FORTE

FORTE_APP01 – forte.dll description

Application note

ASIX s.r.o.
Staropramenna 4
150 00 Prague
Czech Republic

www.asix.net

support@asix.net

sales@asix.net

ASIX s.r.o. reserves the right to make changes to this document, the latest version of which can be found on the Internet.

ASIX s.r.o. renounces responsibility for any damage caused by the use of ASIX s.r.o. products.

Table of Contents

1 forte.dll

1.1 Introduction

Functions implemented in forte.dll enable setting and reading of logical levels on single pins of FORTE programmer. This way it is possible to make various communication protocols.

Except for functions which enable controlling of the single pins, the library contains also functions prepared for communication via SPI, I^2C and 1-Wire buses, functions for supply and programming voltage controlling and supply voltage and GO button reading.

1.2 Programmer pins marking

Single pins are marked the same way as they are marked on the box of the programmer.

Pin	Type	Description
P	I/O, VPP	logical input/output or VPP output
VDD	PWR	supply input/output
GND	PWR	ground
D, C, I, L, T, S, R	I/O	log. input/output

Table 1: Features of pins

Sense: I/O - input and output pin, VPP - programming voltage

1.3 How to work with the programmer

Instructions are executed in a queue what corresponds with the USB way of work. The order of reading answers corresponds with the order of the instructions (*Q...* functions) given.

Returned data can be read either blocking way via *AGetBlocking* or not blocking way via *AGet* function. For not blocking reading of bigger amount of data it is also possible to use *AGetBlock* together with *AGetStatus* function.

Waiting for every answer, e.g. from *QGetPins* function, would slow down the work dramatically. When it is not needed to know the previous answer for continuation, it is advisable to wait not blocking way and to read the answers when they are available. In meantime other functions can be called.

At some of the instructions it is advisable to wait for their answer before continuing, it is for example *QOpenProg*. The cycle **instruction → FORTE → answer** takes from several milliseconds to tens of milliseconds.

Output functions are inserted to the queue and executed consecutively, but their answers are sent already when they are inserted to the queue. This applies also to the delay functions.

The functions which read or measure something, answer after they are executed.

1.4 List of the functions

```
void __stdcall QOpenProg(int sn);
void __stdcall QCloseProg(void);
void __stdcall QSetActiveLED(int led);
void __stdcall QPoweronVdd(int delayus, int Volt
age_mV);
void __stdcall QPoweroffVdd(void);
```

```
void __stdcall QPoweronVpp(int Voltage_mV);
void __stdcall QPoweroffVpp(void);
void __stdcall QDelay(int delayus);
void __stdcall QDelay_ns(int delayns);
void __stdcall QSetPullUpDowns(int pullupdowns);

void __stdcall QCheckGoButton(void);
void __stdcall QCheckSupplyVoltage(void);
void __stdcall QSetGPIOAnswer(bool answer);
void __stdcall QSetPins(int pins);
void __stdcall QGetPins(void);
void __stdcall QShiftByte(int databyte, int mode
);
void __stdcall QShiftByte_OutIn(int databyte, in
t mode, int InputPin);
void __stdcall QShiftBytes(int *buf, int mode, i
nt count);
void __stdcall QShiftBytes_OutIn(int *buf, int m
ode, int InputPin, int Count);
void __stdcall QShiftBytes_In(int mode, int Inpu
tPin, int count);
void __stdcall QShiftBits(int data, int mode, in
t bits_count);
void __stdcall QShiftBits_OutIn(int data, int mo
de, int InputPin, int bits_count);
void __stdcall QSetShiftSpeed(int speed);
void __stdcall Q1WireInit(void);
void __stdcall Q1WireWriteByte(int data, int str
ong_pullup_time_us);
void __stdcall Q1WireReadByte(void);
void __stdcall QI2CStart(bool UseIntPullUps);
void __stdcall QI2CStop(void);
void __stdcall QI2CWriteByte(int databyte);
void __stdcall QI2CReadByte(bool ACK);
void __stdcall QI2CSetSpeed(int speed);
bool __stdcall AGet(int *answer);
int __stdcall AGetBlocking(void);
bool __stdcall AGetBlock(int *buf, int count, in
t *count_returned);
bool __stdcall AGetStatus(int *NumberOfAnswers);

void __stdcall AGetProgList(int *sn_list, int co
unt, int *count_returned);
void __stdcall AClearFatalError(void);
```

