Software Documentation for Virginia Tech Smart Infrastructure Lab's DAQ Collection System

Chreston Miller [1]

[1]Research and Informatics, University Libraries, Virginia Tech

September 3, 2015

Contents

List of Figures

Chapter 1

Overview

This is documentation for the Collection System for the Virginia Tech Smart Infrastructure Lab's (VT SIL) building health monitoring system for Goodwin Hall at Virginia Tech.

1.1 Overview of System Design

In Figure 1.1 in a high level outline of the system design and flow of operation. First, a connection is made with the specified DAQ(s). Then, the configuration parameters are loaded and the collection phase is initiated given the parameters. After a specified amount of time (from the configuration parameters) the data is written to a file. After which the collection cycle begins again. The collection system can be shutdown anytime. When shutdown, any currently held data is written to file and the connection to the DAQ(s) is closed.

1.2 Overview of Code Flow

In Figure 1.2 is an outline of the flow of operation according to the code. This is more detailed than the overview of the system design in Section 1.1. Note that each file has an explanation in Section 2. Following the figure, the resource string from the command line (more details on the resource string in Section 4.1) is used to initialize the the session with the DAQ(s). The initialization function in DAQ.cpp (1) is called to set-up the connection (session) with the DAQ(s). This includes setting all the necessary parameters for the session. Here the configuration file (Section 4.2) is parsed and user specified parameters are set. Then the collection cycle is started by the DAQ_Interface object (2) in which the DAQ object's function continuousRawData() (3) is called. This function talks to the DAQ(s) and pulls records as they become available (4). Note that the settings within the code is set-up so the user does not need to worry about setting the Record Size (Section 4.3). The Record Size is set to the Sample Rate (Section 4.3) so that one record equals one second of data. After a certain number of records are collected (5) writeDaqDataToFile() is called and the CSV_Output object writes the data to disk (6). Since a record equals one second of data, the number of records collected per file is specified in the configuration file. Then in (7), the buffer holding the data is flushed and the collection starts again for the next time slice (8). At any point during the collection phase, the user can press 'e' (for end) to close the session and shutdown (9). During this process, any data that remains in the buffer is written to disk, i.e., a file containing a partial time slice is written to disk.

