mPMT Test Stand:

Software Manual

Alex Ezzat, Alex Proskurin, Chuan Du, Noah Tajwar

Version 2: April 20, 2020

**Purpose:**

The mPMT Test Stand is intended to test single and multi-PMTs for the IWCD and Hyper-Kamiokande experiments. This guide includes the following components

at the hand-off of the UBC capstone project.

* Usage guide for all user-facing software components
* Architectural/interface description of all user-facing and internal software components
* Specifications for
  + motor control and temperature measurement
  + serial communication protocol between Arduino and PC
  + MIDAS components

**Table of Contents**

[Usage Guide 4](#_Toc38248610)

[Resources 4](#_Toc38248611)

[Software Architecture Overview 4](#_Toc38248612)

[Software Components 5](#_Toc38248613)

[Motor Control 5](#_Toc38248614)

[Units / Coordinate System 6](#_Toc38248615)

[Kinematics Module 7](#_Toc38248616)

[Timer Module 8](#_Toc38248617)

[Axis Module 9](#_Toc38248618)

[Temperature Measurement 11](#_Toc38248619)

[Serial Communication 13](#_Toc38248620)

[Hardware Layer 14](#_Toc38248621)

[Transport Layer 15](#_Toc38248622)

[Session Layer 15](#_Toc38248623)

[Application Layer 15](#_Toc38248624)

[Boot Sequence 19](#_Toc38248625)

[Link Check 19](#_Toc38248626)

[MessageTerminal 19](#_Toc38248627)

[MIDAS Components 19](#_Toc38248628)

[Units 20](#_Toc38248629)

[Arduino Frontend 21](#_Toc38248630)

[Scan Frontend + Webpage 22](#_Toc38248631)

[Configuration / Calibration 23](#_Toc38248632)

[Configuration 23](#_Toc38248633)

[Calibration 23](#_Toc38248634)

[Default Calibration 24](#_Toc38248635)

[Updating Calibration Values 24](#_Toc38248636)

[Integration with MIDAS 24](#_Toc38248637)

[Adding New Calibration Parameters 24](#_Toc38248638)

# Usage Guide

Detailed information on building and using the test stand software can be found in the [repository README](https://github.com/nuPRISM/mpmt-test-stand).

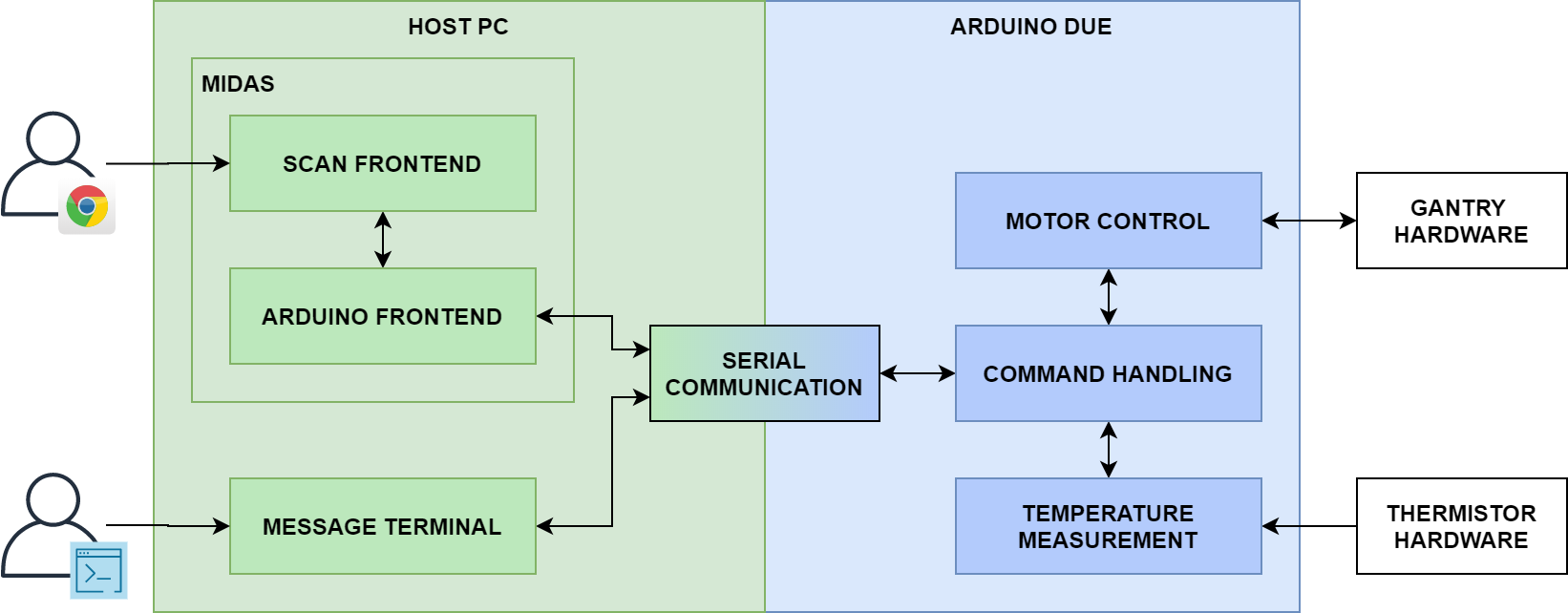
# Resources

|  |  |
| --- | --- |
| mpmt-test-stand Software Repository | <https://github.com/nuPRISM/mpmt-test-stand> |
| MIDAS Wiki | <https://midas.triumf.ca/MidasWiki/index.php/Main_Page> |
| Arduino Due Information Page | <https://store.arduino.cc/usa/due> |
| SAM3X8E Information Page | <https://www.microchip.com/wwwproducts/en/ATSAM3X8E> |
| SAM3X8E Datasheet | <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf> |
| SAM3X-Arduino Pin Mapping | <https://www.arduino.cc/en/Hacking/PinMappingSAM3X> |

*Arduino Due and SAM3X8E datasheets can also be found on the* [*hyperK website*](https://www.hyperk.ca/mpmt/pmt-test-stand/capstone-mpmt-hand-off-documentation/datasheet)*.*

# Software Architecture Overview

The software for the mPMT Test Stand consists of 7 components running across two platforms: a Host PC and an Arduino Due. All software, apart from the user interface, is written in C++ (the user interface is written in HTML/CSS/JS). These components are organized as illustrated in Figure 1.



**Figure 1.** High-level software architecture

The end goal of the software is to enable a user interacting with the Host PC to control and acquire data from the test stand hardware.

