Developing a software interface between Dräger Winaccess API and influxDB Time series database

by

Benn Blessing,

dissertation

Presented to the Faculty of the Graduate School of

University of Southern Queensland

in Partial Fulfilment

of the Requirements

for the Degree of

Bachelor of Computer Systems Engineering (Honours)

University of Southern Queensland

October 2021

Developing a software interface between Dräger Winaccess API and influxDB Time series database

Approved by Dissertation Committee:

Abstract

This project aims to provide a software solution for the comprehensive capture and storage of vitals signs data and in doing so evaluate the suitability of a time series database for storing and accessing this data. The source of the vital signs data will be the API interface on the Draeger Infinity Gateway server, which in turn captures live vital signs data from the Draeger patient monitoring platform.

Numeric values are routinely exported from patient monitoring platforms to electronic medical record in the multi vendor HL7 format (Health Level 7), however typical HL7 exports lack the continuous waveform data which is often sought by clinical research projects. In addition, by developing an understanding of the function of the underlying proprietary Infinity paitent monitoring Network, we will extend the solutions capability to recording data from multiple simultaneous source devices and incorporate recording of alarm events. This will require a multithreaded model to maximise the number of simultaneous beds recorded. The solution is built using C++17, InfluxDB as the database solution, influxDB-cxx and several libraries from the Boost C++ project. It is a windows only solution in order to interface with the vendor supplied Windows DLL which provides the API call functions.

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged. I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Benn Blessing

[redacted]

Contents

Abstract **i**

Chapter 1 Background **1**

1.1 Applicable environment 1

1.2 Current state 2

1.2.1 Existing solutions 3

1.2.1.1 Cambridge University ICM+ 3

1.2.1.2 ADinstruments Lab Chart Lightning 3

1.3 Project aims 3

1.4 Knowledge gap 4

Chapter 2 Literature review **6**

2.1 The Infinity network 6

2.2 The Infinity Gateway Server 7

2.2.1 Dräger Infinity Gateway Protocol Handbook 7

2.2.2 The Winaccess API Test tool 8

2.2.3 The WvApi.h header file 9

2.3 Time series database 9

2.4 Physionet and the MIMIC databases 10

Chapter 3 Methodology **11**

3.1 Resource Requirements 11

3.2 Application development 11

3.2.1 Data types 12

3.2.2 Data sources 13

3.2.3 Multithreading 13

3.2.4 Winaccess API Header file 14

3.3 Building against the DLL 15

Chapter 4 Testing **16**

4.1 Throughput 16

4.2 Data validity 16

Chapter 5 Results **18**

5.1 Data size 18

5.2 http write time to influxDB 20

Chapter 6 Conclusion **21**

Bibliography **23**

Appendices **25**

.1 Early Progress September 2020 25

.2 example sequence per Vendor supplied header file 26

.3 Main.cpp from project 27

.4 WvBed.cpp Bed object implementation from project 29

.5 WvApiObj.cpp general implemenation functions from project 40

Chapter 1

Background

1.1 Applicable environment

In a modern hospital setting, patient monitors capture vital signs data that is useful both for clinical diagnosis, and also for clinical research. This data consists of a range of waveforms, periodic measurement values, and alarm messages.

Waveforms are typically sampled in the range of $100Hz$ to $500Hz$. A well known patient waveform is ECG (electrocardiogram). An ECG waveform, or trace, is the voltage signal measured across one or more cross sections of the heart. As the heart muscle contracts, a voltage is measured between various electrode pairs placed on the patients skin.

In higher acuity or operating theater environments, blood pressure is measured using fluid lines and pressure transducers producing a pressure waveform. Other waveforms may include airway pressures, respiratory flows, and gas concentrations.

Non waveform values will be taken at variable intervals. Heart Rate or blood oxygen saturation (SPO2) may be recorded every second. The automatic blood pressure cuff commonly seen in a doctors office is also a common measurement referred to as NIBP (non-invasive blood pressure). It may be set to measure periodically, for example every 15 or 30 minutes.

