h>
```

on the definition:

```
#include <stm32h7xx.h>
#include <stm32h743xx.h>
```

3 Action logic

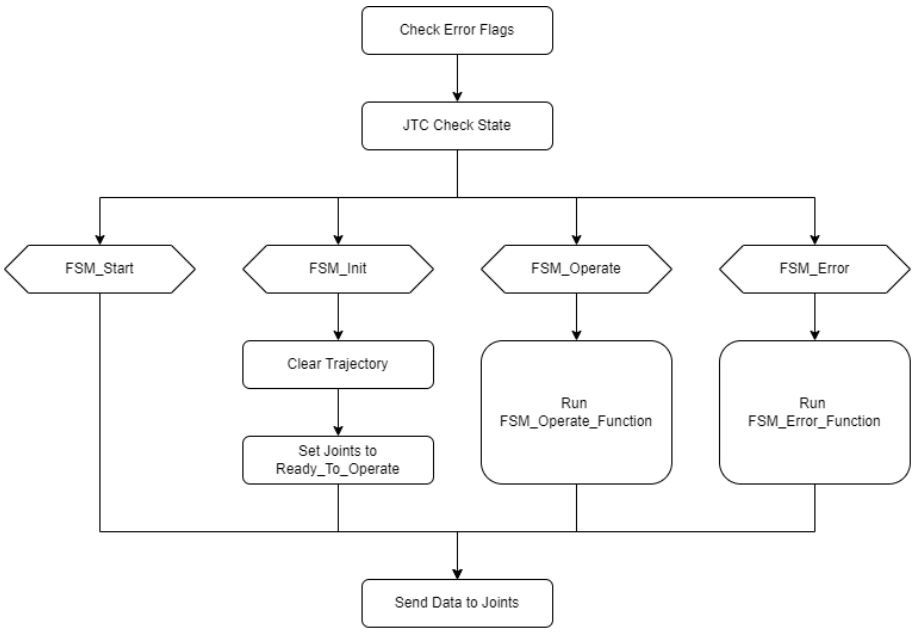
3.1 General control scheme

JTC (Joint Trajectory Controller) can operate in one of four modes:

- FSM_Start – stage right after the processor reset, system initialization,
- FSM_Init – drive initialization, initialization of data from the host system,
- FSM_Operate – normal operation of the device

- FSM_Error – error handling.

The figure below shows a general diagram of the main function of the program. First, the software checks the error flags. Then it checks the current status of the device and the target status of the device and changes it if necessary. Next. one of the four operating modes is supported. After they are completed, communication with the joints is performed. Communication with the host chip is independent of the main program function.



Rys. 3.1 JTC General scheme of operation

The main function of the program is shown below.

```
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

Right after the CPU reset, JTC is in FSM_Start mode. In this mode, the main function of the program is not yet executed. In this mode, the internal circuits of the microcontroller are activated:

- change of the core clock frequency to 480MHz,
- starting the system counter with an interrupt at a frequency of 1kHz,
- starting the clock buses,
- LED configuration
- configuration of global variables
- configuration of serial ports for communication with the host system,
- configuration of the FDCAN bus for communication with drives,
- configuration of kinematic and dynamic parameters of the manipulator for calculations of inverse dynamics,
- configuration of the safety input and output,
- configuration and start-up of the counter with 1kHz interruption to support the main function of the program,

If the above procedure is performed, the device automatically switches to the FSM_Init mode.

3.3 FSM_Init

In this mode, the JTC initializes the drives and initializes the device operating variables. The drives initialize by sending the Joint_FSM_Init command via FDCAN to the drives and waiting for the Joint_FSM_ReadyToOperate command to be received from each drive. Initialization of data from the host system consists in waiting for receiving from the information about the values of the friction coefficient tables, the setting values of the PID controllers and the values of the kinematic and dynamic parameters of the manipulator. The host chip can either send the values of these parameters or issue a command to use the defaults. If during operation any of the drives returns to the Joint_FSM_Init state, JTC also returns to the initialization phase.