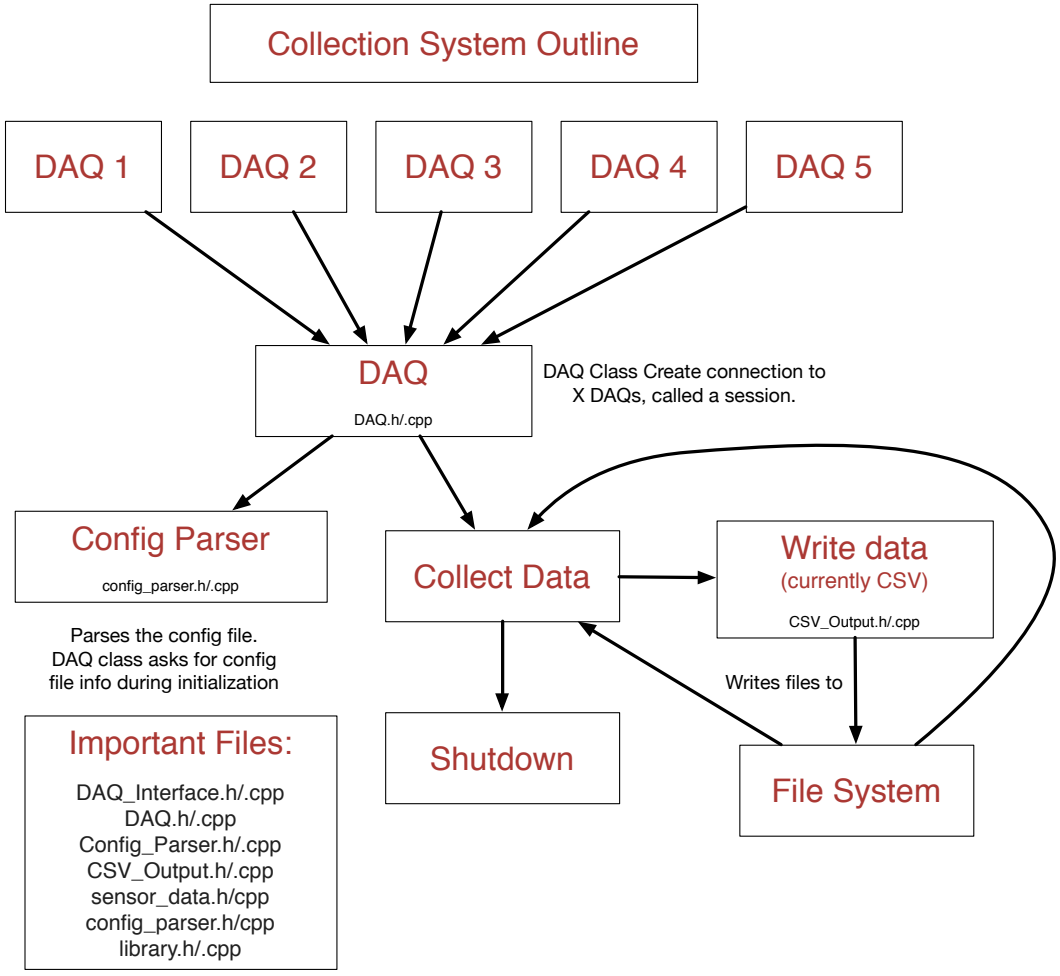

Figure 1.1: Collection system design.