1.5 Functions description

1.5.1 QOpenProg

The function tries to open a FORTE. If the **sn** variable is - 1, the function opens one FORTE regardless of its serial number. In other cases the **sn** means the FORTE serial number. If the FORTE serial number is A6041234, the **sn** should be 0x1234 or 0x041234. The serial number is defined with last 4 or 6 signs in the hex form.

Function definition:

void __stdcall QOpenProg(int sn);

Parameter:

sn - Serial number of the programmer.

Return values:

OPEN_OK - Openning was not succesful.

OPEN_NOTFOUND - Programmer was not found.

OPEN_CANNOTOPEN - It was not possible to open the programmer.

OPEN_ALREADYOPEN - Programmer si already open.

OPEN_BADDRIVERVERSION - Wrong version of the USB driver.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

```
QOpenProg(0x041234); // open FORTE SN A6041234
```

1.5.2 QCloseProg

It closes the FORTE and switches the output voltages off.

Function definition:

void __stdcall QCloseProg(void);

Return values:

CLOSE_OK

CLOSE_CANNOTCLOSE - Programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.3 QSetActiveLED

The function switches on, off or sets blinking of the ACTIVE LED of the FORTE programmer.

Function definition:

void __stdcall QSetActiveLED(int led);

Parameter:

led - The variable defines the required state of the ACTIVE LED, see constants.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To switch the yellow programmer ACTIVE LED on, call QSetActiveLED(LED_ACT_Y) function, to switch it off, call QSetActiveLED(LED_ACT_OFF).

1.5.4 QPoweronVdd

The function switches on the supply voltage from the programmer on its VDD pin, then it waits for specified time and checks whether the current is over 100 mA. If the current is higher, the programmer switches the voltage off. If there is a short circuit on the VDD pin, the supply voltage will not be present for much longer time than the specified time is.

The function returns a value in accordance with the result of the operation. Although the result is returned in about 20 ms, the voltage is already switched off, this is solved in the HW. It is recommended to choose the time carefully, because long specified time is dangerous for the programmer circuits if there is an error in connections.

If the internal supply voltage from the programmer is switched off, an external supply voltage 1.2 to 5.5 V may be connected to the programmer. The data pins logical levels are set in accordance with the VDD supply voltage.

When the supply voltage is under 1.8 V, the programmer can communicate with reduced speed only.

Function definition:

void __stdcall QPoweronVdd(int delayus, int Voltage_mV);

Parameters:

delayus - Time in µs after what the overcurrent will be checked.

Voltage_mV - The size of the voltage supplied from the programmer in mV. The supply voltage can be between 1.2 and 5.5 V.

Return values:

POWERON_OK - The supply voltage has been switched on successfully.

POWERON_OCURR - Overcurrent had been detected, supply voltage was switched off.

POWERON_WRONG_LEVEL - Wrong level of the voltage has been entered.

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To switch on the internal supply voltage of 3.3 V on the VDD pin and to check the overcurrent after 10 ms, call function

```
QPoweronVdd(10000, 3300);
```

1.5.5 QPoweroffVdd

The function switches off the VDD supply voltage provided by the programmer.

Definice funkce:

void __stdcall QPoweroffVdd(void);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.6 QPoweronVpp

The function switches on the programming voltage on the P pin of the programmer. If the overcurrent is detected on the P pin after the voltage is switched on, it is switched off. The function sends the operation result as answer.

Function definition:

void __stdcall QPoweronVpp(int Voltage_mV);

Parameter:

Voltage_mV - The size of the programming voltage in mV. The voltage can be in range 6.5 to 17 V.

Return values:

VPP_OK - Programming voltage has been switched on.

VPP_OCURR - Overcurrent has been detected, the programming voltage has been switched off.

VPP_WRONG_LEVEL - Wrong level of the voltage has been entered.