On the Host PC, a data acquisition framework called [MIDAS](https://midas.triumf.ca/MidasWiki/index.php/Main_Page) (Maximum Integrated Data Acquisition System) is running which provides a user interface and backend mechanisms for logging data, storing configuration information, monitoring “runs” etc. The Scan Frontend and Arduino Frontend are two independently running programs that connect to MIDAS and make use of its various features. The **Scan Frontend** includes a web-browser-accessible user interface for running and monitoring scans of an mPMT or 5x5 PMT array. The **Arduino Frontend** receives instructions from the Scan Frontend and is responsible for sending commands to, and retrieving data from the Arduino Due.

The **Message Terminal** is a command-line-based, alternative method to interacting with the Arduino Due. It provides access to the lower-level messaging API that can be useful for testing and debugging.

The **Serial Communication** component is a shared, cross-platform collection of classes that implement a serial communication protocol to pass messages between the Host PC and Arduino Due.

The **Command Handling** component is the Arduino firmware component responsible for receiving and interpreting serial messages as well as sending back responses and data.

The **Motor Control** component handles all interaction with the Gantry hardware (which includes stepper motor drivers, encoders, limit switches etc.)

The **Temperature Measurement** component handles all interaction with the thermistors placed within the test stand.

Details on the architecture and implementation of each of these components will be covered in the Software Components section below.

# Software Components

## Motor Control

The motor control firmware consists of all of the code in the “firmware/lib/Gantry” directory which includes the following:

* **Gantry.h**
  + Public interface for code outside the firmware
* **Kinematics Module:** Kinematics.h / Kinematics.cpp
  + Helper functions for kinematics calculations
* **Timer Module:** Timer.h / Timer.cpp
  + Helper functions for configuring hardware timers
* **Axis Module:** Axis.h / Axis.cpp
  + Core motor control code

### Units / Coordinate System

The motor control firmware deals with the following two sets of units:

1. steps, steps/s, steps/s2
2. counts, counts/s, counts/s2

A **step** is a unit of distance equal to the distance a gantry axis will travel after a single pulse is sent to the stepper motor driver.

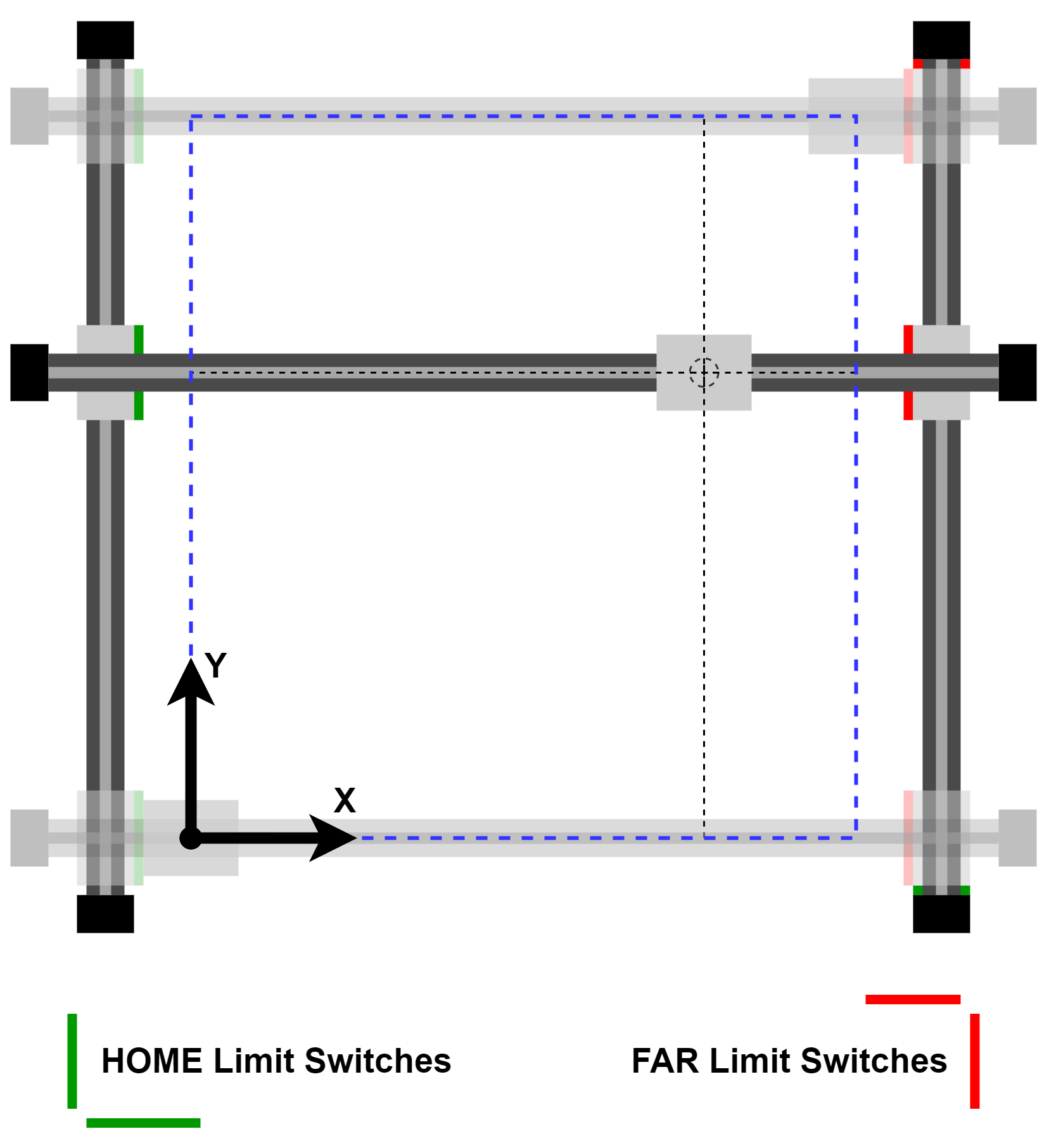
A **count** is a unit of distance equal to the distance a gantry axis travels in between consecutive encoder counts.

The conversion ratio between these units is determined by the properties of the stepper motors and the encoders:

where is the number of encoder counts per revolution of the motor and is the number of motor steps per revolution of the motor. The steps per revolution varies with the microstep setting according to .

In general, throughout the motor control code, velocities and accelerations are dealt with in units of steps/s and steps/s2 while distances are dealt with in units of counts.

Position measurements are made with reference to the coordinate system shown in Figure 2.