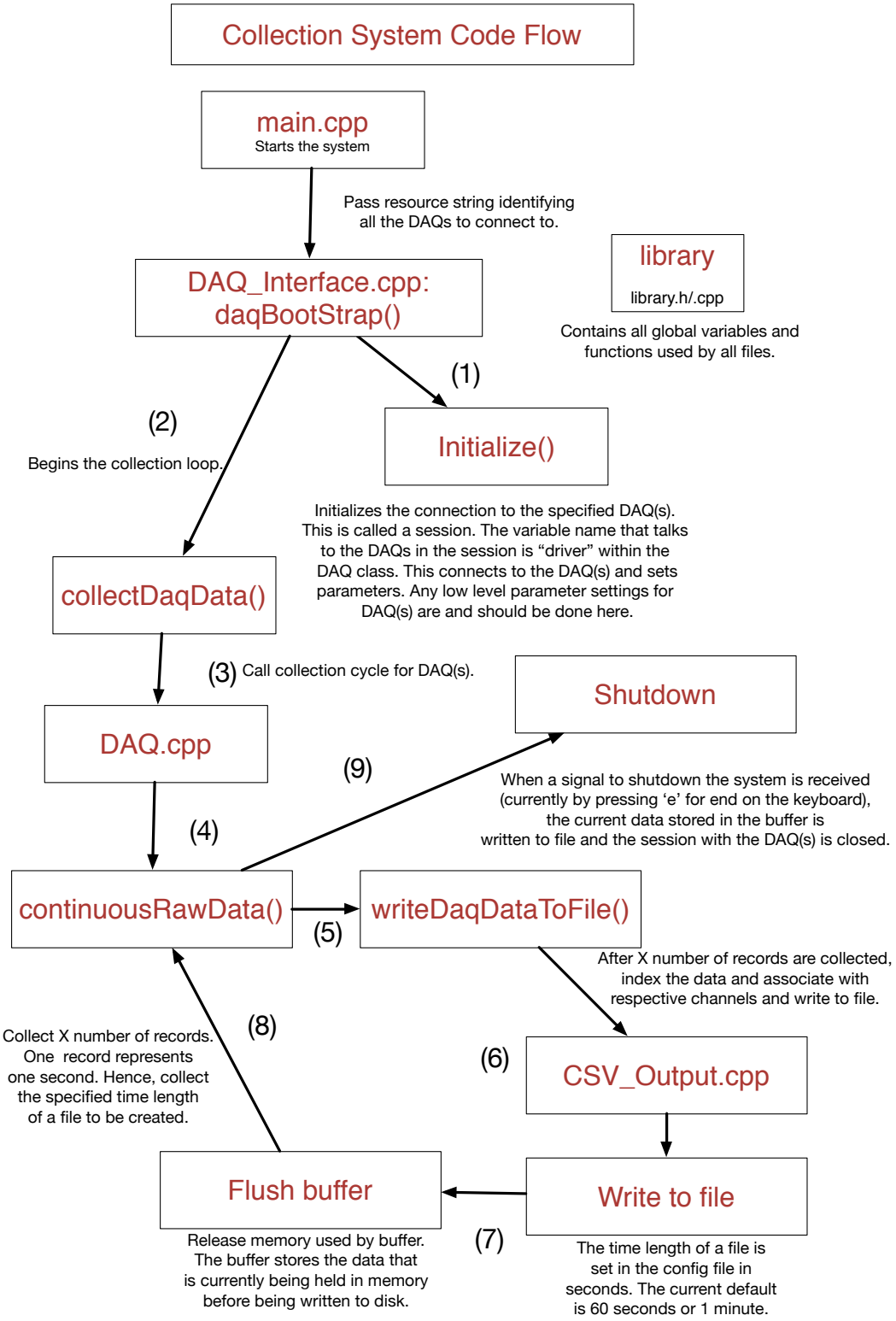

Figure 1.2: Collection cycle overview: The continuousRawData() stores data to a buffer until a specified timeframe has passed for collection, i.e., the time length of a file. Once this is reached, steps (5) → (6) → (7) → (8) are executed. Here the data is written to file, the buffer is flushed, and then data is collected for the next timeframe.

Chapter 2

Documentation for each Class

2.0.1 main.cpp

This is the normal main of any C++ code base. It starts the whole system through creating the appropriate object(s) and then makes the function (of the appropriate object, currently the DAQ_Interface class, but should be changed to the DAQ class) call that starts the entire collection cycle.

2.0.2 DAQ_Interface.h/cpp

This class was initially used to act as a controller for all the DAQs the collection system connects to, an interface to them. Each DAQ would have an instance of the DAQ class (next section) to handle interacting with the respective DAQs. Each instance would create a connection to a DAQ through what is called a *session* historically named *driver* in the code provided by VTI. However, after learning more about how to synchronize the DAQs, only one session is needed and hence, only one DAQ class instance. This resulted in the DAQ_Interface acting as a controller for operating the one DAQ instance.

2.0.3 DAQ.h/cpp

This class was initially meant to represent a connection (session - defined in previous section) to a single DAQ. After learning more about how to connect and synchronize to multiple DAQs, only one instance of this class was needed. Hence, one object of this class is used to control all the interactions with the DAQs. These are the following operations performed by this class:

- **Initialize:** Establish a connection to DAQ(s) connected to the local network given a resource string defining the IP addresses or hostnames of each DAQ. Here all the desired parameters of the DAQ(s) are set such as the sample rate and system flags such as collecting only from DAQ channels with sensors.
- **Collect Data:** As the DAQ(s) to provide a record of data from each. The data is returned in records which is defined in Section 4.3. Once a specified number of records is collected, the data is written to a specified medium, e.g., a CSV file.
- **Write to Medium:** This simply passes the information needed to the specified output medium for saving the data. After this is done, the currently held data is flushed for the next batch (memory released).

2.0.4 Data_Output.h/cpp

This is a base class for performing inheritance in which multiple output methods can be defined, e.g., CSV or HDF5. Currently only CSV is defined, but using inheritance allows for flexibility to define others in the future.

2.0.5 Data_Output_Factory.h/Data_Output_Factory_Implementation.h

This is how different output types are created in C++. The factory creates a specified output object, e.g., a CSV object of the CSV_Output class (next section). The implementation is written in a .h file as is historical practice.

2.0.6 CSV_Output.h/cpp

This class provides the method to write the DAQ data to a CSV file. A majority of this class is in the method that takes in the appropriate parameters from the DAQ class object so as to be written to file. This class contains the logic for indexing the DAQ data so as to organize the data by channel.

2.0.7 config_parser.h/cpp

This class parses the configuration file that defines specific user defined parameters. The format of this file is defined in Section 4.2. The DAQ class uses this class to set respective parameters for the DAQ(s). Currently, the name and location of the configuration file is hard-coded to be called "config_file.txt" located in the build directory.

2.0.8 library.h/cpp

This contains all the shared functions and global variables. This allows multiple classes to use the same functions and variables for various tasks, e.g., a conversion function or a global flag.

2.0.9 sensor_data.h/cpp

This function acts as a wrapper for data from the DAQ(s). This data includes the SAFEARRAY containing the actual data and the BSTRs containing the timestamp information.

Chapter 3

Hardware

- **Hardware Inventory:** This spreadsheet lists all the DAQs, cards, etc.
- **Map of Accelerometer Locations:** PDF map of where the accelerometers are located in Goodwin Hall: Accelerometer Locations
- **DAQ User Manuals:**
  - Manual for EMX 2500
  - See also the VTI Instruments product page for the EMX 2500 manual.
  - Manual for EMX 4250, 4350, 4380
  - See also the VTI Instruments product page for the EMX 4250, 4350, 4380.

Chapter 4

Software

This chapter details the software of the collection system.