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.7 QPoweroffVpp

The function switches off the programming voltage on the P pin.

Function definition:

void __stdcall QPoweroffVpp(void);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.8 QDelay

The programmer waits for specified time.

Function definition:

void __stdcall QDelay(int delayus);

Parameter:

delayus - Waiting time in µs.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or

AGetBlock functions.

Example:

To do a delay of 7 ms in the signals, call function

```
QDelay(7000);
```

1.5.9 QDelay_ns

The programmer waits for specified time. The timer granularity is 16.67 ns, the specified value is rounded to the nearest higher multiple of 16.67 ns.

There can appear a longer delay in the signals, because of the commands delays on USB.

Functions definition:

void __stdcall QDelay_ns(int delayns);

Parameter:

delayns - Waiting time in ns.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To do a delay of at least 33 ns in the signals, call function

```
QDelay_ns(33);
```

Programmer will do delay of 16,67*2=33,34 ns.

1.5.10 QSetPullUpDowns

The function connects/disconnects the 2k4 pull-up or pull-down resistors on selected pins of the programmer. In the default state the resistors are disconnected.

Function definition:

void __stdcall QSetPullUpDowns(int pullupdowns);

Parameter:

pullupdowns - Variable specifying which resistors will be connected to the data pins, see constants.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To connect the D pin pull-up resistor and L pin pull-down, call function

```
QSetPullUpDowns((PULLUP<<D_PULL)|
(PULLDOWN<<L_PULL));
```

1.5.11 QCheckGoButton

The function checks the programmer button and sends its state as result.

Function definition:

void __stdcall QCheckGoButton(void);

Return values:

GO_BUTTON_NOT_PRESSED

GO_BUTTON_PRESSED

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To find out if the programmer button has been pressed, call *QCheckGoButton* function and then if the *AGet(data)* function returns 0x90001, the button has been pressed.

```
if (data==GO_BUTTON_PRESSED)
 {// the button is pressed}
```

1.5.12 QCheckSupplyVoltage

In its answer the function sends a code corresponding with the supply voltage measured on the VDD pin of the programmer.

Function definition:

void __stdcall QCheckSupplyVoltage(void);

Return values:

SUPPLY_VOLTAGE_CODE + measured voltage in V x10, e.g. 33 means 3.3 V

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example: To check the VDD supply voltage value, call *QCheckSupplyVoltage* function and then read the result with *AGet* function. The *AGet* will return for example 0x7001B, where 0x1B is 10x the size of the voltage in hexadecimal form, so 27 in decimal form, then the measured voltage is 2.7 V.

1.5.13 QSetGPIOAnswer

This function enables or disables the answers from the output functions. After the programmer has been opened, the answers are always enabled. Even when the answers are disabled, functions always return NOT_OPENED, when they are called and the programmer is not open.

When there appears a fatal error, it replaces the answer of the called function. When the output functions answers are disabled, the fatal errors are returned after calling of the input functions only.

This function affects answers of *QDelay*, *QDelay_ns*, *QSetPins*, *QShiftByte*, *QShiftBytes*, *QShiftBits*, *QSetShiftSpeed*, *Q1WireWriteByte*, *QI2CStart*, *QI2CStop*, *QI2CWriteByte* and *QI2CSetSpeed* functions.

Function definition:

void __stdcall QSetGPIOAnswer(bool answer);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.14 QSetPins

The function sets the output pins of the programmer in accordance with the constants. The D and C, I and L, P and R, S and T pins are always set together. When only one pin of the coulple is defined, the state of the second pin is set in accordance with its saved value.

The state of the other not defined pins does not change.

Function definition:

void __stdcall QSetPins(int pins);

Parameter:

pins - The variable defines required values on the programmer pins. See constants.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To set D to log.1, C to log.0 and the other pins leave unchanged, call function

```
QSetPins((PINS_HI<<PINS_D_BIT)|
(PINS_LO<<PINS_C_BIT));
```

1.5.15 QGetPins

The function sends back the values that the programmer sees on its pins. See constants for *QGetPins*.