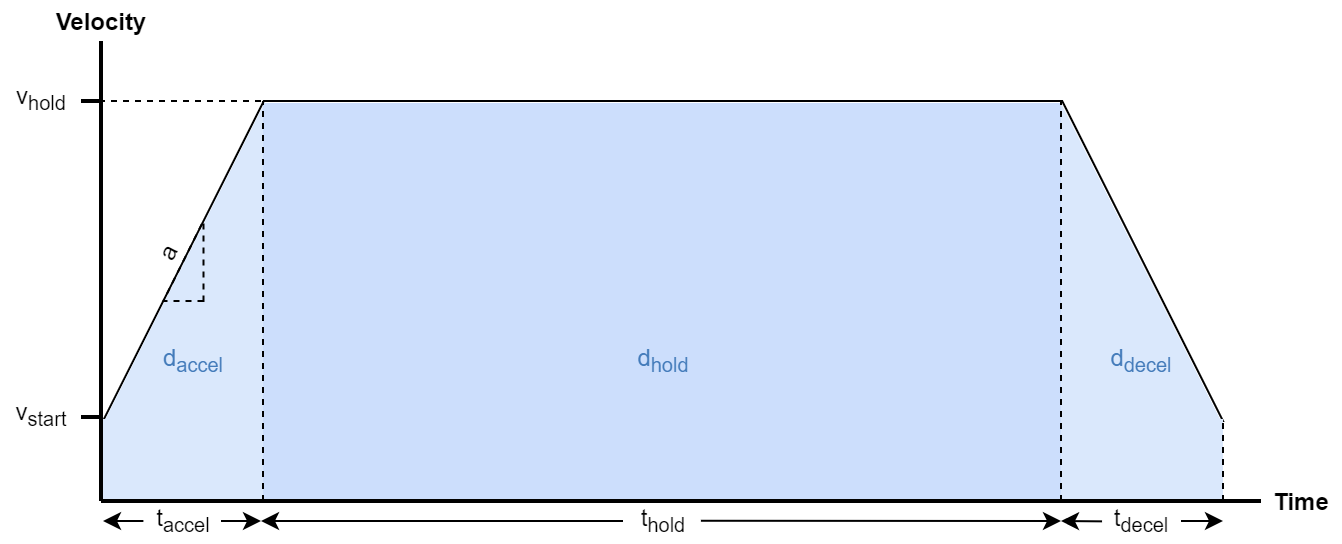


**Figure 2.** Definition of coordinate system

The origin of the coordinate system (referred to as the home position) is the point at which both HOME limit switches are just released (i.e. any movement in the negative direction would start to press them). The farthest reachable point of the coordinate system is that where both FAR limit switches are pressed. These bounds are indicated by the blue dashed rectangle in the diagram.

### Kinematics Module

In order to reduce mechanical vibration in the gantry and facilitate driving at high speeds, the stepper motors must be gradually accelerated and decelerated at the beginning and end of a motion as illustrated by the velocity profile in Figure 3.



**Figure 3.** Velocity profile

The Kinematics code takes the following set of quantities as inputs (which fully define a velocity profile):

* starting velocity
* holding velocity
* acceleration
* total distance =

And calculates the individual quantities , , using the standard kinematics equation :

These are the quantities required by the Axis module to execute a motion.

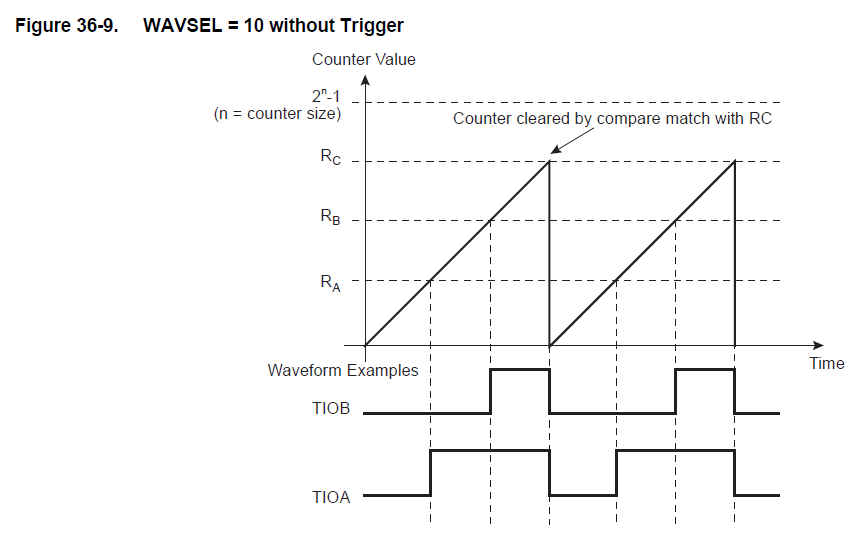
The Kinematics module does not operate in any particular unit system. The output values will have units corresponding to those of the input values (e.g. if the input values are in steps, steps/s, and steps/s2, then the output values will be in the same units).

### Timer Module

The stepper motor drivers require PWM signals as inputs where each pulse on the PWM signal causes the stepper motor to take one step. To achieve high speeds and precise timing, hardware timers on the Arduino Due are used to generate these PWM signals as well as the required acceleration / deceleration ramps. The Timer Module provides an interface to configure, start and stop the hardware timers.

There are three timer counter (TC) peripherals on the Arduino Due, each with three independently configurable channels for a total of nine timer channels. Each timer channel has two outputs, TIOA and TIOB (although not all of them are exposed on the Due pins). Each timer channel can also trigger an interrupt which runs a different interrupt service routine (ISR).

The Timer Module can configure timer channels to output a PWM waveform on the TIOA output as illustrated in Figure 4, with a duty cycle determined by RA and a frequency determined by RC. An interrupt will also be generated when the counter reaches RC. This configuration is used to drive the stepper motor drivers. The frequency of the PWM signal (i.e. number of pulses per second) corresponds to the velocity of the motor in steps/s since each pulse causes a single motor step.



**Figure 4.** Arduino Due timer controller output signals (source: SAM3X8E datasheet)

The Timer Module can also configure a timer to only produce interrupts at a particular frequency. This configuration is used to produce a discrete approximation to the acceleration ramp by increasing or decreasing the set velocity at regular intervals.

