DXLite - Dynamixel System Library (Light Version) User Guide

Gary Lee

September 2015

Git repository can be found in `https://github.com/KM-RoBoTa/DXLite.git`.

This is the light version of DXSystem, which is composed of less files and functions. For extended capabilities, please check DXSystem. (This guide is adapted from the DXSystem's user guide.)

Contents

1 About Dynamixel Resources

1.1 Dynamixel Motors

1.1.1 Communication - Half-Duplex

Dynamixel motors use half duplex serial communication, where data can be transmitted in both directions along the same wire. This protocol has the advantage of communicating with multiple devices through a single bus. However, this means that data cannot be transmitted and received simultaneously, and that only one device can transmit signals at a time. Understanding this limitation is crucial so as to avoid timeout and corruption when communicating with Dynamixel motors.

1.1.2 Communication - RS485 and TTL

There are two main serial communication protocols that dynamixel motors use, depending on the motor model - RS485 (4-pin) and TTL (3-pin). Half duplex operation is used for both communications (i.e. native TTL/UART on single-board computers and microcontrollers cannot directly communicate with Dynamixel motors without an intermediate component or a mutex).

1.1.3 Fine Motor Control - PID and Compliance Slope/Margin

Different dynamixel motor models possess different fine control mechanisms. MX-series motors generally possess closed loop controls (PID), while RX-28 and AX-12 rely on compliance margin and slope settings to control the flexibility of motors. Definitions of these terms can be found in the documentations of the respective motor models.

1.2 Dynamixel SDK

The Dynamixel SDK/API is available for Windows and Linux platforms. The SDK is written in C (although the Windows SDK is designed for use with Visual Studio). The SDK can also be used for other unsupported platforms by setting up the platform-dependent source (`dxl_hal.c/.h` files); the `dynamixel.c/.h` file is platform independent.

1.2.1 Control Table - EEPROM and RAM

Each motor possesses a control table that contains data pertaining to its status and operations. The control table may vary for different models, where some addresses (corresponding to certain functions) may be missing or may define different parameters. Notably, MX series motors have additional registers (68 and above) not found in the AX and RX motors.
In our interface (as will be further described later), the control table is coded in the `control_table_constants.h` header file for easy reference and access.

2 Changes to SDK/API

2.1 Object-Oriented Modifications - DXLHAL and dynamixelClass

The USB2Dynamixel SDK for Linux platforms was used as the basis for this work. However, as the SDK uses global variables when connecting to the device (in dxl_hal.c), the program is limited to 1 USB-to-Dynamixel device. To isolate these settings for individual ports, the functions of the SDK has been converted into methods of a class. Hence, an instance for each serial port corresponds to its own object, and there is no interference between the different ports. All other features are the same. However, as will be expounded later, a higher abstraction layer that introduces newly implemented classes and functions has been introduced. Hence, direct access into the object-oriented variant of this SDK is not necessary.

2.2 Queue Bytes in Dynamixel SDK

The SDK provided supports reading and writing bytes to the motor's control table. However, queueing instructions, which allows for (almost) simultaneous transmission to motors, is not provided by the SDK. The functionality to queue bytes and words has been added, in a similar fashion to reading and writing. The execute function, which is the signal to transmit all queued instructions, has also been introduced.

2.3 Changes to Writing Bytes for Speed

Writing bytes to the Dynamixel motors can be achieved by using the instructions INST_WRITE and INST_SYNC_WRITE. The SDK recommends using INST_WRITE for writing to a single motor and INST_SYNC_WRITE for simultaneous control to multiple motor. However, analysis of execution time (in C++) revealed that the time taken to process INST_SYNC_WRITE is significantly faster than INST_WRITE (15 μs vs 1-5 ms). Additionally, INST_SYNC_WRITE is also capable of communicating with only 1 motor.
The main drawback of INST_SYNC_WRITE is the lack of a status return packet, which would return information such as the error bit on the individual motors. Nevertheless, noting that the error bits returned is usually 0 (for a well-built and well-designed system/program) and that they can also be received through the read command as part of a closed feedback loop, the status return packet from writing is not essential. Given these information, it seems that the use of INST_SYNC_WRITE is more advantageous than INST_WRITE, even when writing to a single motor. Hence, the Dynamixel SDK has been modified to replace INST_WRITE with INST_SYNC_WRITE. (On this note, it is generally recommended to perform a read function after writing to obtain the status packet.)

