DXSystem - Dynamixel System Library User Guide

Gary Lee

August 2015

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.
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 the error bit on the individual motors. Nevertheless, noting that the error bits returned is usually 0 (for a well-designed system and 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.

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 also allows for reading bytes from selected motors, and not limited to all motors within the system. 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, two consecutive bytes (i.e. a word) or four consecutive bytes (i.e. two words). In the case of the latter, 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).

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

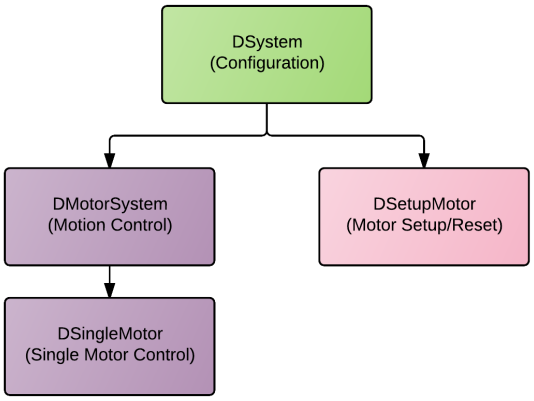

Figure 1: Hierarchy of the classes introduced in the new SDK

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 - MX106, MX64, MX28, RX28 and AX12A. (motors such as RX24F and AX18 are also supported, but not extensively tested.) Other Dynamixel motors may be used, on the condition that their control table formats does not vary from either the AX12A or 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. Currently, models MX106, RX28 and AX12A have been tested while information about other motor models have also been coded. 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. AX12A and RX28 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.)

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, `DMotorSystem`, `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.)

4.1.2 Motor Configuration Settings

These functions can be used to set or read the configuration settings 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.
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 reopened at the new baud rate.)

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

4.1.3 Motor Referencing Information

These 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);
```

4.1.4 Other Generic System Functions

These functions are mainly used for initializing and checking the DSystem object.
Upon creating a DSystem 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();
\end{lstlisting{
This function checks if a specific motor possesses information that is queued (or registered,
\begin{lstlisting}
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 register'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 to All Motors

```
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 DMotorSystem can access all the public functions of DSystem 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);
```

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 center position
d.resetAllMovingSettings(); // Reset all motors to 0 speed and center 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 DSingleMotor 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

```
DMotorSetup dsetup = new DMotorSetup(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 Sample Usage

(Some of the high level functions have been changed to allow for more parameters and flexible control. This user guide will be updated soon to include these changes.)

5.1 File Organization

The files are organized as follows

- `dxllibclass` contains the modified dynamixel SDK
- `dclass` contains all implementations of the new classes and additional resources
- `dtools` contains additional `cpp/h` files with functions that help with high-level tasks, e.g. parsing files, instantiating master/slave system and setting up the system.
- All new source files (including `main.cpp`) should be in the folder **src**. `userinputfn.cpp/.h` and `user_settings.h` allow users to change settings and behavior of the system.
- All CSV files will be read and written to the folder `data`.