4.1 Data Pipeline

Here we describe how the data gets to the collection system and its journey through the collection system until it is written to disk:

1. A session is opened that connects to X number of DAQs - X is defined in the session connect call. E.g.,
   - driver = new IVTEXDsaPtr(__uuidof(VTEXDsa)); → create the object (named driver) that will talk to the DAQ
   - driver→Initialize(resource_str, VARIANT_FALSE, VARIANT_TRUE, ""); → connects to the DAQs defined in the resource_str (list of address strings), format is: "TCPIP::IP_address_1::INSTR|TCPIP::IP_address_2::INSTR|..." where "|" is a vertical pipe.
   - If a hostname is given to a DAQ, that can be used in the place of the address strings: "DAQ-4E.local|DAQ-1E.local|...". Note the ".local" is needed to tell the system it is a local hostname. The actual host name does not have the ".local". **The order of the hostnames is the order the data is given by the DAQs**. I.e., if the first hostname is DAQ-4E.local, then the first X number of channels will be from the DAQ with that hostname, where X is the number of channels on that DAQ. If the second DAQ is DAQ-1E.local, then the next Y channels will be from the DAQ with that hostname, where Y is the number of channels on that DAQ.
   - This allows LAN synchronization in which the connected DAQs are time synced, i.e., all samples from all DAQs are synchronized in time.
   - This is useful since when the data comes in, it is in the same format as if data from a single DAQ, hence, no changes needed in code after the few lines for synchronization.
2. During the initialization of the session with the DAQs, the configuration file is read in.
3. Then collection of data is started. The data is read from the DAQ memory (memory stream call). Each record for each channel in all DAQs are pulled in parallel making for fast acquisition.

4. After a specified number of records are pulled from the DAQs, those records are sent to be written to disk.

    - A file containing the sensor data (sensor readings and timestamps for all sensors) is created in for a specified time slice.
    - E.g., Each file can represent data from a 1 minute time slice or 10 minutes. **The setting for this is in seconds.**

4.2 Configuration File

A configuration file (config file) is provided to set certain parameters to specified values. This allows the user to specify relevant values without needing to set them in the code. The format consists of a header describing the different lines of the config file. These lines begin with a '#' to signify to the parser to ignore them. For lines that contain parameter values, they are numbered with a value following. A semi-colon delimiter (';') is used between the numbered line and the value since it is an uncommon character. If a colon ':' or comma ',' are used, there may be parsing problems given some values may include them. An example of a config file is as follows:

```
#DAQ configuration set-up description
#All lines begin with a special identifier for a values
# 0; B → set clock frequency
# 1; C → set prescaler
# 2; D → set sample rate
# 3; E → length in seconds of each file produced
# 4; Move Path: set the path where collection files are stored locally. NOTE: need quotes
    around path and \\instead of \.
# 5; Store all channel data: bool to specify to store all channel data, or just channels with sensors,
    value: 1 to store all, value 0: only with sensors
# 6; Display records to be collected on DAQ. This will show how many records are currently on
    the DAQ. This is good for monitoring if the records are being produced faster than can be
    collected and processed. This shows if the collection program can keep up with the DAQ.
#
# DAQ → begin of configuration parameters for DAQ
DAQ
0;204800
1;1
2;1600
3;60
4;"C:\\Users\\itsloaner\\Downloads\\SIL_Collection_Data"
5;0
6;1
#End of config file
```

4.3 Parameters and Data Structures

Here are the definitions of the parameters used by the system. This includes what the values mean, how it is calculated, and what each is based on.

Clock Rate or Clock Frequency:

- This is either 204,800 or 131,072.
- This can be set in the code: driver→Measurement→Sampling→ClockFrequency.
- Data Structure: integer

Sample Rate:

- This is the number of samples taken per second.
- This can only be set to a 2 divisor of the Clock Rate.
  - E.g., if the Clock Rate is 204,800, then the sample rates can be: 400, 800, 1600, 3200,..., 204,800.
- The Sample Rate is calculated using the equation: ClockFrequency\Prescaler 2^x where x is a number from 0 through 16.
- Data Structure: float

Prescaler:

- Either set to 1 or 5. Default is 1.
- I do not touch this as it affects the calculations of the other parameters (see equation above).
- Data Structure: integer

Record Size:

- The number of samples returned per record.
- A record is a unit used to store samples. The samples from the DAQs are returned in record form.
- E.g., if a record size is 10, then 1 record for 20 channels is 200 samples.
- **NOTE: This parameter is not set in the config file as it is just the way the DAQ driver handles the data. The record size is set automatically based on the defined sample rate. The record size is set to the sample rate so one record contains one second of data. Hence, a file time length can accurately be defined in increments of 1 second.**
- Data Structure: integer

Record Size Sample Rate:

- This gives the time slice of the samples in a record.
- E.g., for Sample Rate 1600Hz and Record Size 160, then $160/1600 = 0.1$ samples per second $\rightarrow$ returning samples every tenth of a second.
- This shows up as the samples in each record within a tenth of a second will have the same time stamp (e.g., 0.1). The next set of samples will have the time stamp 0.2 and so on.