2.4 Implementing Bulk Read for MX Motors

In the latest update to the Dynamixel communication protocol, the instruction INST_BULK_READ has been introduced to the MX-series motors. This instruction allows for reading bytes of multiple motors through a single instruction, minimizing the amount of overhead and packet transmission. The use of bulk read has been incorporated into our modified SDK. However, INST_BULK_READ is only recognized by MX-series motors. To ensure compatibility with non-MX models, the bulk read function within the modified SDK has different behaviors depending on the motor model detected. For the MX-series, the corresponding motor IDs will be incorporated into the transmission packet as part of INST_BULK_READ. For all other motor models, the function is equivalent to sequential INST_READ. The function is also able to handle a system comprising a mixture of MX and non-MX motors.
To simplify the behavior and use of bulk read, we limited it to reading the same address(es) across all motors. The INST_BULK_READ allows for reading different addresses for different motors (e.g. reading position from Motor 1 and speed from Motor 2), but not multiple, non-consecutive addresses on a single motor (e.g. voltage and position from a single motor). The INST_BULK_READ is also not limited to reading from all motors within the system (i.e. selective reading is possible). But since we are generally interested in information on the same parameter across all motors within a closed feedback loop, the above simplification is justified. The bulk read function allows for reading a single byte or two consecutive bytes (i.e. a word). (Reading four consecutive bytes (i.e. two words) has also been implemented, where the information returned is a concatenation of two vectors, each containing information of a two-byte parameter (e.g. position and speed, which are two-byte parameters located next to each other on the control table). However, this feature is subsequently not used, to simplify the functions and features of the library.)

3 Library with New Classes

3.1 Higher Abstraction Layer

While the USB2Dynamixel SDK helps encapsulate the implementation details between the controller and the firmware of the motors, it still requires technical knowledge of the control table and specifications of the motor. The goal of this work is to create a higher abstraction layer that directly implements common functions of the motor, such as configuring the system and setting its position and speed.

3.2 Hierarchy

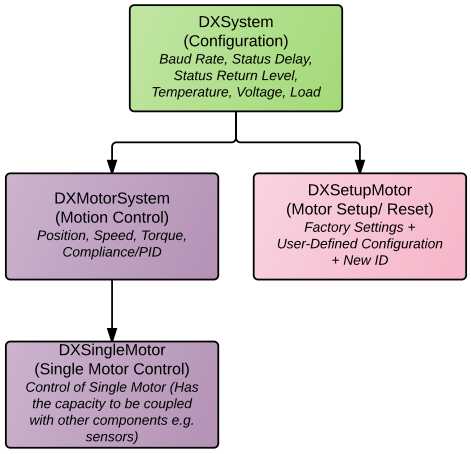

Figure 1: Hierarchy of the classes introduced in the new SDK

In the new SDK, the new classes introduced are:

- `DXSystem` (Superclass) - for basic configuration of the system of motors; contains an object corresponding to the modified USB2Dynamixel SDK
- `DXMotorSystem` (Subclass of `DXSystem`) - adds functions for common motor controls (position, speed and torque control)
- `DXSingleMotor` (Subclass of `DXMotorSystem`) - limits the functions on `DXMotorSystem` to a single motor
- `DXSetupMotor` (Subclass of `DXSystem`) - for resetting and reconfiguring motors to project specifications

