Dynamixel Documentation

rkuroi

Summary The purpose of this documentation is to explain how to use the dynamixel servos with the Dynamixel SDK (Software Development Kit). Due to time constraints, it focuses mainly on C, C++ and python programming languages, though the main concepts are the same for all programming languages supported by the SDK. Also, though there are two protocols for the SDK, here only *Protocol 2.0* is addressed, and the functions and examples described here have only been tested on a PC with a U2D2 interface.

I will try to explain as much as I can to allow for basic manipulation of the SDK but you should always check the official documentation out at ROBOTIS e-Manual.

Keywords

control table item variable present in the servo controller memory (see ROBOTIS e-Manual (ex: XM430-W210)).

computer I will use this word to refer to the hardware running the programmes, but keep in mind that you can indeed use the SDK with a computer, but also with SBCs (such as the Raspberry Pi) and MCUs, for instance.

controller Unless specified otherwise, refers to the dynamixel servos built-in controller.

1 Basic concepts

1.1 Best practice

The very first thing you should do before even starting to write the shortest piece of code is read the documentation of the motor you will be using (for instance the XM430-W210). This is actually probably the most important thing in the whole process of using the motors as the control table of one motor is most likely not the same as the one of another. The control table description includes names, descriptions, addresses, sizes, range, units and other useful information on the different items you can -and will have to- manipulate.

1.2 The 'vital' functions

Here I will describe the functions necessary for your programme to work, or at least not crash on starting.

portHandler and packetHandler These two functions are *the two functions* that should always be present right at the beginning of your code. They initialize the necessary structures for the communication between the hardware running the programme and the servos.

openPort and setBaudrate These functions should be called right after *portHandler* as it sets the communication speed between the computer and the dynamixel controller

groupSyncWrite and groupSyncRead These functions are useful when using multiple servos at once, if you manipulate the same item on all motors. They also serve to initialize communication with the motors, but with a different packet structure.

groupBulkWrite and groupBulkRead These functions are almost the same as the previous ones but allow for manipulating different items on each dynamixel.

closePort This function should be called at the end of the programme (I don't know what would happen if you do not call it though)

1.3 Common functions

Here are some of the most often used functions.

writeNByteTxRx This is actually a family of functions to write N bytes of data to a given address in the dynamixel. In a normal usage, N should be equal to 1, 2 or 4. There is also a TxOnly variant, though I did not have the opportunity to see it in use, so I don't know the differences.

readNByteTxRx Family of functions used to read N bytes of data from a given address.

groupSync/BulkWrite functions Family of functions to build instruction packets to write in the memory of the controller and send them. As there are quite a few and that they are more complicated to use, I will explain them further in the next section.

groupSync/BulkRead functions Family of functions to build instruction packets to read from the memory of the controller, send them and retrieve the data sent back by the servo. As with the previous write functions, I will explain them further in the next section.

2 Functions syntax

I will here describe in more details what the functions take as arguments and what they return (in C).

Main functions

```
int portHandler(const char* device_name)
```

Takes a string containing the name of the device (for instance COMx for windows or /dev/ttyUSBx for linux (x stands for a number)) and returns an int usually called port_num which is used in other functions.

```
void packetHandler()
```

Does not require arguments and returns nothing, as it only creates structures.

```
uint8_t openPort(int port_num)
```

Takes the port_num returned by portHandler and opens the port associated with it. Returns an uint8_t representing the success of the operation (true stands for success).

```
uint8_t setBaudRate(int port_num, const int baudrate)
```

Takes the port_num returned by portHandler and sets the baudrate according to the second parameter.

```
void closePort(int port_num)
```

Closes the port referred to by port_num.

```
uintx_t readNByteTxRx(int port_num, int protocol_version, uint8_t id, uint16_t address)
```

Read N bytes of data from the memory of the servo referred to by *id* starting from *address*, using protocol *protocol_version* on port *port_num* (x should fit the size, usually x=8*N).

```
void writeNByteTxRx(int port_num, int protocol_version, uint8_t id, uint16_t address, uintx_t data)
```

Write N bytes of data to the memory of the servo referred to by *id* starting from *address*, using protocol *protocol_version* on port *port_num* (x should fit the size, usually x=8*N).