Function definition:

void __stdcall QGetPins(void);

Return values:

GETPINS_CODE + pins values

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To read the I pin state, call the *QGetPins* function and then read returned data using *AGet* function. The *AGet* function for example returns 0x4000C value. In this value all the pins values are returned, so the I pin state must be filtered with **GETPINS_PINI** constant. In our example the value which has been read on the I pin is log. 1.

```
if (AGet(*data))
 { if ((data & GETPINS_PINI)==GETPINS_PINI )
     {  //on the I pin there is log. 1}
    else {//on the I pin there is log. 0}
 }
```

1.5.16 QShiftByte

The function sends a Byte on the D pin and generates clock signal on the C pin. The Byte value is specified by the **databyte** variable. The **mode** variable specifies a mode in accordance with the SPI definition.

When the user selects a mode that does not correspond with the current logic level on the C pin, the C logic level is first set to the required state. For example if there is log.0 on the C pin and mode=3, the C will first change to log. 1 and then the **databyte** will be sent.

The LSB is sent first, the communication frequency can be set using *QSetShiftSpeed* function.

Function definition:

void __stdcall QShiftByte(int databyte, int mode);

Parameters:

databyte - Variable for data to be sent.

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To send a 0x3A Byte in the SPI mode 1, call function

```
QShiftByte(0x3A, SHIFT_MODE1);
```

1.5.17 QShiftByte_OutIn

The function generates the C and D signals in accordance with specified parameters as *QShiftByte* function do, but in addition it also reads data from the chosen pin at the same time. The input pin can be chosen with **InputPin** variable value. See constants defining possible values of the **InputPin** variable.

If the D pin is chosen as input, it is first set to the high impedance state and the programmer only reads.

Function definition:

void __stdcall QShiftByte_OutIn(int databyte, int mode, int InputPin);

Parameters:

databyte - Variable for the data to be sent.

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

InputPin - The input pin is chosen in accordance with this variable.

Return values:

SHIFT_BYTE_OUTIN_CODE + read data

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To send a 0x4C Byte in SPI mode 3 and at the same time to read input data on the I pin, call function

```
QShiftByte_OutIn(0x4C, SHIFT_MODE3, SHIFT_OUTIN_
PINI);
```

1.5.18 QShiftBytes

Similarly as the *QShiftByte*, which sends one Byte only, this function sends more Bytes on the D and C pins.

Function definition:

void __stdcall QShiftBytes(int *buf, int mode, int count);

Parameters:

buf - Array of data to be sent.

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

count - Variable defining the number of Bytes to be sent.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.19 QShiftBytes_In

Same as the *QShiftByte_OutIn*, this function reads data from the selected input pin and generates clock signal on the C pin, but it is also able to read more Bytes of data on one calling, which can be useful e.g. for large SPI memories reading.

This function is input only, it cannot send data.

If the D pin is chosen as input, it is first set to the high impedance state.

Function definition:

void __stdcall QShiftBytes_In(int mode, int InputPin, int count);

Parameters:

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

InputPin - The input pin is chosen in accordance with this variable.

count - Variable defining the number of Bytes to be read. With this function it is possible to read maximally 512 Bytes.

Return values:

SHIFT_BYTE_OUTIN_CODE + read data - A value is returned for each of the read Bytes.

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To read 100 Bytes in SPI mode 0 on the I pin, call function

```
QShiftBytes_In(SHIFT_MODE0, SHIFT_OUTIN_PINI, 100);
```

1.5.20 QShiftBytes_OutIn

Similarly as the *QShiftByte_OutIn*, which sends and reads one Byte only, this function sends more Bytes on the D and C pins and reads from a selected pin.

Function definition:

void __stdcall QShiftBytes_OutIn(int *buf, int mode, int InputPin, int Count);

Parameters:

buf - Array of data to be sent.

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

InputPin - The input pin is chosen in accordance with this variable.

count - Variable defining the number of Bytes to be sent.

Return values:

SHIFT_BYTE_OUTIN_CODE + read data - A value is returned for each of the read Bytes.

OK

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.21 QShiftBits

The function sends selected number of bits on pins D and C the same way as *QShiftByte* sends Bytes.