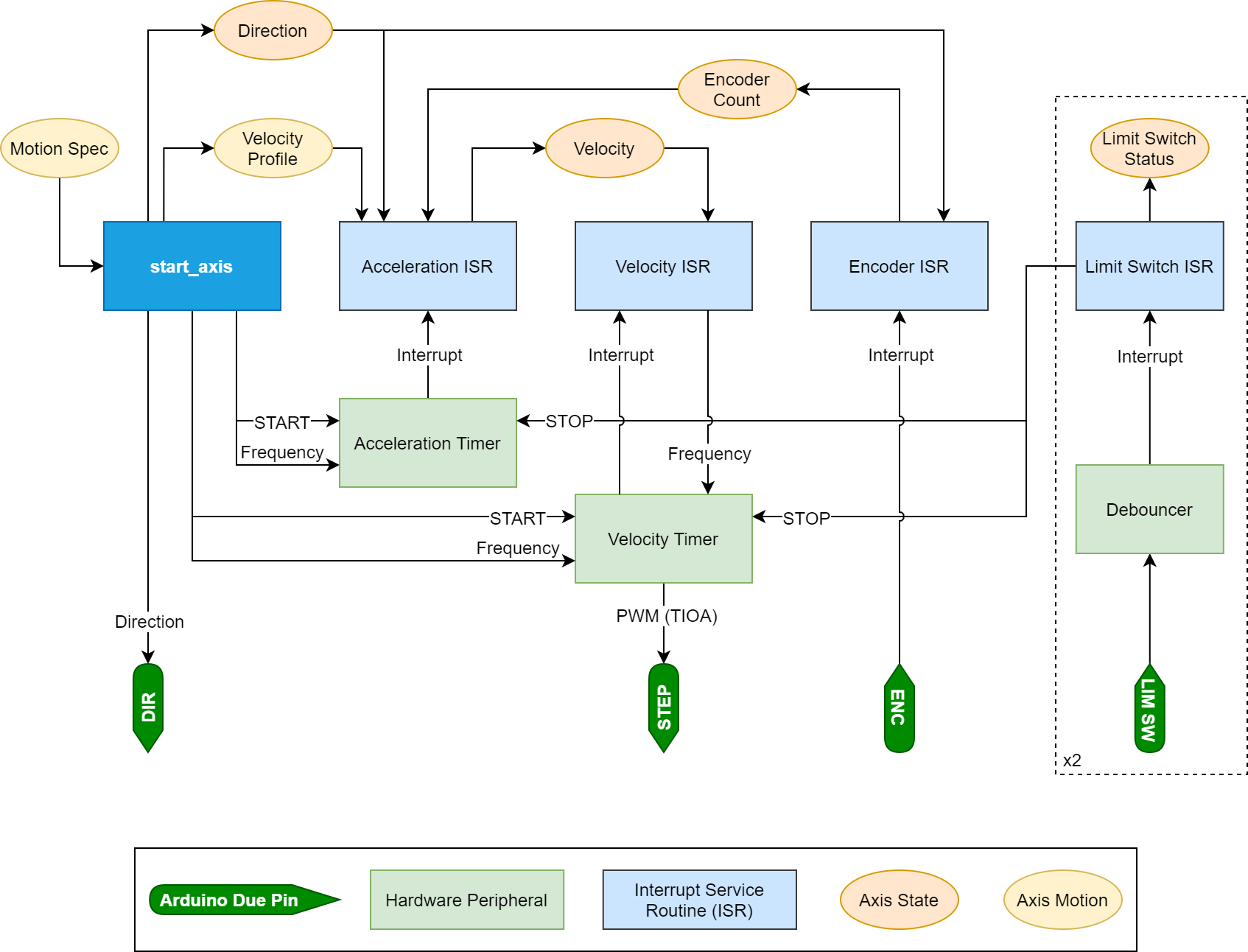
Detailed information on the hardware timers is available in the SAM3X8E datasheet (linked in the Resources section).

### Axis Module

The Axis Module brings together all the elements of motor control.

The Gantry is broken down into two axes: X and Y. A physical axis represents the combination of a stepper motor, stepper motor driver, drive train, encoder and two limit switches that enable controlled motion in a single dimension. Within the firmware, each axis is represented by an instance of the Axis struct defined in (Axis.cpp). The Axis Module exposes methods to configure, start, stop, reset and monitor each axis.

The operation of each axis is illustrated by Figure 5.



**Figure 5.** Concept of operations for a single gantry axis

1. Starting a Motion
   1. A Motion Spec defining the parameters of the desired motion is passed to the start\_axis function
   2. The start\_axis function performs the following actions:
      1. Processes the Motion Spec and generates a Velocity Profile
      2. Stores the Direction into the Axis State
      3. Drives the DIR pin according to the direction (this pin is connected to the stepper motor driver)
      4. Sets the frequency for the Acceleration Timer to the acceleration rate
      5. Sets the frequency for the Velocity Timer to the initial velocity
      6. Starts the Acceleration and Velocity timers
2. Acceleration
   1. The Acceleration Timer triggers an interrupt at the acceleration frequency
   2. The Acceleration ISR uses the Velocity Profile, Direction and Encoder Count to update the set Velocity
3. Velocity
   1. The Velocity Timer triggers an interrupt at the velocity frequency
   2. The Velocity ISR updates the Velocity Timer frequency with the set Velocity
   3. The Velocity Timer drives a PWM signal on its TIOA output which is connected to the STEP pin (this pin is connected to the stepper motor driver)
4. Encoder
   1. The ENC pin (connected to the A output of the encoders) triggers interrupts on every encoder count
   2. The Encoder ISR updates the Encoder Count in the Axis State
5. Limit Switches
   1. Each LIM SW pin is debounced in hardware
   2. The debounced LIM SW signal triggers interrupts when the switch state changes
   3. The Limit Switch ISR stops the Velocity and Acceleration Timers (halting axis motion) and updates the Limit Switch Statuses in the Axis State

There are several conditions under which the Axis Module will refuse to start a motion (and will return the appropriate error code instead - see AxisResult in Gantry.h):

* The requested axis is already in motion (you must stop the axis first, then issue a new motion)
* Trying to move beyond a limit switch, which includes:
  + Trying to move in the positive direction while the far limit switch is pressed
  + Trying to move in the negative direction while the home limit switch is pressed
* Specifying mathematically invalid kinematic parameters:
  + E.g. zero acceleration when the hold velocity is higher than the start velocity
  + E.g. hold velocity less than the start velocity

## Temperature Measurement

The temperature library is located in “firmware/lib/TemperatureDAQ” and is designed to work with thermistors with negative temperature coefficients (NTC). A thermistor is a device which varies its resistance based on changes in temperature. NTC refers to the fact that the thermistor’s resistance decreases as the temperature increases. Using the Arduino Due onboard 12-bit resolution analog-to-digital converter (ADC), an analog value between 0 - 4096 is mapped to a voltage between 0 - 3.3 V. This is subsequently converted to a resistance using the voltage divider series resistance value.