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 five 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 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 dereferencing 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);
```

This function instantiates multiple `DXMotorSystem`, using the device port numbers 0 to `num_of_port-1`

5.4 Read/Parse CSV File with Joint Angle Data

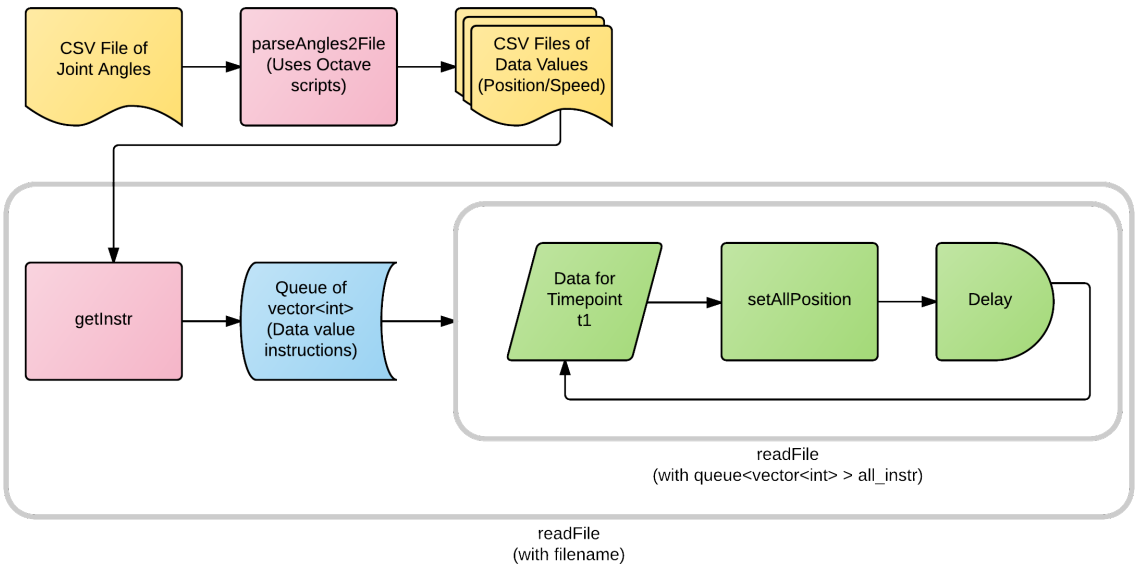

Figure 2: Read Joint Angle File Scheme

The functions to read and parse CSV files can be found in `readfile.cpp/.h` in the folder `dtools`. All CSV files are read and written from the data folder, while octave scripts are run from the run directory (in the same folder as the executable, or at the directory stipulated by the project settings). Names of the files written will be appended with the system date and time. There are several functions that may be used, depending on the format of the data within the CSV file. To use these functions, the header file has to be included.

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

The first function, dxlWriteInfo, writes some model-dependent information (max position, angle range and max speed) for every motor in the `DXMotorSystem` object into a CSV file. The function returns the name of this file.

```
dxlWriteInfo(d);
```

For the following functions, a CSV file can be read and parsed; however, the format of the data should comply with the template requirements. The CSV file should have a column and row header, with raw data starting from the second row, second column. The CSV file should also have number of columns corresponding to number of motors present + 1 (header) – each column (or group of consecutive columns, if multiple values are passed per motor) after the header corresponds to the motor number within the system.
The second function parses raw angles (encoded in degrees or radians) from a CSV file and converts them into digital position values for each of the motor, recorded within a new CSV file. The function also calls dxlWriteInfo within its definition, as the motor information is required for the conversion. The function also returns the name of the written file.

```
parseAngles2File(d, filename);
parseAngles2File(d); // default filename is "data/angles.csv"
```

The subsequent functions are able to write the corresponding data values from a CSV file to all motors at every time step. Between each time step, a delay is imposed so as to facilitate a smooth motion within the motor systems. This delay can defined and can either be system-dependent (e.g. until all motors stop moving before proceeeding) or time-based (e.g. delay for 10 ms). Delay behavior can be defined in the function `timepointDelay` in `src/userinputfn.cpp`.
Data to be written to the control table may be a single byte or a word (two bytes, consisting of a low and high byte). Hence, the functions `readFile2Byte` and `readFile2Word` are given, depending on what kind of attributes the data represent. (E.g. Speed and Position are words while PID gains are bytes.) There are two ways that these functions may be used.
The first method is directly passing in the CSV files as parameters.

```
readFile2Byte(d, filename, delayTime);
readFile2Word(d, filename, delayTime);
```

The parameter delayTime is optional, and is set at 4 ms if no value is given.
The second method is parsing the file into a 2D vector prior to parsing the information.