It can be useful for some protocols to be able not to send data in Bytes only.

The LSB is sent first, the communication frequency can be set using *QSetShiftSpeed* function.

Function definition:

void __stdcall QShiftBits(int data, int mode, int bits_count);

Parameter:

data - Variable for the data to be sent.

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

bits_count - Variable defining number of bits to be sent. It is possible to send 1 to 16 bits.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To send 2 bits 0, 1, in SPI mode 0, call function

```
QShiftByte(0x02, SHIFT_MODE0,2);
```

1.5.22 QShiftBits_OutIn

The function sends selected number of bits on pins D and C and reads on a selected pin the same way as *QShiftByte_OutIn* sends Bytes.

It can be useful for some protocols to be able not to send data in Bytes only.

If the D pin is chosen as input, it is first set to the high impedance state and the programmer only reads.

The LSB is sent first, the communication frequency can be set using *QSetShiftSpeed* function.

Function definition:

void __stdcall QShiftBits_OutIn(int data, int mode, int InputPin, int bits_count);

Parameters:

data - Variable for the data to be sent.

mode - Variable defining SPI mode, its value may be 0, 1, 2 or 3.

InputPin - The input pin is chosen in accordance with this variable.

bits_count - Variable defining number of bits to be sent. It is possible to send 1 to 16 bits.

Return values:

OK

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Example:

To send 4 bits 0xA, in SPI mode 3 and read on the I pin, call function

```
QShiftBits_OutIn(0x0A, SHIFT_MODE3, SHIFT_OUTIN_
PINI, 4);
```

1.5.23 QSetShiftSpeed

The function sets the clock frequency on the C pin for *QShift...* functions.

Function definition:

void __stdcall QSetShiftSpeed(int speed);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

WRONG_INPUT - Wrongly entered parameters.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Parameter:

speed - Defines the clock speed, see constants definition.

Example:

To set the clock frequency for *QShift...* functions to 1 MHz, call function

```
QSetShiftSpeed(SHIFT_CLK_1000kHz);
```

1.5.24 Q1WireInit

This function does the initialization sequence on the 1-Wire bus, it makes a reset pulse and reads a presence pulse from the device.

The 1-Wire bus functions communicate on the P pin of the programmer. On the bus a pull-up resistor in accordance with the 1-Wire specification have to be connected.

Function definition:

void __stdcall Q1WireInit(void);

Return values:

_1WIRE_PRESENT - Device has answered, log.0 has been read on the bus.

_1WIRE_NOT_PRESENT - Device has not answered, log.1 has been read on the bus.

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.25 Q1WireWriteByte

It sends a Byte on the 1-Wire bus. When a nonzero time for strong pull-up is selected, log.1 is connected to the bus during this time.

Strong pull-up connection is implemented as connection of log.1 to the bus, the maximal current drawn from the pin must not be higher than the current stated in the programmer specifications in its manual.

Function definition:

void __stdcall Q1WireWriteByte(int data, int strong_pullup_time_us);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Parameter:

data - Variable for the data to be sent. Data are sent LSB first.

strong_pullup_time_us - This variable defines the time, during which the strong pull-up (log.1) should be connected, after the Byte has been sent on the bus.

Example:

To send 0xCC Byte and not to use the strong pull-up, call function

```
QSetPrestoSpeed(0xCC, 0);
```

1.5.26 Q1WireReadByte

The function reads one Byte from the 1-Wire bus.

Function definition:

void __stdcall Q1WireReadByte(void);

Return values:

_1WIRE + read data

NOT_OPENED - The programmer has not been opened.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.27 QI2CStart

This function makes a start bit on the I²C bus.

The I²C bus functions communicate on the D (SDA) and C (SCL) pins of the programmer. With a parameter it is possible to select if the internal pull-up resistors should be connected to both of the pins or if there are external resistors.

Communication frequency can be set using *QI2CSetSpeed* function.

Function definition:

void __stdcall QI2CStart(bool UseIntPullUps);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Parameter:

UseIntPullUps - When it is true, internal pull-up resistors are used. When it is false, internal pull-up resistors remain in the same state as they were before the function has been called.