The thermistor readings can be done in two ways:

* double readTemperature();
  + The current firmware takes a single analog reading at the interval specified by MIDAS.
* double readAveragedTemperature(int numSamples);
  + If there is noticeable noise and fluctuation in the reading, it may be minimized by taking multiple measurements consecutively and outputting the averaged temperature across these samples.
  + Navigate to “firmware\src\mPMTTestStand.cpp”.
  + In void mPMTTestStand::handle\_get\_temp() replace readTemperature()with readAveragedTemperature(5) where numSample is 5 as an example.
  + While function did not appear to improve the precision or accuracy of reading during our testing, it may become useful under the gantry operation conditions.

This thermistor library depends on four calibration constants to convert analog voltage reading into its corresponding temperature in degrees Celsius (see the ThermistorCalibration struct in TemperatureDAQ.h)

* resistor
  + The thermistor reading is obtained across the thermistor and ground. On the high-side, there is a 10 kΩ series resistor. Since this is a precision resistor, the nominal value can be used for all thermistor circuits without additional modification of this value.
* c1,c2,c3
  + These are the Steinhart-hart equation constants specific to the USP10976 10kΩ thermistors.

(Steinhart-Hart Equation)

* + A = c1, B = c2, C = c3 and T is in degrees Kelvin.
  + An online calculator from “STS” by Stanford Research Systems Inc.[[1]](#footnote-1) is used to extract Steinhart-Hart model coefficients through entering three resistance and temperature mappings provided by the lookup table (see Datasheet for lookup table). If another 10kΩ thermistor from another manufacturer is used, the coefficients and calibration can be recalculated using this method.

The default calibration constants (see the Configuration/Calibration section) are for USP10976 10kΩ thermistors. More information on this component such as its datasheet and calibration lookup table can be found in the [Datasheet folder](https://www.hyperk.ca/mpmt/pmt-test-stand/capstone-mpmt-hand-off-documentation/datasheet/thermistor) on the hyperK website.

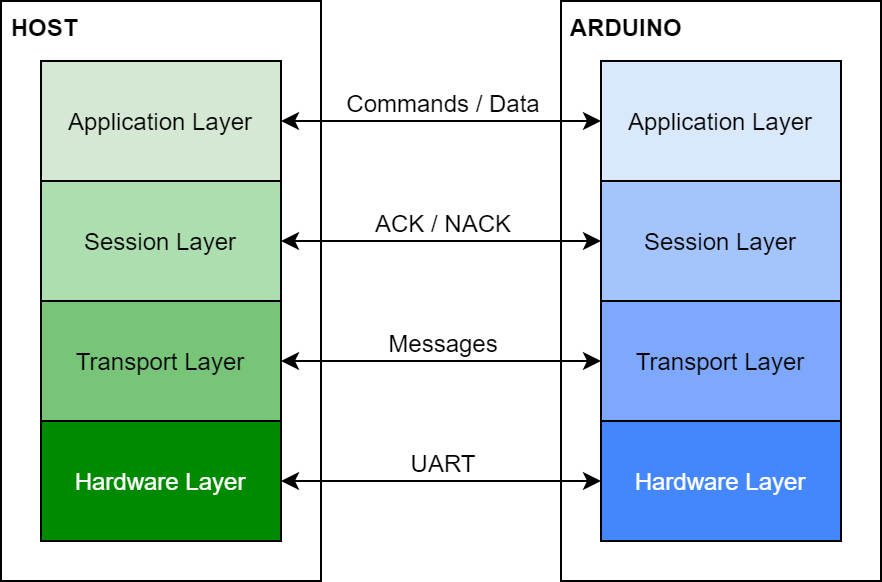
**Table 1.** The resistance and temperature values used in order to calculate Steinhart-Hart model coefficients for thermistor calibration.

|  |  |
| --- | --- |
| **Resistance (Ω)** | **Temperature (°C)** |
| 25392 | 5 |
| 10000 | 25 |
| 4368 | 45 |
| **Steinhart-Hart model coefficients** | |
| A (c1) | 1.127587560e-3 |
| B (c2) | 2.343528476e-4 |
| C (c3) | 0.8698059497e-7 |

## Serial Communication

The Arduino Due includes several UART peripherals one of which is connected to the programming micro-USB port through an on-board USB-to-Serial converter. This serial connection is used for communication between the Arduino Due and the Host PC.

A custom binary protocol was designed and implemented to facilitate reliably passing commands and data between the two devices. The protocol which is loosely based on the [OSI model](https://en.wikipedia.org/wiki/OSI_model), is structured with four layers as illustrated in Figure 6.



**Figure 6.** Serial communication protocol layer structure

* **Hardware Layer:** This layer is responsible for communication with the hardware serial device to send and receive individual bytes.
* **Transport Layer:** This layer packages data into “messages” to transmit and receives full messages.
* **Session Layer:** This layer is responsible for sending and checking for ACK or NACK messages in response to receiving and sending messages.
* **Application Layer:** This layer is responsible for constructing the actual messages used by the application and handling multi-message exchanges.

The software / firmware implementing the transport, session and part of the application layer (a shared base class providing helper methods) is located in the shared/TestStandComm directory. The hardware layer and main application layers are implemented separately for each device. For the Host PC, these are located in the shared\_linux/LinuxSerialDevice and shared\_linux/TestStandComm directories. For the Arduino these are located in the firmware/lib/Comm directory.

Each layer is implemented by the classes listed in Table 2:

**Table 2.** Classes belonging to each serial communication layer

|  |  |  |
| --- | --- | --- |
| **Layer** | **Host PC Implementation** | **Arduino Implementation** |
| **Application** | TestStandComm.cxx | |
| TestStandCommHost.cxx | TestStandCommController.cxx |
| **Session** | SerialSession.cxx | |
| **Transport** | SerialTransport.cxx | |
| **Hardware** | LinuxSerialDevice.cxx | ArduinoSerialDevice.cxx |

### Hardware Layer

**Arduino**

On the Arduino side the hardware layer (ArduinoSerialTransport) just acts as a wrapper around the Arduino framework’s HardwareSerial class. The UART port being used for serial communication can be configured as discussed in the Configuration section below. The HardwareSerial instance that is connected to the USB-to-serial converted is called Serial.

**Host PC**

On the Host PC side, the hardware layer (LinuxSerialTransport) configures, opens and communicates through a serial port specified by the path to a device file (e.g. /dev/ttyACM0). The serial port will be configured in a “raw” mode (where no control characters are interpreted) as the protocol is binary, so all bytes must pass through to the application. After the Host PC programs exit, the serial port will be left in this configuration.

The baud rate for the serial communication can be configured at compile time. See the Configuration section below for details.

### Transport Layer

The transport layer introduces the concept of a “message”. The byte structure of a message (of length n bytes) is described in Table 3.