Sample Rate * Number of channels:

- This is the number of samples collected every second.

Channel ID:

- This is the unique identifier for each channel.
- The channel ID is returned along with the sensor data.
- The format of the channel ID is:
  - DAQ#<no space>Card Slot #<no space>!<no space>Channel #
- E.g., For DAQ 2, card slot 3, channel 6:
  - 103!CH6
  - DAQ#<no space>Card Slot #<no space>!<no space>Channel #
  - **NOTE: DAQ # is in increments of 10, so DAQ 1 is <empty, no space or anything>, DAQ 2 is 10, DAQ 3 is 20, and so on.**
- Data Structure: BSTR

Timestamps:

- The timestamp is provided in 2 pieces: Seconds (variable name: ts_sec), and fractions of a second (variable name: ts_frac).
- The seconds is the Unix epoch time, i.e., the number of seconds since January 1st 1970.
- The fractions of a second provide accuracy to the nanosecond.
- Data Structure: SAFEARRAYs for each. Both SAFEARRAYs are synced with each other as in index i for ts_sec corresponds to index i in ts_frac. Then both pieces together give the full timestamp which allows higher accuracy.
- Both variables are SAFEARRAYs that are the same size as the channels bstr. When requesting one record from a memory read, each sample in a record (record is X number of samples) for each channel has the same timestamp. Hence, everything in one record has the same timestamp.
- To provide an exact timestamp for each sample from the DAQ(s), the timestamp of the first record of a file is recorded along with the sample rate and the length of the file (in seconds). Then with this information, the timestamps for each sample can be calculated.
- In other words, to get the exact time for each sample, one needs the time of the first sample (t_0) collected and the sample rate (f_s). Where the time step (Δt) is calculated:

t_0
$t_0 + \Delta t$
$t_0 + \Delta t * 2$
...
...
$t_0 + \Delta t * n$

Table 4.1: Example calculation of the time of each sample in a file. Note: n is the last line in the file.

4.4 Setting up Development Environment

Here is the list of software and steps for installing the needed software to develop with the DAQs.

- A tool to identify LXI devices that are attached to the network must be installed: LXI Discovery Tool. This is needed since it forces you to install Apple Bonjour print services which is required to connect to the DAQs via their hostname (and not IP).
- Have the latest version of Microsoft Visual Studio. This is used to provide the computer with a compiler.
- In order to interface with the DAQ you have to install the following (in this order, **VERY IMPORTANT!**):
  1. Windows SDK (windows 7) and .NET Framework 4
  2. IVI Shared Components
  3. VTI's EMX Driver
- In order to interface with the thermal coupler hardware (EM-1048A):
  1. The PNP driver assuming 64 bit WindowsOS.
  2. The IVI driver.
  3. Password for access to IVI driver to make changes through web interface: ex1048
- For Qt development environment (includes Qt Creator) - available from `https://www.qt.io/download-open-source/`

4.5 Software Versioning

Here is described a simple introduction to software versioning. Versioning is used to keep a history of changes and allow a safety net for development and protect against breaking a code base. In Figure 4.1, we see a simple flow of how versioning (or version control) works. One starts with a *master* version of the code. This is the stable version that is meant to see the light of day. This version is never touched unless what is being added is known to be stable. Because of this, a *working development* branch is created for development. When a new feature wants to be tested, a new branch is created (*cool feature 1*). The code in this branch is developed and tested until the operation is satisfactory. Then that code is merged back into the *working development*.