1.5.28 QI2CStop

This function makes a stop bit on the I²C bus.

Communication frequency can be set using *QI2CSetSpeed* function.

Function definition:

void __stdcall QI2CStop(void);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

1.5.29 QI2CWriteByte

This function writes a Byte on the I²C bus.

Communication frequency can be set using *QI2CSetSpeed* function.

Function definition:

void __stdcall QI2CWriteByte(int databyte);

Return values:

NOT_OPENED - The programmer has not been opened.

I2C_ACK - After the Byte had been sent, the device answered (ACK).

I2C_NACK - After the Byte had been sent, the device did not answer (NO ACK).

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Parameter:

databyte - Variable for data to be sent. The Byte is sent MSB first.

Example:

To send 0xAB Byte, call function

```
QI2CWriteByte(0xAB);
```

1.5.30 QI2CReadByte

This function reads a Byte from I²C bus.

Communication frequency can be set using *QI2CSetSpeed* function.

Function definition:

void __stdcall QI2CReadByte(bool ACK);

Return values:

NOT_OPENED - The programmer has not been opened.

I2C_CODE + read data

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Parameter:

ACK - When it is true, programmer sends ACK, after the Byte has been sent, else it sends NO ACK.

1.5.31 QI2CSetSpeed

This function sets the communication frequency for functions communicating on the I²C bus. See constants specification.

Function definition:

void __stdcall QI2CSetSpeed(int speed);

Return values:

OK

NOT_OPENED - The programmer has not been opened.

Using *QSetGPIOAnswer* function it is possible to disable the answers, in such case the function returns NOT_OPENED only, when the programmer is not open.

Return values are returned via *AGet*, *AGetBlocking* or *AGetBlock* functions.

Parameter:

speed - Specifies selected communication frequency. After the programmer has been opened, the frequency is set to 100 kHz. See constants specification.

Example:

To set the I²C communication frequency to 400 kHz, call function

```
QI2CSetSpeed(I2C_CLK_400kHz);
```

1.5.32 AGet

The function returns bool value which says whether an answer is available. If the answer is available, its value is returned in the function parameter.

Function definition:

bool __stdcall AGet(int *answer);

Return values:

The function returns True if the returned data are available, if they are not, it returns False.

answer - Returned answer value.

Example:

To find out if the programmer has answered and what its answer is, test it with function

```
if (AGet(*data))
```

```
 { // the returned value is available in the dat
a variable}
```

1.5.33 AGetBlocking

The function waits until the answer is available and then it returns the answer value.

Function definition:

int __stdcall AGetBlocking(void);

Return value:

The function returns answer value.

Example:

To wait until the programmer answer is available and then to continue, use the *AGetBlocking*. This function can be used for example after the programmer has been opened.

```
QOpenProg(-1);
if (AGetBlocking()==OPEN_OK)
  {// programmer open OK}
else
  {// programmer open failed}
```

1.5.34 AGetBlock

The function returns requested number of answers. When there is not enough answers available, it returns as much answers as much are available. The function is not blocking.

It is useful to use this function together with *AGetStatus* function.

Function definition:

bool __stdcall AGetBlock(int *buf, int count, int *count_returned);

Parameters:

buf - Array of integer, where the answers are returned.

count - Variable defining number of answers to be read.

count_returned - Variable returning number of answers, which have been returned in buf.

Return values:

The function returns False, when there appears a fatal error. In such a case it returns only one value in the array, only the error code.

When there is no fatal error, function returns True and in the **buf** array it returns the answers, in the **count_returned** variable it returns number of the returned values.

The function can return maximally 65536 values.

Example: To wait until 10 answers is available and afterwards to read them at once, use *AGetBlock* function

```
int data[10];
int data_returned;
int i;

i=0;
while (i<10)
 {
  if (!AGetStatus(*i))
    { // Fatal error, exit with error report}
 }
if (!AGetBlock(data, 10, *data_returned))
  { // Fatal error, exit with error report}
```

1.5.35 AGetStatus

In a parameter the function returns number of answers, which are available.

Function definition:

bool __stdcall AGetStatus(int *NumberOfAnswers);

Parameter:

NumberOfAnswers - Variable returning number of answers, which are available.