Sync functions

```
int groupSyncWrite(int port_num, int protocol_version, uint16_t start_address, uint16_t data_length)
```

Initiates the sync communication process by stating the address of the control item in memory, its length, the protocol version used and the port in use. Returns *group_num*, which will be used in the sync functions (its the same as *port_num* in that respect).

```
uint8_t groupSyncWriteAddParam(int group_num, uint8_t id, uint32_t data, uint16_t input_length)
```

Store the value *data*, of actual length *input_length* to be written in the memory of the motor referenced by *id*, using the *group_num* parameters. Returns true or false (if wrong parameters).

```
void groupSyncWriteRemoveParam(int group_num, uint8_t id)
```

Remove the data and the ID of the dynamixel referenced by *id* from the structures of *group_num*.

```
void groupSyncWriteClearParam(int group_num)
```

Removes all IDs from the structures of *group_num*. Usually called after sending packet or if you want to cancel.

```
void groupSyncWriteTxPacket(int group_num)
```

Sends packet built with the data contained in the structures of *group_num*.

`int groupSyncRead(int port_num, int protocol_version, uint16_t start_address, uint16_t data_length)`

Initialises structures used to read *data_length* bytes from the memory of dynamixel motors starting from *start_address*, using *protocol_version* on port *port_num*. Returns *group_num* (not the same as the one for writing).

`uint8_t groupSyncReadAddParam(int group_num, uint8_t id)`

Adds *id* to the list of IDs to read from. Returns true or false.

`void groupSyncReadRemoveParam(int group_num, uint8_t id)`

Removes *id* from the list of IDs to read from.

`void groupSyncReadClearParam(int group_num)`

Erases the list of IDs to read from.

`int groupSyncReadTxRxPacket(int group_num)`

Asks the dynamixels listed to send back the data for which *group_num* is instantiated.

`uint8_t groupSyncReadIsAvailable(int group_num, uint8_t id, uint16_t address, uint16_t data_length)`

Checks if the data *data_length* bytes long from *address* for dynamixel *id* has been received.

`uint32_t groupSyncReadGetData(int group_num, uint8_t id, uint16_t address, uint16_t data_length)`

Extracts the data from the packet sent back by the motor.

Bulk functions

`int groupBulkWrite(int port_num, int protocol_version)`

Like groupSyncWrite, creates new structures for building and sending packets but it does not take any argument besides *port_num* and *protocol_version*, as you can write different items for each motor. Again, it returns a *group_num* to use in other functions.

`uint8_t groupBulkWriteAddParam(int group_num, uint8_t id, uint16_t start_address,`

`uint16_t data_length, uint32_t data, uint16_t input_length)`

Similar to groupSyncWriteAddParam, but as I mentioned just now, here you have to precise the item you want to write to with *start_address* and *data_length*.

`void groupBulkWriteRemoveParam(int group_num, uint8_t id)`

Same as groupSyncWriteRemoveParam, removes *id* and associated data from the structures of *group_num*.

`void groupBulkWriteClearParam(int group_num)`

Again, it's the same as groupSyncWriteClearParam, removes all IDs and data from structures of *group_num*.

`int groupBulkWriteTxPacket(int group_num)`

Same as groupSyncWriteTxPacket, builds and sends the instruction packet for *group_num*.

`int groupBulkRead(int port_num, int protocol_version)`

Similar to groupBulkWrite, but returns a different *group_num*.

`uint8_t groupBulkReadAddParam(int group_num, uint8_t id, uint16_t start_address, uint16_t data_length)`

Again, similar to groupSyncReadAddParam, but you can here assign a different target item to each motor.

```
void groupBulkReadRemoveParam(int group_num, uint8_t id)
```

Removes *id* from the structures of *group_num*.

```
void groupBulkReadClearParam(int group_num)
```

Removes all IDs from the structures of *group_num*.

```
int groupBulkReadTxRxPacket(int group_num)
```

Like groupSyncReadTxRxPacket, polls the motors and waits for the returned packet.

```
uint8_t groupBulkReadIsAvailable(int group_num, uint8_t id, uint16_t address, uint16_t data_length)
```

Checks whether the data from *id* has been received. As you can assign different item for each motor, you here have to give the parameters of the targeted item again (*address* and *data_length*).

```
uint32_t groupBulkReadGetData(int group_num, uint8_t id, uint16_t address, uint16_t data_length)
```

Extracts the data received from *id*. As for groupBulkReadIsAvailable, you have to precise the *address* and *data_length* yet again.