These measurements are used for

- Clinical review to guide treatment decisions
- Alerting clinical staff is a measurement breaches a configured alarm limit
- Populating data into electronic medical records (EMR)

- Clinical research

This project concerns making this data available for clinical research projects and assessing the suitability of the influxDB time series database for this use case.

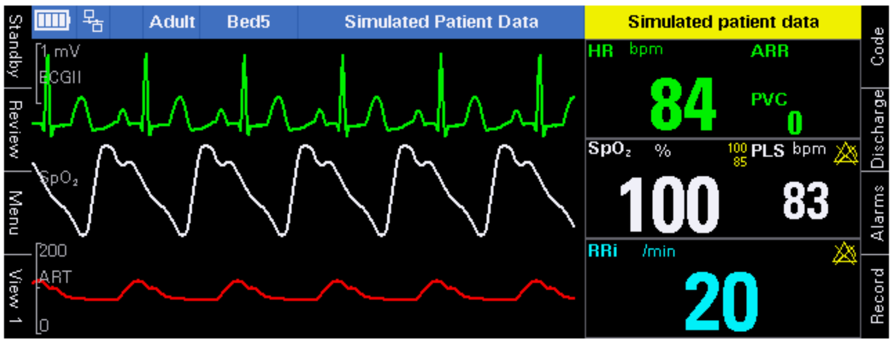


Figure 1.1:

1.2 Current state

When clinical teams seek to collect data from their bedside monitors for research purposes, they are often frustrated by the challenge of accessing data in a form that can be easily processed in a spreadsheet or using data processing packages such as R Studio.

A Common scenario might consist of a group of patient monitors which are networked such that data is visible on a connected computer running proprietary software. This device receives and displays the clinical data for review. Such a computer is commonly referred to as a Patient Monitoring Central Station. A Central Station records waveform, alarm and trend data for a limited period.

Data export is possible from Central stations, but this is typically not ideal for clinical research projects due to limitations on number of waveforms exported and proprietary data format for alarm, waveform and trend data.

In addition to Central monitoring stations, hospitals increasingly have networked data export in order to populate electronic medical records (EMR). These EMR typically accept data in the HL7 format (Health Level 7). HL7 does not typically support waveform or alarm data. The software interface between the patient monitoring network, and external systems such as EMR is typically referred to as a vendor gateway.

1.2.1 Existing solutions

1.2.1.1 Cambridge University ICM+

There is an existing solution from a team based at Cambridge University, called 'ICM+'(Cambridge 2021). It has support for a range of source devices connecting via either direct RS232 or over the network via vendor gateways. ICM+ has a capture driver for Dräger Winaccess API, which is the same interface this project will be targeting. However it differs in the following aspects.

- ICM+ requires the user launch an instance of the application and manually start a case file per bed space and patient, whereas this project will support simultaneous capture from multiple beds automatically.
- ICM+ records to a file in HDF5 format. This project is targeting influxDB time series database
- This project will capture alarm events which are not supported by ICM+

1.2.1.2 ADinstruments Lab Chart Lightning

Older patient monitoring platforms supported bulk waveform output via analog signals. These could be recorded via analog capture interfaces compatible with Lab Chart or similar data plotting platforms. Newer monitoring platforms only have limited analog output channels used for analog triggering, and not bulk waveform output so these bulk analog interfaces no longer meet this use case.

ADinstruments Lightning(ADINSTRUMENTS 2021) supports data capture via RS232 export, but does not currently provide an interface suitable for waveform capture via custom API(adinstruments 2021).

1.3 Project aims

This project aims to provide a software solution that interfaces a patient monitoring vendor gateway (Dräger Medical Infinity Gateway) to capture desired waveforms, trend values and alarm events, and then store this data into a time series database. In doing so, the capability of the API will be investigated. Additionally, the suitability of the chosen time series database will be evaluated for this use case.

The database should have an easy to use interface that allows an end user to select parameters and time period for export to their preferred data processing platform.