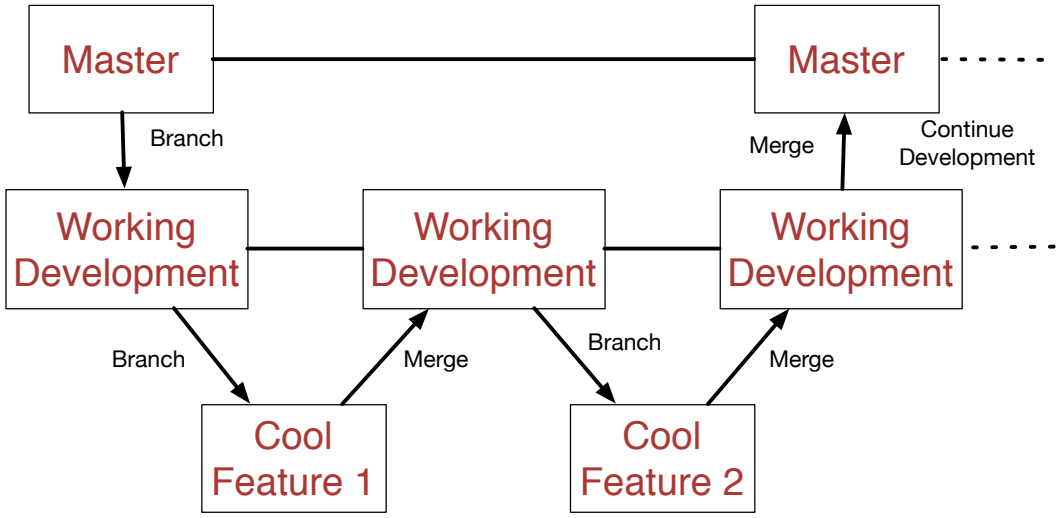

Figure 4.1: Example of how versioning works. One starts with a *master* version of the code. Then a *working development* branch is created for development. When a new feature wants to be tested, a new branch is created (*cool feature 1*). When the operation of the feature is satisfactory, that code is merged back into the *working development*. When the *working development* is stable, it is merged back into *master*.

Now *working development* has the new cool feature 1. This is continued until a stable version of *working development* is reached and is ready to be merged back into *master*. After which, the cycle continues for the life of the code.

4.6 Code Base and Github

The C++ code base is stored in a GitHub repository at

- https://github.com/VTUL/VT-SIL-Collection-System

To gain access to this repository, you can fork or clone it. To clone (i.e., download project and entire version history) use the following command from the github power shell:

- git clone https://github.com/VTUL/VT-SIL-Collection-System

It will download the code base into the folder "smart-infrastructure-laboratory" in the current directory.

Assuming, the user already understands versioning of software, the following commands are what is needed for development:

- git branch
- git branch [branch-name]
- git status
- git add
- git rm

- git commit -m “useful message”
- git push origin [current branch-name]

Please refer to Appendix A for more details and explanations on git commands.

When committing changes and saving to github, perform the following commands:

- git status → See what files have been changed.
- git add → Add any files that were changed (one ‘git add’ per file)
- git commit -m “useful message” → Commit changes to the local repository on the computer. The message is for describing what the changes are.
- git push origin [current branch-name] → Pushes the changes to github.

To see what the current branch-name is, use the command: git branch.

4.7 Running the System

Here is described how to run the system for collection of data. The system can be run from a console with a simple command. To do this, what needs to be known is:

- Location of executable.
- Location of configuration file.
- The DAQ network IDs that one wants to connect to.

In the current setting for development and since github is being used, the location of the source code will be:

- C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface

Where <user name> is the account name currently logged into on the computer.

The locations of the debug and release versions of the executable will be:

- Debug: C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface\build\debug
- Release C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface\build\release

If running the system from an IDE such as Qt Creator, the config file (named daq_config.txt) must be in the following location:

- Debug: C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface\build

If running the system from the command line, the config file (named daq_config.txt) must be in the following location for debug and release, respectively:

- Debug: C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface\build\debug\
- Release: C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface\build\release\

To run **using Qt Creator**, double-click the project .pro file located in:

- C:\Users\<user name>\Documents\Github \VT_SIL_DAQ_Interface\DAQ_Interface.pro

Or use the alias on the desktop (if available). To enter which DAQ(s) to connect to, go to the Project settings on the left-hand side, then to Run (top of settings screen). There is a line for arguments. Enter the DAQ resource string as described in Section 4.1. While your cursor is in the arguments box, hit 'enter' then save (control+s) to have the resource string you just entered to stick. I do not know why, but this must be done.