```
queue<vector<int> > all_instr;
getInstr(all_instr, filename);

readFile2Byte(d, all_instr, delayTime);
readFile2Word(d, all_instr, delayTime);
```

The second method is useful if the readFile functions are used in a loop, but the file remains unchanged (i.e. avoiding opening the file and parsing it multiple times).
Special versions of the readFile have also been defined for parsing Position/Speed values and Position/Speed/PID values. However, the template of these CSV file have to comply with requirements. The use of these functions are largely the same as above.

```
readFile2PosSpd(d, filename, delayTime);
readFile2PosSpdPID(d, filename, delayTime);
```

```
queue<vector<int> > all_instr;
getInstr(all_instr, filename);

readFile2PosSpd(d, all_instr, delayTime);
readFile2PosSpdPID(d, all_instr, delayTime);
```

5.5 Feedback Loop

Another way of controlling the motors is through a feedback loop. In this scheme, the desired position (and speed) for the motors at the next timepoint is calculated based on the previous/current state (and sensor readings, if any). The transfer function utilized for this loop is defined under newPosFn in src/userinputfn.cpp, which can be modified to suit the project demands. Demonstration of the feedback loop is combined with the multi-threading sequence samples.

5.6 Multi-Threading

Multi-threading allows for functions to be run in parallel. Sample sequences involving threading are demonstrated in threadedseq.cpp/.h. These functions may be adopted and modified as a template or used as-is (by including "dtools/threadedseq.h"), depending on the demands of the program. To use the sequence as is, the following may be used