**Table 3.** Transport layer message structure byte layout

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Byte** | 0 | 1 | 2 | 3...n-4 | n-3..n-2 | n-1 |
| **Name** | START | ID | LEN | DATA | CRC16 | END |
| **Description** | 0x7B | Message ID | Length of DATA | Message data | Not impl. | 0x7D |

The upper layers of the messaging protocol are only concerned with the ID, LEN and DATA portions of the message (represented in the software by the Message struct in Messages.h). The transport layer takes care of filling in the START, CRC16 and END sections when sending a message, as well as parsing them out when receiving a message.

NOTE: The CRC16 field is intended to be a 16-bit CRC calculated over the ID, LEN and DATA fields to improve error detection. This feature has not yet been implemented, so this field will always be sent with 0x0000 and will not be checked when receiving a message.

### Session Layer

The session layer handles sending and receiving message acknowledgements. This includes sending an ACK message (ID = 0x01) in response to every received message and checking for an ACK response after sending a message. The session will wait a predefined amount of time for an ACK response before returning an error. This can be configured by changing ACK\_TIMEOUT\_MS in SeriaSession.cxx.

### Application Layer

A description of all implemented messages is available in Table 4a, Table 4b and Table 4c. The “Data Layout” column specifies the byte addresses of the fields within the data. The message IDs are allocated as follows:

* **0x00 to 0x0F:** Reserved for use by the session layer
* **0x10 to 0x3F:** Two-way messages (can be sent in either direction)
* **0x40 to 0x7F:** Host PC → Arduino messages
* **0x80 to 0xFF:** Arduino → Host PC messages

**NOTE:** All numerical values are transmitted in network (i.e. big endian) byte order.

**Table 4a**. Two-way messages

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Description** | **Data Layout** |
| 0x10 | *PING* | Check if the recipient is still alive (indicated by an ACK at the session layer)  Expected response: None | None |
| 0x11 | *ECHO* | Request the receiver to send back the exact data it receives  Expected response: *ECHOED* | Arbitrary |
| 0x12 | *ECHOED* | Response to an *ECHO* message | Same as received in *ECHO* message |

**Table 4b**. PC → Arduino messages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Description** | **Data Layout** | |
| 0x40 | *GET\_STATUS* | Request the current status of the Arduino  Expected response: *STATUS* | None | |
| 0x41 | *HOME* | Start the gantry homing routine  Expected response: None | None | |
| 0x42 | *MOVE* | Start the gantry moving to a new position  **vel\_hold**: Holding velocity [steps/s]  **dist\_counts**: Total distance to travel relative to the current position [counts]  **axis**: One of AxisId (see Gantry.h)  **dir**: One of AxisDirection (see Gantry.h)  Expected response: *AXIS\_RESULT* | [0..3]  [4..7]  [8]  [9] | vel\_hold  (uint32\_t)  dist\_counts  (uint32\_t)  axis  (uint8\_t)  dir  (uint8\_t) |
| 0x43 | *STOP* | Immediately stop the gantry if it is currently moving | None | |
| 0x44 | *GET\_POSITION* | Request the current position (X and Y coordinates) of the gantry  Expected response: *POSITION* | None | |
| 0x45 | *GET\_AXIS\_STATE* | Request the current state information of one of the axes  **axis**: One of AxisId (see Gantry.h)  Expected response: *AXIS\_STATE* | [0] | axis  (uint8\_t) |
| 0x46 | *GET\_TEMP* | Request new temperature measurements from the themistors  Expected response: *TEMP* | None | |
| 0x47 | *CALIBRATE* | Update a calibration value  **key**:One of CalibrationKey (see Calibration.h)  **value:** Depends on the CalibrationKey | [0]  [1..n] | key  (uint8\_t)  value |

**Table 4c.** Arduino → PC messages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Description** | **Data Layout** | |
| 0x80 | *LOG* | Send a string from the Arduino for the Host PC application to log  **log\_level**: One of LogLevel (see TestStandMessages.h)  **msg**: String to log  Expected response: None | [0]  [1..n] | log\_level  (uint8\_t)  msg  (char[]) |
| 0x81 | *STATUS* | Response to *GET\_STATUS* containing the current status of the Arduino  **status:** One of Status (see shared\_defs.h)  Expected response: None | [0] | status  (uint8\_t) |
| 0x82 | *POSITION* | Response to *GET\_POSITION* containing the current coordinates of the gantry  **x\_counts**: Current x coordinate [counts]  **y\_counts:** Current y coordinate [counts]  NOTE: Both values can be negative if the home position was miscalibrated or any encoder counts were missed  Expected response: None | [0..3]  [4..7] | x\_counts  (int32\_t)  y\_counts  (int32\_t) |
| 0x83 | *AXIS\_STATE* | To be implemented | To be implemented | |
| 0x84 | *TEMP* | Response to *GET\_TEMP* containing the latest temperature measurements from all of the thermistors.  All temperatures are returned as integers in units of , providing 6 decimal places of precision. The scale factor can be changed in TempMeasure.h  **ambient:** Ambient temperature  **motor\_x**: Temperature near the motor driving the X axis  **motor\_y**: Temperature near the motor driving the Y axis  **mpmt**: Temperature near the mPMT  **optical**: Temperature inside the optical box | [0..3]  [4..7]  [8..11]  [12..15]  [16..19] | ambient  (int32\_t)  motor\_x  (int32\_t)  motor\_y  (int32\_t)  mpmt  (int32\_t)  optical  (int32\_t) |
| 0x85 | *AXIS\_RESULT* | Response to *MOVE* containing success/error information for the attempted gantry movement  **result**: One of AxisResult (see Gantry.h) | [0] | result  (uint8\_t) |

### Boot Sequence

When the Arduino firmware boots, it waits in a loop of sending *PING* messages to the Host PC after completing its own configuration. Once the Host PC ACK’s the message, it proceeds to its main loop. This way both the Arduino and the Host PC start their main execution in a state knowing the serial link is working.

### Link Check

There is a method included in the TestStandComm class called link check that sends an *ECHO* command containing every byte value between 0x00 and 0xFF as data and confirms the *ECHOED* response is received and contains the correct data. The purpose of this check is to confirm that all bytes can be successfully transmitted over the serial connection, in particular that no bytes are being filtered out as control characters. The Arduino is not susceptible to this; however, if the Linux serial port device is mis-configured, it may process and remove control characters from the data stream so they are never actually transmitted or received. The LinuxSerialDevice class should take care of configuring the serial port, so this check is included as a redundancy and to make it easier to diagnose serial issues.