3.3 Considerations and Limitations

The functions were developed and tested for the following motors - MX-106, MX-64, MX-28, RX-28, RX-24F and AX-12A. (other motor models are also supported, but not extensively tested.) Newer Dynamixel motors may be used, on the condition that their control table formats does not vary from either the AX/RX-series or the MX-series motors. (E.g., if the control table addresses corresponding to position and speed are the same, functions controlling position and speed can be used. In particular, certain MX motors may have the order of PID registers on the control table swapped, which may result in erroneous behavior.) However, the specifications for these motors must be updated in the `DModelDetails` files.
The functions were also developed and tested for Joint Mode. While alternative modes (Wheel Mode and Multi-Turn Mode) has been taken into account, further tests are required to ensure robustness.
Attempts were made to optimize the functions for memory usage and efficiency. For most settings (with the exception of some parameters such as baud rate, status return level and status delay time, as will be touched on in Section 4), the user can choose between writing to a single motor or writing to all motors. When writing to a large number of motors, writing to all would be more efficient (on the communication level) than looping to write on the motors individually.

3.4 Other Functions and Headers

3.4.1 DModelDetails - Hard-Coding Specifications of Motors

Functions in `DModelDetails` provide details on the specifications for the respective motor models. Most of the motor models have been coded in the respective files. To establish support for other Dynamixel motors, the functions within `DModelDetails` have to be updated with the specifications of the respective model.

3.4.2 Control Table Constants - Macros for Control Table Addresses

The header file `control_table_constants.h` contains macros that matches the control table attributes to their respective numerical addresses. As such, the different addresses on the control table can be accessed using the symbolic name, omitting the need for technical knowledge of the control table addresses. The addresses contain the prefix "P_" (following the convention put forth within the sample code of the Dynamixel SDK). These macros should correspond to the mapping on the motor's control table and hence should not be changed.
Some addresses may be used in more than one macro, since different motors may use the same address number for different attributes. Furthermore, some of the attributes may not exist for certain motor models (e.g. AX and RX motors do not have PID control). As the header file does not check for motor model, the addresses should be called properly. (Using functions in `DModelDefailts` to check for capabilities, such as PID/compliance settings, is strongly encouraged.) In the event that the control table has been changed in newer Dynamixel motor models, it is recommended to include the mappings in this file, but not delete any of the previous entry to ensure backward compatibility.

3.4.3 DConstants - User-Defined Macros for Motor Settings

The header file `dconstants.h` defines "constants" (as macros) that may be frequently used. Some of the constants are states, i.e. constants with symbolic names defined with an arbitrary number, for use as settings and logical comparisons.
None of these constants should be changed. In the situation that new constants need to be introduced, users are advised to add them to the file `src/user_settings.h` instead.

4 Main Functions

(To demonstrate the use of functions, the variable `d` is used to denote a `DXMotorSystem` object, `ds` to denote a `DXSingleMotor` object and `dsetup` to denote a `DXSetupMotor` object. The superclass `DXSystem` would hardly be used, since its subclass inherits almost all its functions.)

4.1 Configuring with DXSystem (superclass)

The `DXSystem` possesses functions within the following groups (under modules in the complete Doxygen documentation) - motor status getters, motor configuration settings, motor reference information and manual adjustments.
Its subclasses, `DXMotorSystem`, `DXSetupMotor` and `DXSingleMotor`, inherit `DXSystem` functions. (However, some functions are protected and hence inaccessible for `DXSingleMotor` objects.)
As the capabilities of `DXSystem` is limited, instantiating a `DXSystem` object is generally discouraged. Its subclasses should be used instead.

4.1.1 Motor Status Getters

These functions can be used to obtain the current load, temperature and voltage of a motor in the system.

```
int load = d.getLoad(motor_number);
int temperature = d.getTemperature(motor_number);
int voltage = d.getVoltage(motor_number);
```

(The returned values are the digital values as encoded in the control table.)
Specifically for load, we can also read this parameter across all motors.

```
vector<int> all_loads = d.getAllLoad();
```

(Note: The motor's 'load' may not necessarily be proportional to the force/torque on the motor. In its implementation, it is loosely related to the current consumption and the differentce between goal and current position on the motor. Further characterization is required to determine the relation between this parameter and the 'true load' on the motor.)