3 Examples

I will now give you some examples of how to use the different functions in C. Keep in mind that these examples are minimal and conceptual, only here to give you an idea of how to implement the functions, so you can't copy and paste it into your own code and they do not provide any safety against exceptions, errors, bad arguments and whatnot.

3.1 Simple read and write

```c
#include "dynamixel_sdk.h" //contains header files for all dynamixel source files (includes
↪ other headers)

#define BAUDRATE ...
#define PROTOCOL 2.0
#define DEVICE_NAME /dev/ttyUSB0 //default for Linux-based OS, COM1 would be for windows,
↪ for instance
#define ID ...
#define TORQUE_ADDR 64 //cf control table for motor on ROBOTIS e-Manual
#define GOAL_POS_ADDR 116
#define GOAL_POS_LEN 4
#define PRES_POS_ADDR 132
#define PRES_POS_LEN 4

int main(int argc,char** argv)
{
    int port_num = portHandler(DEVICE_NAME); //as I said, should be the first function to be
    ↪ called
    packetHandler(); //followed by this

    openPort(port_num);
    setBaudRate(port_num, BAUDRATE); //and then this. Be careful to set BAUDRATE according
    ↪ to the value in the control table : if you set BAUDRATE to 100000 and the dynamixel
    ↪ is configured with Baud Rate 1 (i.e. 57600), which is the default value for some
    ↪ motors, you won't be able to communicate properly

    int32_t goal_pos,pos; //you should determine the type of your variables according to the
    ↪ control table (for instance, Goal Position and Present Position are coded on 4 bytes
    ↪ (hence 32 bits), and can take both positive or negative values, thus you should use
    ↪ an int32 (signed integer on 32 bits)

    write1ByteTxRx(port_num, PROTOCOL, ID, TORQUE_ADDR, 1); //this would write 1 (true) in
    ↪ the Torque Enable register in the memory of the controller, thus enabling torque

    write4ByteTxRx(port_num, PROTOCOL, ID, GOAL_POS_ADDR, goal_pos);

    pos=read4ByteTxRx(port_num, PROTOCOL, ID, PRES_POS_ADDR); //reads 4 bytes from memory
    ↪ beginning from the Present Position address

    printf("%d: Present position: %d",ID,pos);
    return 0;
}
```

3.2 Sync read and write

```c
#include "dynamixel_sdk.h"

#define BAUDRATE ...
```

```
#define PROTOCOL 2.0
#define DEVICE_NAME ...
#define ID1 ...
#define ID2 ...
#define TORQUE_ADDR 64
#define GOAL_POS_ADDR 116
#define PRES_POS_ADDR 132

int main(int argc,char** argv)
{
    int port_num=portHandler(DEVICE_NAME);
    packetHandler();
    openPort(port_num);
    setBaudRate(port_num, BAUDRATE);

    int groupwrite_num = groupSyncWrite(port_num, PROTOCOL, GOAL_POS_ADDR, GOAL_POS_LEN);
    int groupread_num = groupSyncRead(port_num, PROTOCOL, PRES_POS_ADDR, PRES_POS_LEN);

    int32_t goal_pos[2];
    int32_t pos[2];

    write1ByteTxRx(port_num, PROTOCOL, ID1, TORQUE_ADDR, 1); //you can still use functions
    ↪  for a single motor to interact with a motor in the chain
    write1ByteTxRx(port_num, PROTOCOL, ID2, TORQUE_ADDR, 1);

    groupSyncReadAddParam(groupread_num, ID1); //this will store the IDs of the motors you
    ↪  want to retrieve data from
    groupSyncReadAddParam(groupread_num, ID2);

    groupSyncWriteAddParam(groupwrite_num, ID1, goal_pos[0], GOAL_POS_LEN); //this will
    ↪  store the parameters for building the instruction packet
    groupSyncWriteAddParam(groupwrite_num, ID2, goal_pos[1], GOAL_POS_LEN);
    groupSyncWriteTxPacket(groupwrite_num); //sends the instruction packet
    groupSyncWriteClearParam(groupwrite_num); //erase the stored parameters

    groupSyncReadTxRxPacket(groupread_num); //polls motors
    int res=groupSyncReadIsAvailable(groupread_num, ID1, PRES_POS_ADDR, PRES_POS_LEN);
    ↪  //checks if packet with the data has been received
    if(res)
    {
        pos[0]=groupSyncReadGetData(groupread_num, ID1, PRES_POS_ADDR, PRES_POS_LEN);
        ↪  //extracts data from received packet
    }
    return 0;
}
```