To run from **command line**, open a power shell using Github, an alias should be on the desktop. The power shell is provided when installing Github for Windows. Using the power shell, navigate to either the debug or release folder described above. Then the command to run is:

- DAQ_Interface.exe "<resource string>" <then hit enter>
- Where <resource string> is as described in Section 4.1.
- An example: DAQ_Interface.exe "DAQ-4E.local"→ to start the system connecting to DAQ-4E
- An example: DAQ_Interface.exe "DAQ-4E.local|DAQ-1E"→ to start the system connecting to DAQ-4E and DAQ-1E (in that order).

To stop the system type 'e' for 'end' while the power shell window is in focus.

4.8 Technical Details

Described here are details about using the IVI driver for the DAQs in terms of working with their data structures. Think of this is as a "lessons learned" section for technical details.

- The data format used by the DAQs for storing sensor and other data: For each record returned by the DAQ, the sensor data is one long array of values with the first X values belonging to channel 1, the next X to channel 2 and so forth, where X is the record size, i.e. how many sensor points per record.
- All this data is stored in a SAFEARRAY, a data structure optimized for passing to COM objects (e.g., storing in shared memory so another program can use it, just like a copy-paste clipboard). To access the sensor data, the example code for the DAQ driver shows how to iterate through the sensor data one value at a time.
- To speed up the system, the SAFEARRAY for each record is stored to avoid iterating through the data more than once. Then when it came time to write to disk, iterate through the SAFEARRAY by "jumping" around to pull out all the data for channel 1 in all the records, then for channel 2 and so forth so. This would put all the values for each channel together and lead to linear time access to the SAFEARRAY as each element is accessed only once.
- When accessing multiple elements from a SAFEARRAY, it is best to use SafeArrayAccessData() before accessing and SafeArrayUnaccessData() when done. An example can be seen here: https://msdn.microsoft.com/en-us/library/ms221620.aspx. Also a code snippet from the system is below:

```
 double *currentSensorDataDouble;
HRESULT hr;
// Get a pointer to the elements of the array.
hr = ::SafeArrayAccessData(currentSensorData->getData(), (void**)&?
    currentSensorDataDouble?);?


if (FAILED(hr)){
     cout<<"Error getting access to SAFEARRAY of sensor data. Aborting..."<<
    endl;
     return;
}

double dd = 0.0;
for (int i = 0; i < currentSensorData->getData()->rgsabound->cElements; i++)
{
     dd = (double)currentSensorDataDouble[i];
     outputStdStr.append(std::to_string(dd));
}
```

- Accessing the data this way, allowed writing the data to a CSV file with each column representing the sensor data from one channel.

4.9 Challenges and Solutions

Here is described a number of challenges faced along the way. Solutions to some of these challenges are provided, but not all currently have solutions.

4.9.1 DAQ Buffer Filling Up

This by far has been the biggest challenged faced to date. This entailed writing sensor data to disk fast enough to keep up with the data streaming in from the DAQ(s). On can find out if the collection system is keeping up by pinging the DAQs for how many records are currently ready. If that number is increasing, then the onboard DAQ memory will slowly fill up and sensor data will eventually be lost.

Methods performed to streamline speed:

- Not using Qt in the collection system. Using the QString data structure for storing data before writing to disk ended up being slow since the QString needed to be converted to a standard C++ string to be written out. This conversion was very slow with so much data (5 times slower).
- Changing to using standard C++ strings speed up the collection system considerably.
- Simplifying the file I/O to one write per file created. This is done by collecting all the sensor data for Y records, then writing to disk once. The sensor data is stored in a C++ standard string before being written to disk. The string is used to provide proper formatting for the output file, e.g., CSV file with header(s). Once written, the memory (RAM) used by this string is reclaimed by the computer.

- One thing discovered is any method for writing a string to file was about the same speed. This includes using the C++ output operator << and C style writing to disk (outStream(buffer,size) where size is the length of the buffer).

It was discovered that the source of the slowdown is converting the sensor value (in the form of a double) to a string that can be written to disk. It is this conversion from a floating point number to a string that is considerably slow. Currently, there appears to not be any conversion that is faster. In the future using different output file types, e.g., HDF5 or binary, may increase the speed. However, when the system is run in 'release mode' instead of 'debug mode', the conversion is $\sim$ 15 times faster. Due to the speed provided by release mode, the entire collection system can run and keep up with the data from the DAQ(s).

4.9.2 Slow-down in Windows 8.1

There appears to be a known problem in Windows 8.1 where code compiled using Microsoft Visual Studio compiler VS2013 is $\sim$ 50% slower than in Windows 7. I observed this when running the code on the SIL desktop. On the Windows 7 laptop used for development, the system was faster despite the fact that the desktop is significantly more powerful. If possible, it is recommended to use Windows 7 for the new server. However, the server should have Windows Server (2008 or 2012) at some point. But before that, testing running the system on Windows Server would need to be done on a separate computer to verify operability.