4.1.2 Motor Configuration Settings

These functions can be used to set or read the configuration settings (for temperature and voltage operating limits, status return level and delay, and baud rate) on the motors.
Maximum Temperature:
(The recommended max temperature is 70-80°C, depending on the material of the motor body, and should not be changed.)

```
d.setMaxTemperature(motor_number, max_temperature);
int max_temperature = d.getMaxTemperature(motor_number);
```

Voltage Limits:

```
d.setVoltageLimits(motor_number, volt_lower_limit, volt_upper_limit);
int volt_lower_limit = d.getMinVoltage(motor_number);
int volt_upper_limit = d.getMaxVoltage(motor_number);
```

Set all

```
d.setAllMaxTemperature(v_max_temperature);
d.setAllVoltageLimits(v_volt_lower_limit, v_volt_upper_limit);
```

System Delay Time, Status Return Level and Baud Rate are also settings that can be changed, but the changes have to be made across all motors. (These settings should be thought of as global across all motors, as differing settings may cause unpredictable behavior.)
System Delay Time (the amount of time delay between instruction packet transmitted and status packet received):

```
d.setAllDelay(delay_time);
int delay_time = d.getDelay(motor_number);
```

Status Return Level (corresponding to the settings for when status packet is given):

```
d.setAllStatusReturnLevel(SRL);
int SRL = d.getStatusReturnLevel(motor_number);
```

Baud Rate:
(When baud rate is changed, the port is automatically reopened at the new baud rate by the function.)

```
d.setAllBaudRate(baud_rate);
int baud_rate = d.getBaudRate(motor_number);
```

4.1.3 Motor Referencing Information

The following functions may be used to obtain the model and native ID (on the control table) of individual motors, and the total number of motors detected on the port.

```
int motor_model = d.getModel(motor_number);
int num_motors = d.getNumMotors();
int dxl_ID = d.motorNum2id(motor_number);
int motor_number = d.id2motorNum(dxl_ID);
```

Note: In the implementation of the library, we abstracted the use of ID (on the registers) and replaced it with motor number for referencing (i.e. always starting from 0, and enumerated in increasing order of the IDs detected; doing this reduces the need for users to store their IDs.)

4.1.4 Other Generic System Functions

These functions are mainly used for initializing and checking the `DXSystem` object.
Upon creating a `DXSystem` object, it is possible to check if the system has been initialized successfully using the following function

```
if (d.isInitialized()) ...
```

During setup, if the baud rate has changed, communication with the motors will be disrupted. To re-establish the communication, the device should be re-opened with the new baud rate set. The following function facilitates the re-opening and initializing of the port at the new baud rate

```
d.reopenPort(new_baud_rate);
```

Changing the baud rate (using `setBaudRate`) would also automatically call `reopenPort`.
Upon the end of motor setup, the EEPROM can be locked to prevent any further changes to the settings. However, upon locking, the EEPROM can only be unlocked and changed after restarting the system (i.e. unplug and replug the power supply to the motors). Locking EEPROM of all motors can be done with the following function.

```
d.lockEEPROM();
```

For troubleshooting, the motor number of respective motors has to be known. The following function runs a sequence that helps to identify the motor numbers by lighting up the LEDs in turn. The sequence is executed on the console, and the user would press 'Enter' to go through the sequence.

```
d.identifyMotors();
```

This function checks if a specific motor possesses information that is queued (or registered, on the control table).

```
if (d.isQueued(motor_number)) ...
```

4.1.5 Manual Adjustments

For advanced users who seek manual access to the control table, functions to read, write and queue data to the control table are provided. These functions require knowledge of the control table and the corresponding limits and valid inputs, since there are no checking mechanisms introduced to these functions within the SDK.
Definitions:

- Byte: corresponds to the value within a single address

- Word: composed of two bytes (low and high), corresponds to two consecutive addresses, starting with the low byte. (Dynamixel SDK provides functions like

```
int word = DXLObject.dxl_makeword(lowbyte, highbyte);
int lowbyte = DXLObject.dxl_get_lowbyte(word);
int highbyte = DXLObject.dxl_get_highbyte(word);
```

  which are useful for converting words to bytes and vice versa.)