Return values:

Function returns False, when there appears a fatal error.

When there appears no fatal error, function returns True and in the **NumberOfAnswers** it returns number of answers, which are available for read.

1.5.36 AGetProgList

In a parameter the function returns list of the FORTE programmers, which are available.

Function definition:

void __stdcall AGetProgList(int *sn_list, int count, int *count_returned);

Parameters:

sn_list - Array of integer, which returns the list of the serial numbers of the available FORTE programmers. The serial nubers are returned as 24bit values, same as they are listed in the UP software.

count - Variable defining the number of the serial numbers to be read.

count_returned - Variable returning number of serial numbers, which have been returned in sn_list.

Return values:

Regardless of the fatal errors, the function returns list of available programmers in **sn_list** and the number of the returned serial numbers in the **count_returned**.

1.5.37 AClearFatalError

The function erases fatal error.

After the error is erased the FORTE is closed and it must be opened again. No commands in the queue will be executed and the answers that should have come via *AGet*, *AGetBlocking* or *AGetBlock* are lost.

Function definition:

void __stdcall AClearFatalError(void);

1.6 Fatal errors

None of the above described functions *Q...* returns fatal errors, they are generated asynchronously. If such an error appears, the *AGet*, *AGetBlocking* and *AGetBlock* repeats the one error value until the error is erased with *AClearFatalError*. After the fatal error is erased, the FORTE is closed and it must be opened again. Any instructions in the queue will not be executed and the answers that should come via *AGet*, *AGetBlocking* or *AGetBlock* are lost.

The fatal errors appear if there is overcurrent detected on the supply voltage or on the programming voltage power supply or if there is more than 6 V measured on the VDD pin.

Attention! If the fatal error is caused by a voltage over 6 V detected on the VDD pin, the fatal error does not save the programmer against its damage. First of all, the programmer must be immediately disconnected from the power supply.

1.7 Answers

```
OPEN_OK = 0x10000;
OPEN_NOTFOUND = 0x10001;
OPEN_CANNOTOPEN = 0x10002;
OPEN_ALREADYOPEN = 0x10003;
OPEN_BADDRIVERVERSION = 0x10004;
CLOSE_OK = 0x20000;
CLOSE_CANNOTCLOSE = 0x20001;
POWERON_OK = 0x30000;
POWERON_OCURR = 0x30001;
POWERON_WRONG_LEVEL = 0x30002;

GETPINS_CODE = 0x40000; //
ored with GETPINS_PINx
      GETPINS_PIND = 0x01;
      GETPINS_PINC = 0x02;
      GETPINS_PINI = 0x04;
```

```
      GETPINS_PINL = 0x08;
      GETPINS_PINP = 0x10;
      GETPINS_PINR = 0x20;
      GETPINS_PINS = 0x40;
      GETPINS_PINT = 0x80;
OK = 0x50000;
NOT_OPENED = 0x50001;
WRONG_INPUT = 0x50002;
SHIFT_BYTE_OUTIN_CODE = 0x60000;
SUPPLY_VOLTAGE_CODE = 0x70000;
VPP_OK = 0x80000;
VPP_OCURR = 0x80001;
VPP_WRONG_LEVEL = 0x80002;
GO_BUTTON_NOT_PRESSED=0x90000;
GO_BUTTON_PRESSED=0x90001;
SHIFT_BITS_OUTIN_CODE = 0xA0000;
_1WIRE = 0xB0000;
_1WIRE_PRESENT = 0xB0100;
_1WIRE_NOT_PRESENT = 0xB0200;
I2C_CODE  = 0xC0000;
I2C_ACK   = 0xC0100;
I2C_NACK  = 0xC0200;

FATAL_OVERCURRENTVDD = 0x01;
FATAL_OVERCURRENTVPP = 0x02;
FATAL_OVERVOLTAGEVDD = 0x04;
FATAL_OTHER = 0x08;
```