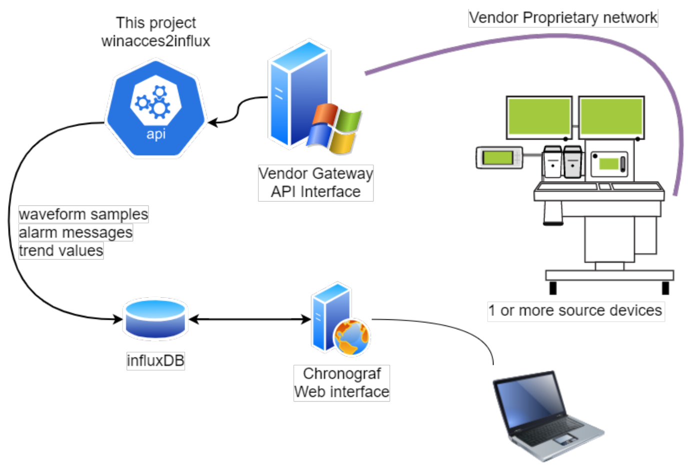


Figure 1.2:

The solution will be configurable to record data from one or more beds simultaneously, and the target trend and waveform parameters for capture will be selectable. This paper will refer to the developed tool as winaccess2influx. The data path is visualised in figure 1.2.

1.4 Knowledge gap

The API interface is provided by the vendor for use via supplied Windows DLL file, LIB file and C++ header files. The header file, WvApi.h, provides a broad description of the DLL calls. This project will develop a C++ application, winaccess2influx, to intelligently make suitable API calls in order build a useful data collection tool. Due to the nature of the API, the developed application must poll for data periodically, in addition to monitoring bed status.

Initial steps were to build a simple C++ program which includes the supplied header file, and successfully call a DLL function. This must then be extended to implement all functions needed. Steps such as

- Connect to API interface
- List online beds
- Connect to required beds
- list available parameters and waveforms
- Build waveform filter
- Request waveform samples, trend values and alarm messages
- process data into format suitable for sending to database
- provide status logging and configuration options

In addition, the project seeks to assess the suitablity of the time series database compared to simple dat file formats which store sample values in a bianry data structure rely on an initial timestamp combined with known sample interval.

Chapter 2

Literature review

2.1 The Infinity network

The Infinity network is the proprietary patient monitoring network protocol of the Dräger patient monitoring platform. It is built on top of standard TCP/IP and UDP network protocols. Groups of monitors communicate using multicast groups. The Gateway server which provides the API must also have multicast connectivity to the patient monitoring devices. This project does not interact directly with the Infinity network. The Vendor Gateway acts as the interface to the proprietary network and provides the provides the API to third party uses such as Winaccess2influx. However, an understanding of the Infinity protocol is helpful when using the API. the following details are relevant to the Winaccess2influx project.

- Waveforms are presented at 200Hz or 100Hz dependant on the parameter
- Alarm message status of each bed is updated once per second
- The bedside monitors include two timestamps in data that is sent. Ticktime and UTC time

Ticktime refers to a monotonic millisecond counter. This value is made available such that waveform data can be reliably transmitted wihout distortion in case of the UTC clock being adjusted. IE, if the realtime clock is adjusted backwards to match internet time, the ticktime counter will not go backwards, rather it will continue to increment every millisecond.

2.2 The Infinity Gateway Server

The Infinity Gateway server participates in the Infinity Network, joining the required multicast groups. It has access to the following data.

- Current parameter values at connected beds
- Last measured values for non continuous measurements, such as NIBP, and the time the measurement was taken
- A buffer of 2000 waveform samples per selected waveforms for connected beds (the Gateway server itself buffers this data)
- Current alarm status for connected beds
- Operating status of all beds in the configured multicast groups

The Gateway server makes this data available via several interfaces, in particular

- HL7 (Health Level 7) - A widely used multi-vendor protocol for exchanging data in the hospital evironment
- Winaccess API - A vendor specific interface supporting function calls via supplied windows DLL These functions and data structures are described in the supplied c++ header file WvApi.h. This interface provides more capabilities than HL7, including waveform and alarm data.

2.2.1 Dräger Infinity Gateway Protocol Handbook

The Dräger Infinity Gateway protocol handbook descibes the WinAccess API Developers Tool on page 23(DraegerWerk 2018). It does not fully describe usage of the DLL, but it contains some important guidelines for correct operation.