- 2Word: composed of two words, or four bytes, corresponds to four consecutive addresses. This is specifically for controlling two settings, each corresponding to two bytes, and are located next to each other on the control table (e.g. speed and position). This function is provided for advanced users who wish to maximize communication efficiency.

(For the subsequent functions, `P_ADDRESS` and `P_ADDRESS_L` refer to the macro defined on `control_table_constants.h`. For words, `P_ADDRESS_L` is the address of the parameter's lower byte, which is also suffixed by an `_L`.)
Reading:

```
int byte = d.readByte(motor_number, P_ADDRESS);
int word = d.readWord(motor_number, P_ADDRESS_L);
```

Writing:

```
d.setByte(motor_number, P_ADDRESS, value);
d.setWord(motor_number, P_ADDRESS_L, value);
d.set2Word(motor_number, P_ADDRESS1_L, value1, value2);
```

Reading from All Motors:

```
// Bulk read implemented
vector<int> allBytes = d.readAllByte(P_ADDRESS);
vector<int> allWords = d.readAllWord(P_ADDRESS_L);
```

Writing to All Motors:

```
d.setAllByte(P_ADDRESS, vector_of_values);
d.setAllWord(P_ADDRESS_L, vector_of_values);
d.setAll2Word(P_ADDRESS1_L, vector_of_values1, vector_of_values2);
```

Queuing:

```
d.queueByte(motor_number, P_ADDRESS, value);
d.queueWord(motor_number, P_ADDRESS_L, value);
d.queue2Word(motor_number, P_ADDRESS1_L, value1, value2);
d.executeQueue(); // Write queued values
```

4.2 Controlling with DXMotorSystem

The `DXMotorSystem` class is the main class to use to control a sequence of motors connected to a single USB device.
A pointer to a `DXMotorSystem` object can be instantiated by

```
DXMotorSystem *d_pointer = new DXMotorSystem(int devicePort, int dxlBaudRate)
```

where `devicePort` refers to the serial number corresponding to the USB port (dev /.ttyUSB* where * is typically 0, 1, 2...) and `dxlBaudRate` refers to the data value corresponding to the desired baud rate (`DEFAULT_BAUDRATE`, which is typically 1 – 10 Mbps). The `DXMotorSystem` can access all the public functions of `DXSystem` as listed above, in addition to functions that control motor positions, speeds and torque controls.

4.2.1 Motor Motion Limit Settings

These functions are mainly to handle hard limits and settings for the motors.
Position Limits:

```
d.setPositionLimits(motor_number, lower_limit, upper_limit);
d.setAllPositionLimits(v_lower_limit, vector_upper_limit);
int lower_limit = d.getPositionLowerLimit(motor_number);
int upper_limit = d.getPositionUpperLimit(motor_number);
```

Home Position:
(For internal records to what 'default' positions should be. Upon initialization, the positions detected would be saved as home until the `setAllHomePosition` is called.)

```
d.setAllHomePosition(home_positions);
vector<int> homePosition = d.getAllHomePosition();
```

Torque Maximum Limit:

```
d.setTorqueMax(motor_number, upper_limit);
d.setAllTorqueMax(v_upper_limit);
int upper_limit = d.getTorqueMax(motor_number);
```

Torque Enable Settings:

```
d.setTorqueEn(motor_number, enable_boolean);
d.setAllTorqueEn(enable_boolean);
boolean enable = d.getTorqueEn(motor_number);
```

4.2.2 Position and Speed Control

The position and speed of motors can be controlled using the following functions.
Position:

```
d.setPosition(motor_number, goal_position);
d.movetoPosition(motor_number, goal_position); // set and wait until motor stops
d.queuePosition(motor_number, goal_position);
d.setAllPosition(v_goal_position);
int current_position = d.getCurrentPosition(motor_number);
vector<int> positions = d.getAllPosition();
```

Speed:

```
d.setSpeed(motor_number, goal_speed);
d.queueSpeed(motor_number, goal_speed);
d.setAllSpeed(v_goal_speed);
int current_speed = d.getCurrentSpeed(motor_number);
vector<int> current_speeds = d.getAllCurrentSpeed();
int current_set_speed = d.getSetSpeed(motor_number);
vector<int> set_speeds = d.getAllSetSpeed();
```