## MessageTerminal

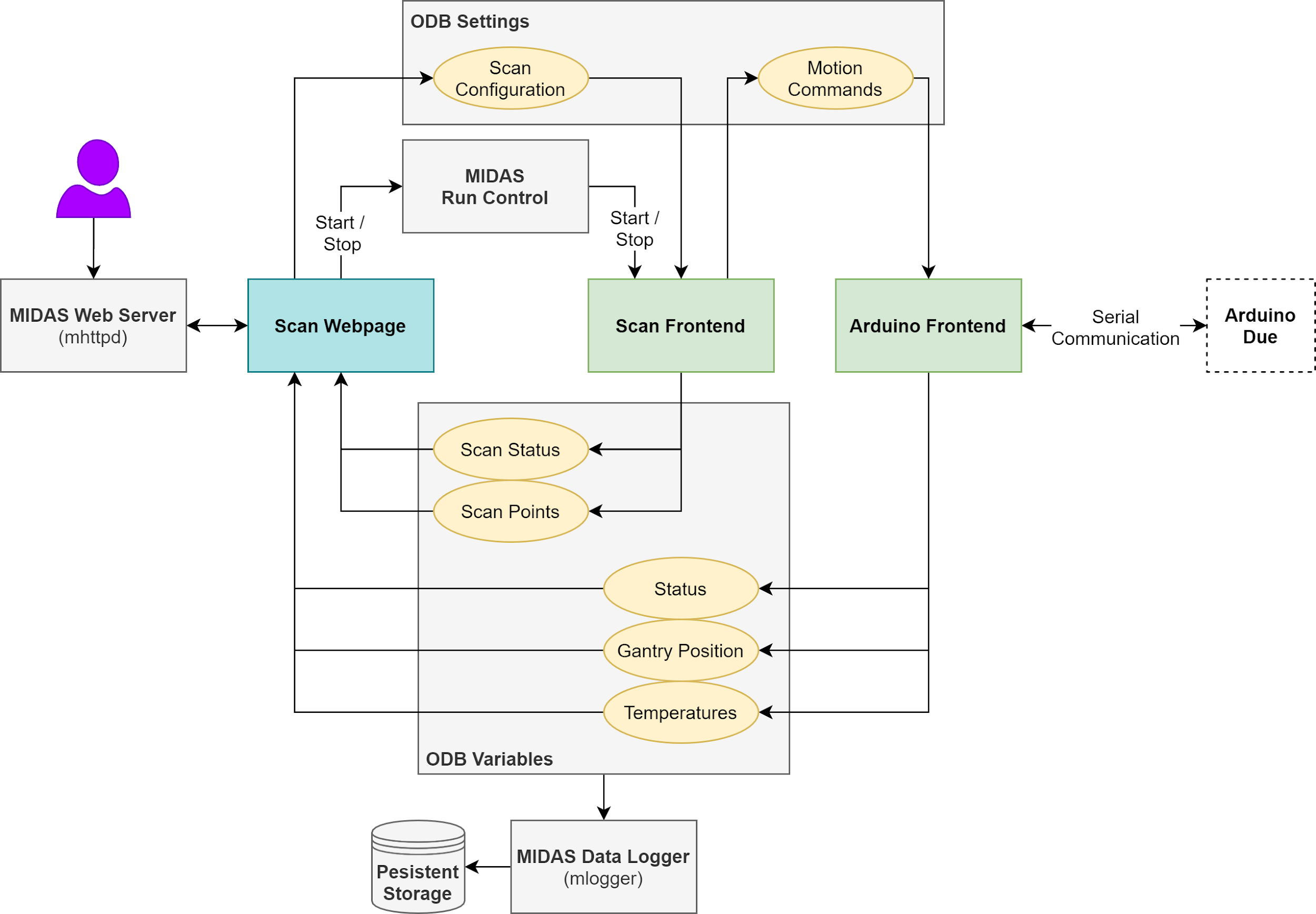
The MessageTerminal application provides a basic command-line interface to manually send messages to the Arduino. Internally it parses the command-line input and calls through to the appropriate methods in TestStandCommHost to send the messages and process the responses. Usage information can be found in the [repository README](https://github.com/nuPRISM/mpmt-test-stand).

## MIDAS Components

There are two custom MIDAS software components that allow a user to operate the test stand:

1. Arduino Frontend
2. Scan Frontend + Webpage

The architecture of how these components interact is shown in Figure 7.



**Figure 7.** MIDAS component architecture and communication

The components communicate with each other by writing and reading variables in the Online Database (ODB) to which they all have access. The intended method of user interaction is through the Scan Webpage; however it is also possible for the user to edit ODB variables directly (through the command line or through the MIDAS built-in webpages).

### Units

The MIDAS components operate in units of mm, mm/s, mm/s2, etc. Since the Arduino operates in units of steps and counts, a conversion must be made before sending values to the Arduino. The conversion ratio depends on the mechanical properties of the gantry as follows:

where linear distance per revolution

This unit conversion is handled by the Arduino Frontend (in ArduinoHelper.cxx).

### Arduino Frontend

The Arduino Frontend has two purposes:

1. Provide an interface to other MIDAS components to send motor control commands to the Arduino
2. Request and log sensor and status information from the Arduino

**Initialization**

When the Arduino Frontend starts, it performs the following initialization tasks (in the frontend\_init function):

1. Establish a serial connection with the Arduino (the serial port is provided as a command-line parameter)
2. Perform a link check as described in the Serial Communication section
3. Initialize ODB variables for Velocity and Destination to default values if the variables do not currently exist
4. Attach hot-links to the MoveRequest and StartHome ODB variables

If any of these steps fail an error code will be returned from frontend\_init.

**Motor Control**

The Velocity and Destination ODB variables can be used to set the parameters of a desired movement. Velocity controls the holding velocity for the gantry in units of mm/s. Destination sets an absolute destination (relative to the coordinate system origin) in units of mm.

The movement is started by setting the MoveRequest ODB variable to “y” which triggers a hot-link. In the hot-link handler, the Arduino Frontend will send a *MOVE* message to the Arduino. If the Arduino returns an error, an error message will be logged to MIDAS (through the [messaging system](https://midas.triumf.ca/MidasWiki/index.php/Midas_Core#Message_System)).

MoveResponse[0] is set to “y” as soon as the hot-link handler starts running, indicating the Arduino Frontend is processing the move request. MoveResponse[1] is to “y” if the move started successfully, otherwise it is set to “n”.