1.8 Constants

1.8.1 QSetPins constants

```
PINS_HIZ = 0x01;
PINS_LO = 0x02;
PINS_HI = 0x03;
PINS_D_BIT = 0x00;
PINS_C_BIT = 0x02;
PINS_I_BIT = 0x04;
PINS_L_BIT = 0x06;
PINS_P_BIT = 0x08;
PINS_R_BIT = 0x0A;
PINS_S_BIT = 0x0C;
PINS_T_BIT = 0x0E;
```

Příklad:

```
PINS_D_HI = PINS_HI << PINS_D_BIT;
PINS_D_LO = PINS_LO << PINS_D_BIT;
PINS_D_HIZ = PINS_HIZ << PINS_D_BIT;
```

1.8.2 QShift... constants

```
SHIFT_OUTIN_PIND = 0x00;
SHIFT_OUTIN_PINI = 0x02;
SHIFT_OUTIN_PINL = 0x03;
SHIFT_OUTIN_PINP = 0x04;
SHIFT_OUTIN_PINR = 0x05;
SHIFT_OUTIN_PINS = 0x06;
SHIFT_OUTIN_PINT = 0x07;

SHIFT_MODE0=0x00;
SHIFT_MODE1=0x01;
SHIFT_MODE2=0x02;
SHIFT_MODE3=0x03;
```

1.8.3 QSetShiftSpeed constants

```
SHIFT_CLK_15000kHz = 1;
SHIFT_CLK_10000kHz = 2;
SHIFT_CLK_7500kHz = 3;
SHIFT_CLK_6000kHz = 4;
SHIFT_CLK_5000kHz = 5;
SHIFT_CLK_3750kHz = 6;
SHIFT_CLK_3330kHz = 7;
SHIFT_CLK_3000kHz = 8;
SHIFT_CLK_2500kHz = 9;
SHIFT_CLK_2000kHz = 10;
SHIFT_CLK_1500kHz = 11;
SHIFT_CLK_1000kHz = 12;
SHIFT_CLK_750kHz = 13;
SHIFT_CLK_600kHz = 14;
SHIFT_CLK_500kHz = 15;
SHIFT_CLK_400kHz = 16;
SHIFT_CLK_375kHz = 17;
SHIFT_CLK_333kHz = 18;
SHIFT_CLK_300kHz = 19;
SHIFT_CLK_250kHz = 20;
```

```
SHIFT_CLK_200kHz = 21;
SHIFT_CLK_150kHz = 22;
SHIFT_CLK_120kHz = 23;
SHIFT_CLK_100kHz = 24;
SHIFT_CLK_75kHz = 25;
SHIFT_CLK_60kHz = 26;
SHIFT_CLK_50kHz = 27;
SHIFT_CLK_40kHz = 28;
SHIFT_CLK_37kHz = 29;
SHIFT_CLK_33kHz = 30;
SHIFT_CLK_30kHz = 31;
```

```
LED_ACT_R_FAST_BLINK  = 0x06;
LED_ACT_YR_FAST_BLINK = 0x07;
```

1.8.4 QSetPullUpDowns constants

```
PULLDOWN = 0x01;
PULLUP  =  0x02;
D_PULL = 0x00;
C_PULL = 0x02;
I_PULL = 0x04;
L_PULL = 0x06;
S_PULL = 0x08;
T_PULL = 0x0A;
P_PULL = 0x0C;
R_PULL = 0x0E;
```

1.8.5 QI2CSetSpeed constants

```
I2C_CLK_100kHz = 0x00;
I2C_CLK_400kHz = 0x01;
I2C_CLK_1MHz = 0x02;
```

1.8.6 QSetActiveLED constants

```
LED_ACT_OFF           = 0x00;
LED_ACT_Y             = 0x01;
LED_ACT_R             = 0x02;
LED_ACT_Y_BLINK       = 0x03;
LED_ACT_R_BLINK       = 0x04;
LED_ACT_Y_FAST_BLINK  = 0x05;
```

2

Document history

Document revision	Modifications made
2015-04-02	Dokument created.
2017-02-01	Functions defintions have been fixed.
	Functions descriptions have been completed and also the chapter How to work with the programmer.
	Added description of new functions AGetProgList, QShiftBytes, QShiftBytes_OutIn.