```
#include "dtools/threadedseq.h"
...
threadedSequenceSample();
```

5.6.1 With Multiple Ports

To facilitate communication with multiple devices at once, multiple ports may be used. This is made possible with the changes introduced to the new SDK.
In the sample sequence, the console requests for the desired number of USB-to-dynamixel ports to open, before attempting to initialize the systems (in parallel).

5.6.2 Between Communication and Calculation

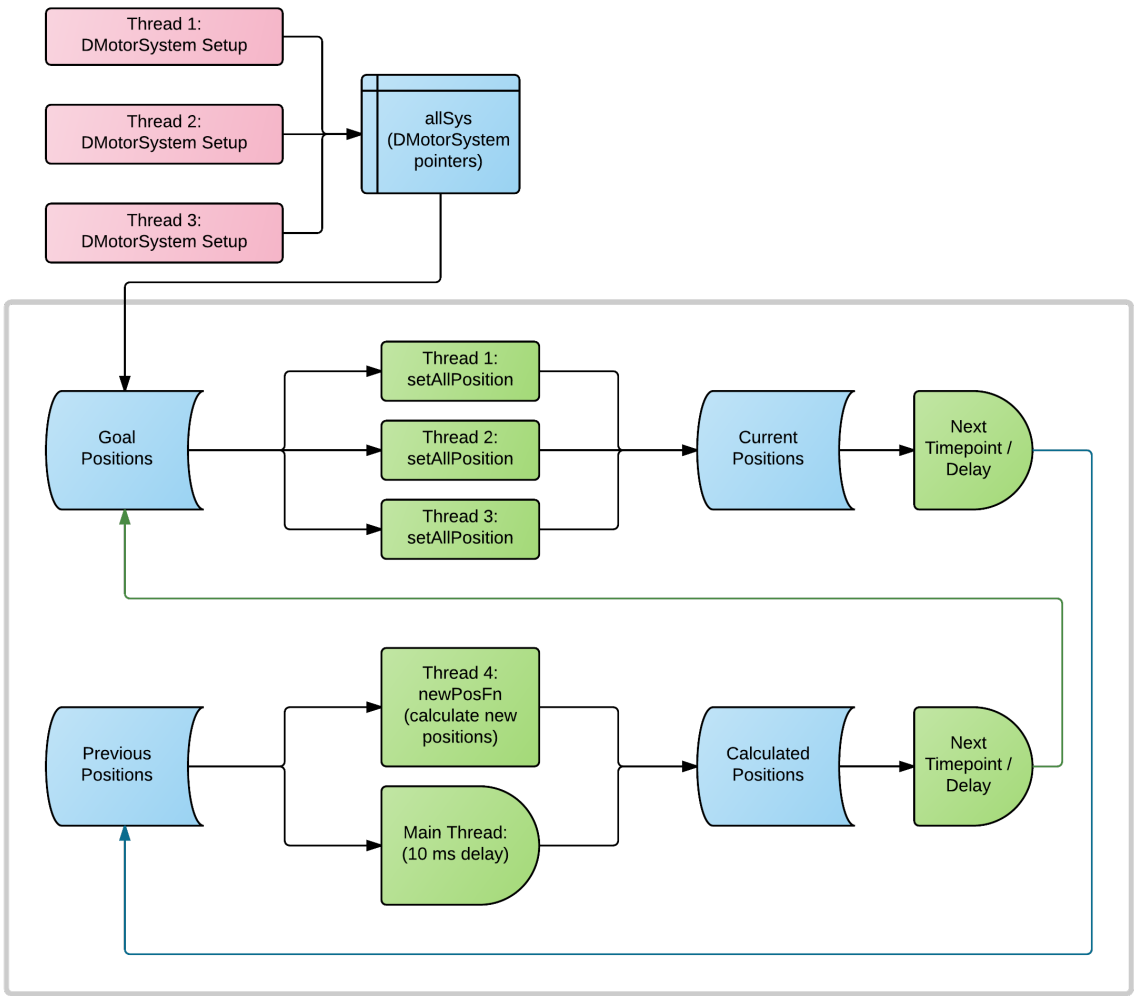

Figure 3: Threaded Feedback Loop

Another place where threading may be introduced is the data communication and calculation. To facilitate parallel computing, computation and communication for each individual port can each be run on a

separate thread (assuming they do not require access to the same variables all at once).
In this specific sequence, a "one-ahead" calculation scheme is used. The desired goal positions are communicated to each port on separate threads, while the calculated goal positions for the next time points are based on the positions of the previous time point. This minimizes the waiting time, and hence the execution time for each loop.

5.6.3 Read File and Feedback Loop

(Tentative Schematic. To be edited.)

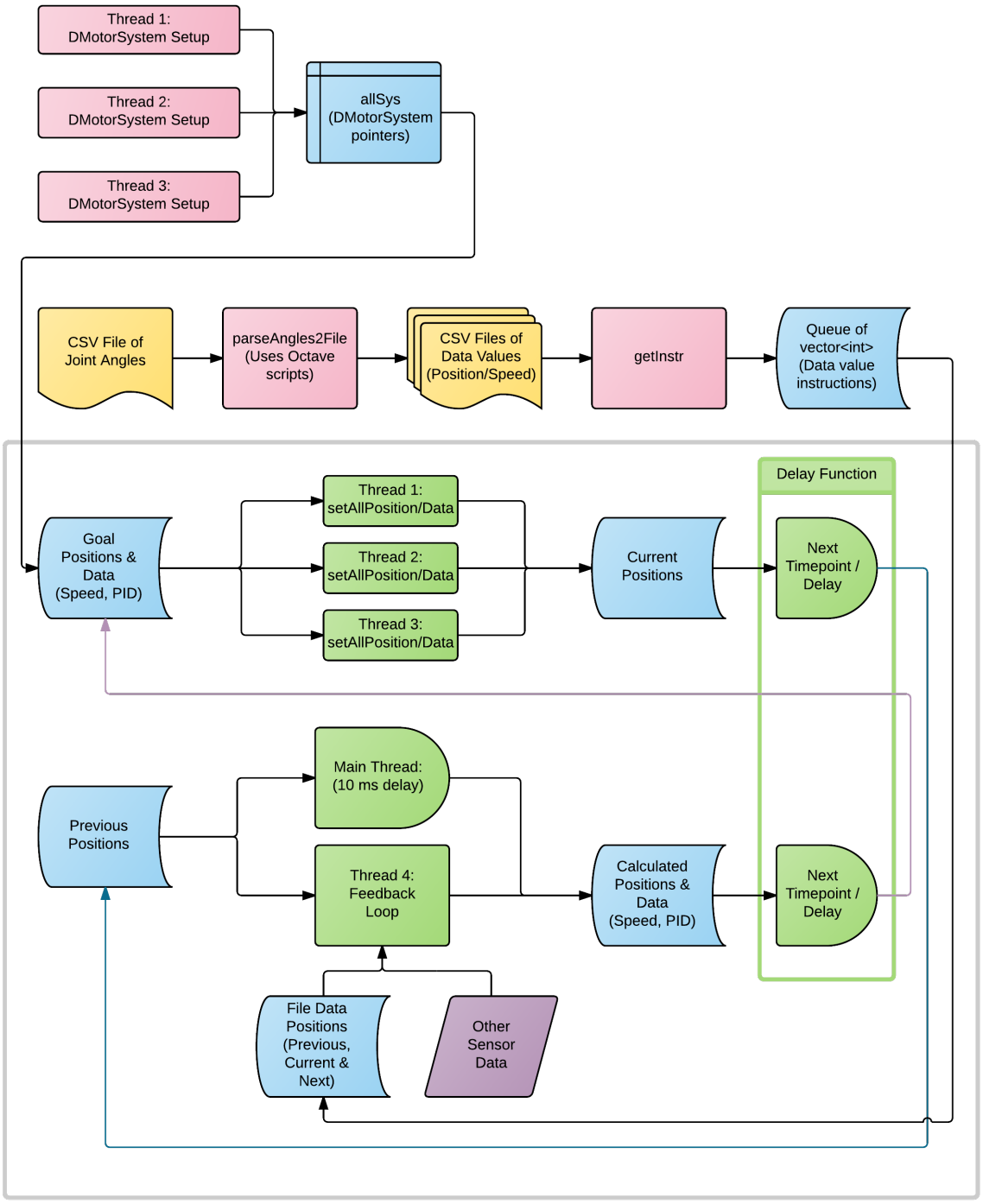

Figure 4: Threaded File Reading and Feedback Loop

5.6.4 Cautionary Note

Unlike sequential programming, parallel computing may result in concurrency issues. One of the most common pitfalls is the concurrent access (and modification) of variables and objects. When multi-threading is in use, potential concurrency issues should be taken into consideration.

6 Graphical User Interface

The library also includes a graphical user interface (GUI) for high-level control features, specifically the file-reading and bilateral master/slave control capabilities. The code uses the Qt framework, and can be compiled in Qt Creator IDE.

(This section will be updated with more specific information about the GUI shortly.)

7 DStopWatch

The class, DStopWatch, is a new class created to characterize execution time. Each DStopWatch instance corresponds to an individual stopwatch. It can be used as follows

```
DStopWatch dsw; // Initialize object
dsw.start(); // Start stopwatch
int timeLap = dsw.lap(); // Lap, record time
...
int timeElapsed = dsw.stop(); // Stop, record time
```

7.1 Characterizing Dynamixel SDK Functions

Using the stopwatch object, the execution time of some basic functions of the new dynamixel SDK were characterized. In summary (and as a rough guide),

- Writing bytes (to a single or all motors) takes about 15-20 μs. (Time taken scales with number of bytes to write, but not with number of motors.)
- Reading an address off a motor takes about 1 ms. (Cumulative time for multiple MX motors is reduced through the use of `BULK_READ`.)

These are based on the execution time of the code, and does not take into account mechanical latencies or communication delays (different system delay times, as per settings on control table).