3.3 Bulk read and write

```
#include "dynamixel_sdk.h"

#define BAUDRATE ...
#define PROTOCOL 2.0
#define DEVICE_NAME ...
#define ID1 ...
#define ID2 ...
#define TORQUE_ADDR 64
#define GOAL_POS_ADDR 116
```

```c
#define PRES_POS_ADDR 132
#define GOAL_CUR_ADDR 102
#define GOAL_CUR_LEN 2
#define PRES_CUR_ADDR 126
#define PRES_CUR_LEN 2

int main(int argc,char** argv)
{
    int port_num=portHandler(DEVICE_NAME);
    packetHandler();
    openPort(port_num);
    setBaudRate(port_num, BAUDRATE);

    int groupwrite_num = groupBulkWrite(port_num, PROTOCOL);
    int groupread_num = groupBulkRead(port_num, PROTOCOL);

    int32_t goal_pos,pos;
    int16_t goal_cur,cur;

    write1ByteTxRx(port_num, PROTOCOL, ID1, TORQUE_ADDR, 1); //you can still use functions for a single motor to interact with a motor in the chain
    write1ByteTxRx(port_num, PROTOCOL, ID2, TORQUE_ADDR, 1);

    groupBulkReadAddParam(groupread_num, ID1, PRES_POS_ADDR, PRES_POS_LEN); //this will store the IDs of the motors and what data you want to retrieve from each of them
    groupBulkReadAddParam(groupread_num, ID2, PRES_CUR_ADDR, PRES_CUR_LEN);

    groupBulkWriteAddParam(groupwrite_num, ID1, GOAL_POS_ADDR, GOAL_POS_LEN, goal_pos, GOAL_POS_LEN); //this will store the parameters for building the instruction packet
    groupBulkWriteAddParam(groupwrite_num, ID2, GOAL_CUR_ADDR, GOAL_CUR_LEN, goal_cur, GOAL_CUR_LEN);
    groupBulkWriteTxPacket(groupwrite_num); //sends the instruction packet
    groupBulkWriteClearParam(groupwrite_num); //erase the stored parameters

    groupBulkReadTxRxPacket(groupread_num); //polls motors
    int res1=groupBulkReadIsAvailable(groupread_num, ID1, PRES_POS_ADDR, PRES_POS_LEN); //checks if packet with the data has been received
    int res2=groupBulkReadIsAvailable(groupread_num, ID2, PRES_CUR_ADDR, PRES_CUR_LEN);
    if(res1 && res2)
    {
        pos=groupBulkReadGetData(groupread_num, ID1, PRES_POS_ADDR, PRES_POS_LEN); //extracts data from received packet
        cur=groupBulkReadGetData(groupread_num, ID2, PRES_CUR_ADDR, PRES_CUR_LEN);
    }
    return 0;
}
```

4 Troubleshooting

4.1 Beginner's mistakes

This is not an exhaustive list of the mistakes you can make while using dynamixel motors, but it should be a good start to check if you haven't made one of the mistake listed here if you ever have issues while using them.

Permissions You should make sure you can read and write on the port you're using.

ID This is probably the easiest thing to forget, but also the most crucial, so always check the IDs of the dynamixels. This includes not giving the same ID to several motors.

Baudrate Also an easy mistake to make, getting the wrong baudrate means you won't communicate properly with the motors, sometimes you won't even detect them.

Wrong control table item When manipulating items for the first time (actually even if it's not the first) you should double-check the parameters (length, address, and such) so they are written/read properly. You should also check the type of the variables you use in combinations with the items (for instance, don't always use unsigned integers for position related items, as those can contain negative values).

4.2 Advanced issues

I will list here issues that are not down to a lack of focus from the user. This list is meant to expand as I try out new things.

USB latency This is a tricky issue, as you have to change both the USB latency in the dynamixel SDK source code and the one registered in your OS. ROBOTIS advise to change it while using the bulk/sync methods with a number of motors. I'd advise checking out the *port_handler* files in the dynamixel SDK (in C, the full name of the file is *port_handler_[OS].c*).