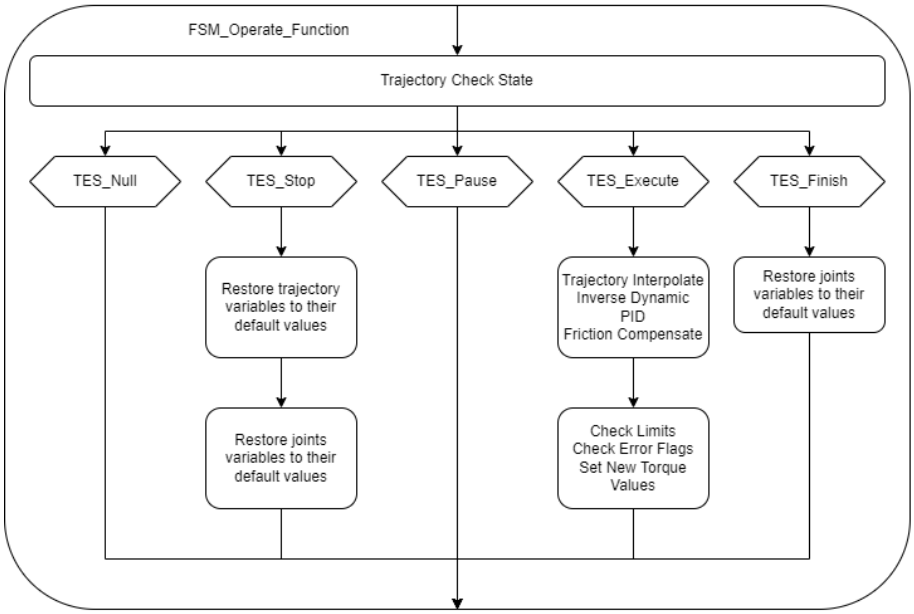
3.4 FSM_Operate

JTC is in FSM_Operate mode only when: all drives were properly initialized, variable values from the host chip were correctly initialized, no error occurred. In this mode, it is possible to implement the trajectory.

The trajectory can be in one of the five basic modes of implementation:

- TES_Null – no received trajectory
- TES_Stop – trajectory received, execution stopped,
- TES_Pause – trajectory received, execution suspended,
- TES_Execute – trajectory received, in progress,
- TES_Finish – trajectory completed.

In the FSM_Operate mode, JTC first checks the current and set status of the trajectory execution. Then it switches it. Next, on the basis of the current status of the trajectory implementation, its implementation is carried out.



Rys. 3.2 FSM_Operate scheme of operation

If the execution status is TES_Null, it means the trajectory has not been received from the host chip or has been cleared by JTC, for example due to previous errors. In this situation, the trajectory should be sent again. In this mode, the drives are in the Joint_FSM_ReadyToOperate state.

If the realization status is TES_Stop, it means that the trajectory has been received, but its realization has not been started yet. The trajectory execution variables are cleared and restored to their starting values. The trajectory should be started by sending an appropriate command. In this mode, the drives are in the Joint_FSM_ReadyToOperate state.

If the execution status is TES_Pause, it means that the trajectory has been received, but its execution has been suspended. The trajectory should be resumed or stopped by sending an appropriate command. In this mode, the drives are in the Joint_FSM_ReadyToOperate state.

If the execution status is TES_Execute, it means that the trajectory has been received and it is being executed. The implementation can be suspended or stopped by sending an appropriate command. The implementation of individual trajectory points is as follows. First, the value of the trajectory points counter is checked and the possibility of reaching the end of the trajectory is checked. The JTC then interpolates the trajectory to obtain a 1ms time step. Further, the inverse dynamics problem is solved. Then, the values of the correction moments of forces from PID controllers are determined. Next, the values of the friction compensation coefficients are determined (two-dimensional interpolation of tables). Based on the above values, a target torque is determined for each drive. Then the limits for the individual values are checked (torque, angular position, angular velocity, angular acceleration, position error). The error flags are then checked to respect the above value limits. If there are no errors, the calculated torque setpoints

are transferred for further transmission to the drives. In this mode, the drives are in the Joint_FSM_OperationEnable state.

If the execution status is TES_Finish, it means that the trajectory has been received and its execution has been completed. You must send a new trajectory, restart the current one. In this mode, the drives are in the Joint_FSM_ReadyToOperate state.

3.5 FSM_Error

JTC goes into FSM_Error on any error. Errors are divided into two main categories: external errors and internal errors.