Other Functions

```
bool is_moving = d.isMoving(motor_number); // Motor is still moving
d.resetMovingSettings(motor_number); // Reset to 0 speed and 'home' position
d.resetAllMovingSettings(); // Reset all motors to 0 speed and 'home' position
```

4.2.3 Fine Motor Control through Compliance/PID Settings

The following setter and getter functions are for finer and more complex control of motor movements.
Punch (Current to drive motor):

```
d.setPunch(motor_number, value);
d.setAllPunch(v_values);
int punch_value = d.getPunch(motor_number);
```

For the MX-series motors, closed loop feedback using PID scheme can be achieved. The following functions can help facilitate the control of the PID loop.
PID:

```
d.setGainP(motor_number, p_value);
d.setAllGainP(v_p_values);
int p_value = d.getGainP(motor_number);
d.setGainI(motor_number, i_value);
d.setAllGainI(v_i_values);
int i_value = d.getGainI(motor_number);
d.setGainD(motor_number, d_value);
d.setAllGainD(v_d_values);
int d_value = d.getGainD(motor_number);
```

For other motors, compliance margins and slopes are used for flexible control of the motors.
Compliance Control:

```
d.setComplianceSlope(motor_number, cw_slope, ccw_slope);
d.setAllComplianceSlope(v_cw_slope, v_ccw_slope);
int cw_slope = d.getComplianceSlopeCW(motor_number);
int ccw_slope = d.getComplianceSlopeCCW(motor_number);
d.setComplianceMargin(motor_number, cw_margin, ccw_margin);
d.setAllComplianceMargin(v_cw_margin, v_ccw_margin);
int cw_margin = d.getComplianceMarginCW(motor_number);
int ccw_margin = d.getComplianceMarginCCW(motor_number);
```

4.3 Controlling with DXSingleMotor

`DXSingleMotor` is a subclass of `DXMotorSystem`, and hence inherits `DXMotorSystem` functions. The object can only hold a single motor within the system; otherwise, an exception is thrown.
A `DXSingleMotor` object can be instantiated by

```
DSingleMotor ds = new DSingleMotor(int devicePort, int dxlBaudRate)
```

As `DXSingleMotor` is a special case of `DXMotorSystem` with only 1 motor, some changes have been introduced to all the functions. Notably, all functions that are meant to write to "all motors" are inaccessible. Additionally, all setter and getter functions no longer require the `motor_number`, as it is default to the value 0 for such a system (with only 1 motor). The definitions and purposes of the functions largely remain unchanged.
This class was designed with the intention of creating a complex module comprising a single motor and other components like sensors. An object class for such a module can be created by extending the `DXSingleMotor` class and include the functions for other sensors in use.

4.4 Setting up with `DXSetupMotor`

`DXSetupMotor` extends `DXSystem` as a class that aids in motor setup and reset. This class can only handle one motor at a time to prevent conflict in motor IDs. (When two motors with the same ID are connected, the behavior of the communication between the system and the motors becomes unpredictable.)
Motor setup can be done through instantiating a `DXSetupMotor` object by

```
DXMotorSetup dsetup = new DXMotorSetup(int newID, int devicePort, int dxlBaudRate)
```

where newID should be a unique number between 0 to 253 for each individual motor. The constructor resets the motor to factory settings before writing user-defined default settings to the motor. (Note: Some manual delays have been introduced within the reset function to prevent corruption in communication. As the setup should be done prior to and separate from the actual program, these delays should not affect the performance of execution when the whole system is connected.)