4.10 Data Storage

The data is sent to the Advanced Research Computing (ARC) group at Virginia Tech for storage: http://www.arc.vt.edu/. There is a script that performs sending the data to ARC.

[[what is this script??]]

Chapter 5

Currently Known Problems

Here is describe the currently know problems with the system. This can be seen as a 'Todo' list of important problems to be addressed for future development.

1. **Large memory leak:** Somewhere is a large memory leak. A potential source is the BSTRs not used that are returned from the Memory Read. These may need to be stored in the Sensor_Data class like the sensor data and time stamps, then release in the same manner. When trying to release these BSTRs in the collection loop, it works with 1 DAQ but not with 5 for some reason.

Appendix A

Github Reference Sheet

Useful Terminal Commands

cd	With no argument, the working directory changes directory to */home/user* abbreviated by ~. Otherwise, the current directory changes to specified target.
pwd	Displays the current directory.
ls	With no argument, un-hidden files and directories in the current directory are displayed. Otherwise, contents of target directory are displayed.
mv	Moves target file or directory to new location
cp	Copies target file or directory to target location
rm	Removes target file or directory
man	displays an extensive manual for the argument given

INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Git for All Platforms
http://git-scm.com

CONFIGURE TOOLING

Configure user information for all local repositories

```
$ git config --global user.name "[name]"
```
Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```
Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```
Enables helpful colorization of command line output

CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

```
$ git init [project-name]
```
Creates a new local repository with the specified name

```
$ git clone [url]
```
Downloads a project and its entire version history

MAKE CHANGES

Review edits and craft a commit transaction

```
$ git status
```
Lists all new or modified files to be committed

```
$ git diff
```
Shows file differences not yet staged

```
$ git add [file]
```
Snapshots the file in preparation for versioning

```
$ git diff --staged
```
Shows file differences between staging and the last file version

```
$ git reset [file]
```
Unstages the file, but preserve its contents

```
$ git commit -m "[descriptive message]"
```
Records file snapshots permanently in version history

GROUP CHANGES

Name a series of commits and combine completed efforts

```
$ git branch
```
Lists all local branches in the current repository

```
$ git branch [branch-name]
```
Creates a new branch

```
$ git checkout [branch-name]
```
Switches to the specified branch and updates the working directory

```
$ git merge [branch]
```
Combines the specified branch's history into the current branch

```
$ git branch -d [branch-name]
```
Deletes the specified branch

Figure A.1: Github reference sheet (page 1).

REFACTOR FILENAMES

Relocate and remove versioned files

```
$ git rm [file]
```

Deletes the file from the working directory and stages the deletion

```
$ git rm --cached [file]
```

Removes the file from version control but preserves the file locally

```
$ git mv [file-original] [file-renamed]
```

Changes the file name and prepares it for commit

SUPPRESS TRACKING

Exclude temporary files and paths

```
*.log
build/
temp-*
```

A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

```
$ git ls-files --other --ignored --exclude-standard
```

Lists all ignored files in this project

SAVE FRAGMENTS

Shelve and restore incomplete changes

```
$ git stash
```

Temporarily stores all modified tracked files

```
$ git stash pop
```

Restores the most recently stashed files

```
$ git stash list
```

Lists all stashed changesets

```
$ git stash drop
```

Discards the most recently stashed changeset

REVIEW HISTORY

Browse and inspect the evolution of project files

```
$ git log
```

Lists version history for the current branch

```
$ git log --follow [file]
```

Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```

Shows content differences between two branches

```
$ git show [commit]
```

Outputs metadata and content changes of the specified commit

REDO COMMITS

Erase mistakes and craft replacement history

```
$ git reset [commit]
```

Undoes all commits after `[commit]`, preserving changes locally

```
$ git reset --hard [commit]
```

Discards all history and changes back to the specified commit

SYNCHRONIZE CHANGES

Register a repository bookmark and exchange version history

```
$ git fetch [bookmark]
```

Downloads all history from the repository bookmark

```
$ git merge [bookmark]/[branch]
```

Combines bookmark's branch into current local branch

```
$ git push [alias] [branch]
```

Uploads all local branch commits to GitHub

```
$ git pull
```

Downloads bookmark history and incorporates changes

Figure A.2: Github reference sheet (page 2).