External errors include:

- detection of a low state on the safety input
- receiving error information from any drive via the FDCAN bus

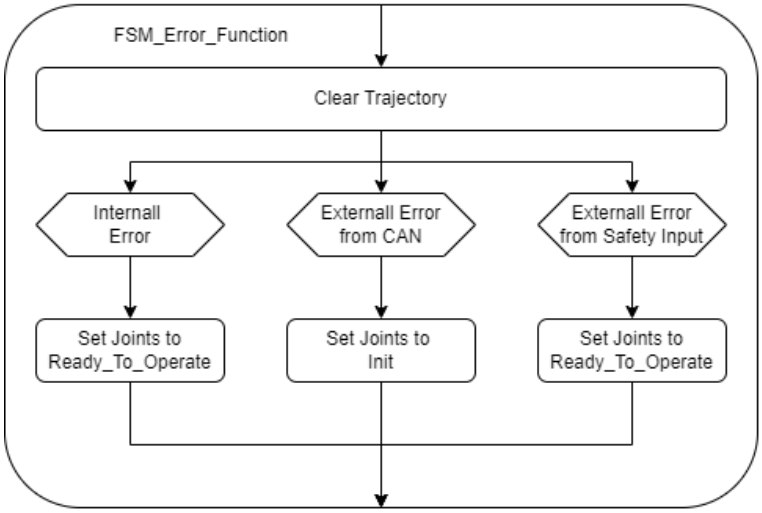
Internal errors include:

- errors in communication with drives via the FDCAN bus (timeout),
- error in communication with the host system (timeout - currently not supported),
- error concerning the exceeding of the permissible ranges for the calculated parameters of the drives (concerning the moment of force, angular position, angular velocity, angular acceleration, position error, lack of the appropriate value of the friction compensation coefficient in the table).

When an internal fault is detected, the safety output is activated. In this situation, the drives change to the Joint_FSM_ReadyToOperate state.

When an external error is detected due to FDCAN receiving an error message from any drive, JTC reinitializes that drive with the Joint_FSM_Init command. All other drives enter the Joint_FSM_ReadyToOperate state. When an external fault is detected due to a low state on the safety input, the drives go to Joint_FSM_ReadyToOperate.

JTC sends information about any errors it detects to the host chip in real time.



Rys. 3.3 FSM_Error scheme of operation

4 Communication JTC – Joints

JTC is adapted to communicate with six drives via the FDCAN bus. The frequency of the FDCAN core in JTC is 85MHz. The speed in nominal mode is 1Mbps, while the speed in data mode is 5Mbps. The bus works in the FD mode with the variable communication speed enabled. The FDCAN configuration parameters are shown in the table.

Tabela 4.1 Configuration of the FDCAN bus

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

FDCAN configuration function:

```
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);
}
```

Communication takes place cyclically with a frequency of 1kHz. In each communication cycle, the JTC sends a broadcast frame that is received by all drives. The frame is 20 bytes long. The frame looks like:

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

Then, in response, each drive sends back a telemetry frame of 12 bytes. These frames are as follows:

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 Communication JTC – Host

5.1 Introduction

Communication between the JTC and the host chip is done via the UART protocol using the RS422 interface. Communication is asynchronous, full duplex, point-to-point. Frame format 115200 8N1, (speed 115200 bps, 8 data bits, 1 stop bit, no parity, no hardware flow control). The host chip should reserve a separate COM port for each connected JTC.

Tabela 5.1 JTC communication - Host: communication parameters

Parameter	Value
Mode	Asynchronous, full duplex
Baudrate	115200 bps (max. 15Mbps)
Number of data bits	8
Number of stop bits	1
Parity control	None
Hardware flow control	None

5.2 Data format and error checking

Data is sent in a binary way in the form of frames. The frame sent to the JTC must be transmitted in a consistent manner, without time gaps. The end of the frame is detected in hardware by detecting an idle state on the receive line for the duration of the two bit transmission. The sending device is responsible for ensuring the consistency of the transmission. If the frame is transmitted intermittently, it will be treated by the JTC as several separate packets. In this situation, JTC has implemented a merge mechanism with timeout. If the interval between packets will be less than 5ms (to be determined - value depending on the speed and communication method - in RT devices there should be no interruptions at all), packets will be merged into one frame. The merging mechanism takes into account the data in the frame, in particular the expected frame length (n field - second and third bytes). If the timeout is exceeded, JTC will send an appropriate message. JTC always transmits frames in a consistent manner with no gaps. The general scheme of the frame is presented in Tabela 5.2

Tabela 5.2 General diagram of the JTC - Host communication frame

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	Depends on the variable	Example data
6	Data1 [B0]	int16_t		
7	Data2 [B3]	uint32_t		Example data