**Sensor / Status Information**

The Arduino Frontend is configured as a periodic equipment so it has a readout function (read\_arduino\_state) that gets called at regular intervals. This is where sensor and status information is retrieved from the Arduino and stored into the following MIDAS variable banks:

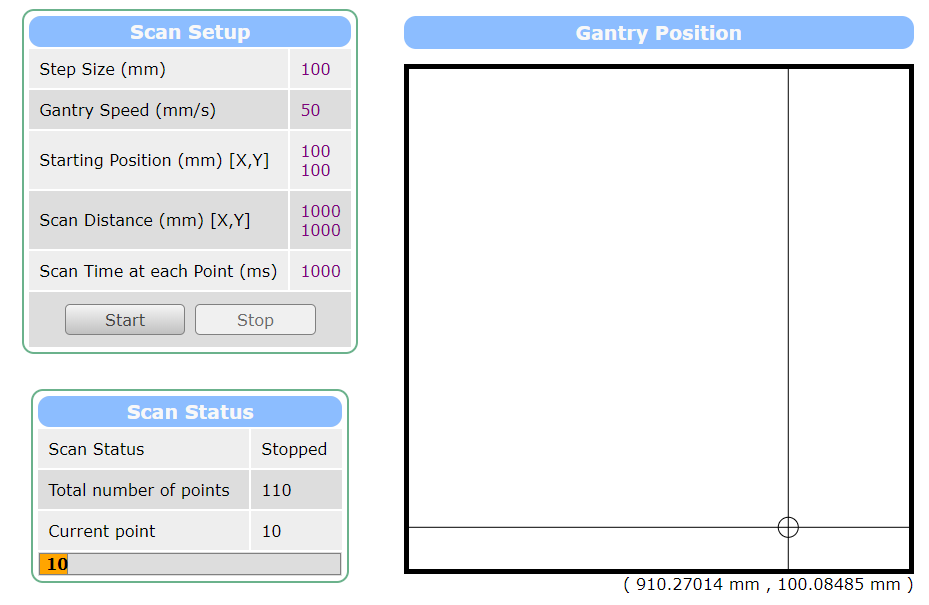
* **STAT**: Stores the status of the Arduino (the response to a *GET\_STATUS* message - see the Status enum in shared\_defs.h)
  + STAT[0] = status
* **GANT**: Stores the current position (X and Y coordinates) of the gantry
  + GANT[0] = X coordinate (mm)
  + GANT[1] = Y coordinate (mm)
* **TEMP**: Stores the latest temperature readings
  + TEMP[0] = Ambient temperature (°C)
  + TEMP[1] = Temperature near the motor that drives the X axis (°C)
  + TEMP[2] = Temperature near the motor that drives the Y axis (°C)
  + TEMP[3] = Temperature near the mPMT (°C)
  + TEMP[4] = Temperature in the optical box (°C)

The update frequency of this data can be configured by changing the “Period” ODB variable under Equipment / ARDUINO / Common.

If the names of any of these ODB variables or paths need to be referenced or used in other components, they can include the feArduino.h header file which includes string #defines for all of the ODB variables and paths.

### Scan Frontend + Webpage

The Scan Webpage provides a graphical interface for the user (Figure 8).



**Figure 8.** Graphical user interface provided by the Scan Webpage

The values in the Scan Setup table are mapped to ODB variables which the Scan Frontend uses to generate a grid of points for the gantry to visit. The Start button is linked to the run control start page where a user can start a run. Once a run has started, the Scan Frontend will begin sending motor commands to the Arduino Frontend through the ODB variables. The JavaScript code in the Scan Webpage runs periodically to update the fields and the gantry position graphic.

Data acquisition is not yet implemented, so for now the Scan Frontend will sit idle for the duration configured in the Scan Setup. There are two functions: start\_measurement() and measurement\_complete() in feScan.cxx which can be replaced with a real mechanism for executing the data acquisition.

# Configuration / Calibration

## Configuration

**Configuration** refers to properties of the software that are set at compile-time and do not change dynamically at run-time. There are shared configuration constants located in the **shared\_defs.h** file, including:

* Serial baud rates
* Communication timeouts
* Gantry motor/encoder specifications

The Arduino firmware additionally has configuration parameters set in **firmware/src/conf.h**. This file contains an instance of the mPMTTestStandConfig struct. This struct contains fields for all configurable aspects of the Arduino firmware including:

* Serial port specifications
* GPIO specifications
* Gantry motor/encoder specifications (drawn from shared\_defs.h)

If any of these fields need to be modified, the firmware must be re-compiled and re-flashed to the Arduino.

## Calibration

**Calibration** refers to properties of the software that might be modified at runtime. The **firmware/include/Calibration.h** file defines the set of available properties for calibrating the Arduino firmware (in the Calibration struct). Each property has a corresponding entry in the CalibrationKey enum. The mPMTTestStand class maintains an instance of the Calibration struct, storing the current calibration parameters.

### Default Calibration

Since the Arduino firmware does not support persistent storage, at boot time, the calibration parameters are initialized with the default values in **shared/DefaultCalibration.h**.

### Updating Calibration Values

Calibration values are updating by sending *CALIBRATE* messages to the Arduino (see the Serial Communication section).

### Integration with MIDAS

The calibration mechanism is integrated with the Arduino MIDAS frontend (feArduino). ODB variables under the /Equipment/ARDUINO/Settings/Calibration tree are created for each calibration parameter. These variables are initialized with the values from DefaultCalibration.h if they do not already exist. When the frontend starts, the current values of these parameters are sent to the Arduino in a series of *CALIBRATE* messages, so the Arduino is immediately updated with the latest values. See the repository README for instructions to manually update calibration parameters.

### Adding New Calibration Parameters

1. Add a new field to the Calibration struct (Calibration.h)
2. Add a new entry to the CalibrationKey enum (Calibration.h)
3. Add a default value for the field to DefaultCalibration.h
4. Add a DEBUG\_PRINT\_VAL statement to mPMTTestStand::debug\_dump\_calibration()
5. Add a case to the switch statement in TestStandCommController::recv\_calibrate(Calibration \*cal\_out)
6. Add a case to the switch statement in TestStandCommHost::calibrate(CalibrationKey key, void \*value)
7. Add a call to arduino\_calibrate(CalibrationKey key, void \*value) in arduino\_calibrate(Calibration \*calibration) in ArduinoHelper.cxx
8. Define an ODB key in feArduino.h
9. Define a global variable in feArduino.cxx initialized to the default value from DefaultCalibration.h
10. Update the Calibration struct in update\_calibration() in feArduino.cxx
11. Add a call to setup\_odb\_var(...) in frontend\_init() in feArduino.cxx

1. <https://www.thinksrs.com/downloads/programs/therm%20calc/ntccalibrator/ntccalculator.html> [↑](#footnote-ref-1)