5 Additional Functions

5.1 Model-Related Functions

Certain parameters, such as range of joint angles and position data or Compliance/PID capabilities, are model specific. To facilitate in encoding the specifications of different motor models, the functions within `dmodeldetails.cpp/.h` provides the information given a model number (corresponding to the value encoded in the EEPROM) as the argument. This parameter can be called from d.getModel(motorNum). As mentioned, new motor models may also be compatible as long as the control table is similar and this file is updated with the new specifications. The main functions are mentioned below.
To obtain the range of position values (minimum, maximum and middle positions):

```
int midPos = modelMiddlePosition(model);
int maxPos = modelMaxPosition(model);
int minPos = modelMinPosition(model); // generally 0
```

To obtain the angle range achievable in joint mode:

```
int angleRange = modelAngleRange(model);
```

To determine whether the motor is capable of PID/compliance settings and bulk-read:

```
bool bulkRead = bulkreadCapable(model);
bool pidMode = pidCapable(model);
bool complianceMode = complianceSettingCapable(model);
```

To determine the default baud rate upon factory reset (for `DXSetupMotor`):

```
int baudRate = modelDefaultBaudRate(model);
```

5.2 Additional Tools

Additional functions that may be useful when using the new library are defined in `additionaltools.cpp/.h`. To use these functions, include the header

```
#include "dtools/additionaltools.h"
```

There are six functions currently defined in the file, that may be useful when controlling the Dynamixel system and using the library.
Asking for inputs is common in many console-based interfaces. Two functions have been created to aid in parsing input stream.

```
int number = getInt(); // To get an integer
bool flag = getYorN(); // To get Y or N (boolean)
```

When parsing a text file, conversion of string (`std::string`) to integer numbers is required. Hence, the function `str2int` is also defined.

```
int number = str2int(string);
```

Delay functions are useful in imposing a fixed period in a loop or providing some latency within communication. The function `delay` takes in time duration in milliseconds, and will cause the current thread to sleep.

```
delay(time_in_milliseconds); // this thread sleeps
```

When writing to a filename, the use of timestamp would help users differentiate differnet files and also prevent files from being overwritten. The function `getTimeStr` provides a string with formatted time (inverse format, YYYYMMDDhhmmss)

```
string time_string = getTimeStr();
```

Lastly, with regards to exception handling, users can decide between throwing exceptions (and catching them to handle it) or simply print it to the console (`cerr`) without interrupting the program. This function should mainly be used for non-vital errors and exception. The function can be used by

```
string msg = "This␣is␣an␣error␣message";
throwPrintError(msg, throw_flag);
```

If the `throw_flag` is true, a `runtime_error` with an error message is thrown. Otherwise, the string is simply printed to the console and the program continues.

5.3 System Setup

A `DXMotorSystem` pointer can be instantiated using the pre-defined function

```
#include "dtools/setup.h"
...
DXMotorSystem* d_pointer = dxsystemSetup(devicePort); //devicePort = 0,1,2...
```

Alternatively, its object can be instantiated by de-referencing the pointer

```
DMotorSystem d = *dxsystemSetup(devicePort); //devicePort = 0,1,2...
```

Calling dxsystemSetup instantiates a DXMotorSystem object, writes user-defined default settings to all motors, performs some checks on the system and returns a pointer to the object.
The default settings used are set via the constants/macros defined in user_settings.h and instructions in the function dxlSetupEEPROM within `userinputfn.cpp` (found in `src` folder). These files may be modified to attain the desired default settings for the motors. Multiple DXMotorSystem pointers can also be instantiated using the function

```
#include "dtools/setup.h"
...
vector<DXMotorSystem*> all_d_pointers = multidxsystemSetup(num_of_ports);
// OR
vector<DXMotorSystem*> all_d_pointers = multidxsystemSetup(vector_of_port_nums);
```

This function instantiates multiple `DXMotorSystem`, using the device port numbers 0 to `num_of_ports-1` for the former case, or all the port numbers listed in the vector argument for the latter case.
The `setup.cpp/.h` also offers the function `dxlSetupSequence` which would aid in setting up the system prior to any use. This is particularly useful for newly built system, where the EEPROM settings have to be reset. This function prints the instructions on the terminal, of which the user should follow. The steps involve connecting the motors one at a time, so as to perform a factory reset, write the default configuration settings and assign a unique ID to the motor.

```
dxlSetupSequence();
// Instructions appear on terminal
```