8	Data2 [B2]	int32_t float	Depends on the variable	
9	Data2 [B1]			
10	Data2 [B0]			
…	…			…
n-2	CRC [B1]	uint16_t	0x0000 – 0xFFFF	CRC16 checksum
n-1	CRC [B0]			

A frame always starts with the Header byte. Then the Frame byte is sent, which denotes the type of the frame. The frame length, which can be from 6 to 60,000 bytes, is then transmitted. Multi-byte data is transmitted starting from the oldest byte. Floats are 4 bytes in size and are transmitted from the oldest byte. The frame ends with a CRC16 check sum calculated from bytes numbered 0 to n-3 (all bytes except checksum bytes). The function that calculates the checksum is shown below.

```
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 JTC - Host communication frames

5.3.1 Types of frames

The table below shows basic information about all transmitted frames, such as: name, number, meaning, length, transmission direction.

Tabela 5.3 Types of frames

Name	Frame	Meaning	Length (n)	Transmission direction
Host_FT_null	0x00	Empty frame, do not send	-	-
Host_FT_ClearCurrentErrors	0x01	The command to clear the current errors in JTC	6	Host -> JTC
Host_FT_ClearOccuredErrors	0x02	The command to clear all errors in JTC	6	Host -> JTC
Host_FT_JtcStatus	0x03	Telemetry data from JTC	164	JTC -> Host
Host_FT_Trajectory	0x04	New trajectory for JTC	14 + 36 * x x – liczba punktów w trajektorii	Host -> JTC
Host_FT_FrictionTable	0x05	New 6 Joint Friction Tables	10566	Host -> JTC
Host_FT_FrictionTableUseDefault	0x06	Command to use the default friction coefficient tables for 6 joints	6	Host -> JTC
Host_FT_PidParam	0x07	New PID settings for 6 joints	126	Host -> JTC
Host_FT_PidParamUseDefault	0x08	Command to use the 6 joint PID defaults	6	Host -> JTC
Host_FT_ArmModel	0x09	New values of dynamic and kinematic parameters of the manipulator	538	Host -> JTC
Host_FT_ArmModelUseDefault	0x0A	Command to use the default dynamic and kinematic parameters of the manipulator	6	Host -> JTC
Host_FT_TrajSetExecStatus	0x0B	Trajectory state change command (Stop, Pause, Execute)	7	Host -> JTC

5.3.2 Host_FT_ClearCurrentErrors

Command frame sent from the host chip to the JTC. 6 bytes long. Used to clear flags for current JTC errors.

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

Command frame sent from the host chip to the JTC. Used to clear flags of all JTC errors.

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

Frame sent from JTC to the host chip. The frame contains confirmation of the last received command from the host and telemetry data. The first 5 bytes are the JTC response to the last frame received from the host. This answer consists of four fields:

- Frame Type – number of the frame to which the response relates received by JTC,
- Status – status of correctness of the frame received by JTC of the frame to which the response relates,
- Data Status – data correctness status in the frame received by JTC of the frame to which the response relates,
- Length – length of the frame received by the JTC and related to the response.

Then the JTC telemetry data is sent:

- JTC Current FSM - the current status of the entire JTC,
- JTC Current Errors - current JTC error flags: 0 - no error, 1 - error,
- JTC Occured Errors - JTC error flags that have occurred: 0 - no error, 1 - error,
- JTC Init Status - JTC initialization flags: 0 - initialized, 1 - uninitialized
- Joints Init Status - ionization flags: 0 - initialized, 1 - uninitialized,
- Traj Execution Status - the current status of the trajectory,
- Traj Num Current Point - number of the currently realized trajectory point (applies to trajectory with 1ms step).

Then the telemetry data for communication via CAN is sent:

- CAN Status - current CAN status,
- CAN Current Errors - current CAN error flags: 0 - no error, 1 - error,
- CAN Occured Errors - current CAN error flags: 0 - no error, 1 - error,

Then the telemetry of the drives is sent:

- Joint Current FSM - current drive status,
- Joint mcCurrentError - data received from the drive via CAN, details in the drive operation description,
- Joint mcOccuredError - data received from the drive via CAN, details in the drive operation description,

- Joint currentError - data received from the drive via CAN, details in the drive operation description,
- Joint currentWarning - data received from the drive via CAN, details in the drive operation description,
- Joint Internall Current Errors - current operating errors of the drive generated in JTC,
- Joint Internall Occured Errors - drive operation errors caused by JTC,
- Joint Current Position - data received from the drive via CAN,
- Joint Current Velocity - data received from the drive via CAN,
- Joint Current Torque - data received from the drive via CAN,
- Joint Current Temp - data received from the drive via 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]		

Function that sends data from 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

The frame is sent from the host chip to the JTC. Used to send a new trajectory. The maximum length of the trajectory is 12,000 points. Each point consists of six angular position values, six angular velocity values and six angular acceleration values (a total of 18 numbers of the type int16_t - 36 bytes). The data is sent in the form of numbers of the int16_t type. Determining the value in the int16_t format should be performed according to the following formulas:

- 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

The trajectory must be transmitted in segments no longer than 1500 points. Each segment is sent as a separate frame. JTC each time confirms the receipt of a data frame by sending information about its correctness or errors. The trajectory must consist of at least one segment. The number of segments cannot exceed 100. The segments must be sent in the sequence from segment number 0 to segment number x-1 (x = number of segments). Correct transmission of segment number 0 clears the current trajectory and changes JTC to the TES_Null status. After receiving all the segments, the JTC goes to the TES_Stop status. Then the trajectory can be started. In case of an incorrect transmission of a given segment, it should be sent again. In case of sending the segments in the wrong order, start again transmitting the whole trajectory from the segment number 0. The host can stop sending the trajectory at any time and start transmitting it from the beginning. The table below shows a frame transmitting one 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]			

A function that receives data in 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);

		// the number of points in the received segment
		np = (nd - 14) / (3 * JOINTS_MAX * 2);
		// Too many points have already been taken from this trajectory
		if((Traj.numRecPoints + np) > TRAJ_POINTSMAX)
		{
```

```
                Com.rxFrame.dataStatus = Host_RxDS_TrajTooManyPoints;
                return;
        }
        // Too many segments on this trajectory have already been received
        if(segNum > TRAJ_SEGSSMAX || segMax > TRAJ_SEGSSMAX)
        {
                Com.rxFrame.dataStatus = Host_RxDS_TrajTooManySegs;
                return;
        }
        // segment with wrong number was received (nip: wrong sequence of transmitted segments)
        if(segNum != 0 && (((int16_t)(segNum - Traj.numSeg) != 1) || (segNum >= segMax)))
        {
                Com.rxFrame.dataStatus = Host_RxDS_TrajIncorrectSegOrder;
                return;
        }
        // an incorrect time step was received (e.g. stepTime = 0)
        if(stepTime == 0)
        {
                Com.rxFrame.dataStatus = Host_RxDS_TrajIncorrectStepTime;
                return;
        }
        // the received data is correct
        Com.rxFrame.dataStatus = Host_RxDS_NoError;
        // segment 0 means new trajectories, the previous one is cleared
        if(segNum == 0)
                Control_TrajClear();
        // checking if the received segment is the last one in the trajectory
        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

The frame is sent from the host chip to the JTC. Used to set new values for the friction compensation tables. The data field contains values for six drives. The values are sent starting with drive number 0. The size of the array for one drive is 22 lines and 20 columns. The first line contains the velocity values in rad / s. The second line shows the temperature in degrees Celsius. The following lines contain the values of the frictional moment for the given speed and temperature. All data in the table are in float format.

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]			

A function that receives data in 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

Command frame sent from the host chip to the JTC. Used to restore the default values of the friction compensation tables.

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

The frame is sent from the host chip to the JTC. Used to set new PID settings. The data field contains values for six drives. The values are sent starting with drive number 0. 5 numerical float values (20 bytes) are sent for each drive. The values are:

- gain of the proportional element pidKp,
- strengthening of the inertial element pidKi,
- gain of the derivative element pidKd,
- saturation of the error integral - minimum value of pidErrorIntMin,
- saturation of the error integral - maximum value of 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]			

A function that receives data in 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

Command frame sent from the host chip to the JTC. It is used to restore the default values of PID regulators.

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

The frame is sent from the host chip to the JTC. Used to set new values for kinematic and dynamic parameters of the manipulator. The data field contains values in the float format (4 bytes). First, the joint coordinate system parameters are sent. Then the parameters of the coordinate systems, moments of inertia and masses for the links are sent.

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]			

A function that receives data in 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

Command frame sent from the host chip to the JTC. Used to restore the default values of the manipulator's kinematic and dynamic parameters.

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

Command frame sent from the host chip to the JTC. Used to change the trajectory execution mode. The allowed modes are: 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]			