"The WinAccess API is implemented as a Windows DLL (Dynamic Link Library), and makes the data available through a set of functions callable from C or C++ programs."

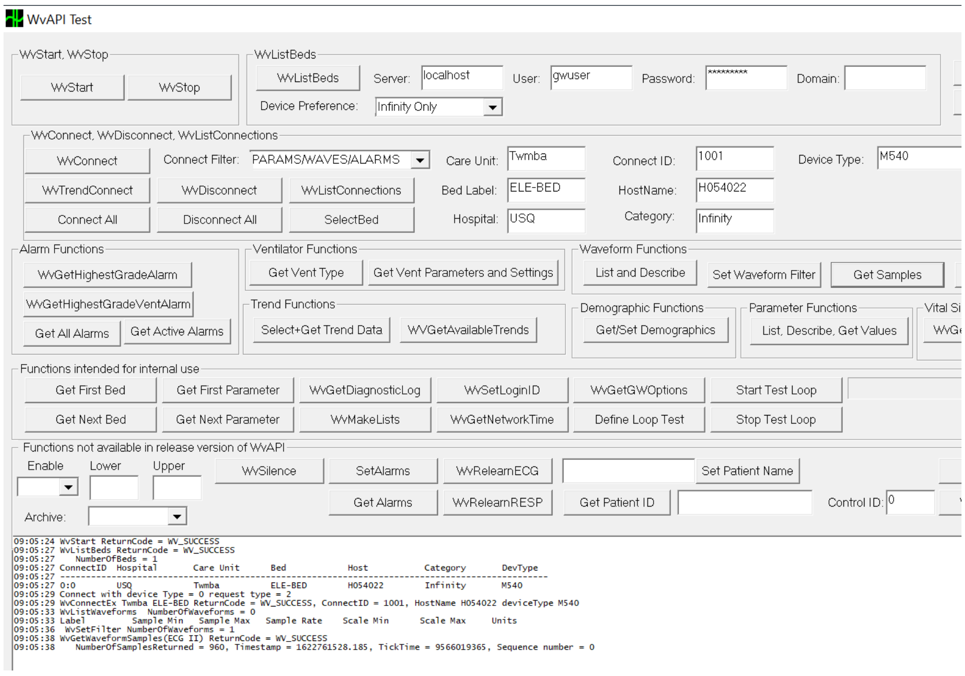
and also ***"CAUTION Each connection should be used by only one thread."***

A connection in this context is the establishment of a relationship between a bedside monitor and the Gateway server, where a connection ID is assigned following a call of the WvConnect DLL function.

This instruction to use a single thread per connection is an important guideline for the software design. This suggests a thread per bed is preferrable over a thread per data type (waveforms, trends, alarms).

2.2.2 The Winaccess API Test tool

The Infinity Gateaway Suite includes a test tool, which is a simple GUI application which is a sample implementation of each of the DLL calls. This is a useful tool to get started, but doesn't give much detail on how to implement a continuous capture tool. It proved useful in understanding the DLL functions and testing assumptions. A screenshot of the tool is shown in figure 2.1.



Server: localhost User: gwuser Password: ********* Domain:

Device Preference: Infinity Only

Connect Filter: PARAMS/WAVES/ALARMS Care Unit: Twmba Connect ID: 1001 Device Type: M540

Bed Label: ELE-BED HostName: H054022

Hospital: USQ Category: Infinity

Control ID: 0

```
09:05:24 WvStart ReturnCode = WV_SUCCESS
09:05:27 WvListBeds ReturnCode = WV_SUCCESS
09:05:27    NumberOfBeds = 1
09:05:27 ConnectID  Hospital       Care Unit      Bed            Host           Category       DevType
09:05:27 ----------------------------------------------------------------------------------------------
09:05:27 0:0        USQ            Twmba          ELE-BED        H054022        Infinity       M540
09:05:29 Connect with device Type = 0 request type = 2
09:05:29 WvConnectEx Twmba ELE-BED ReturnCode = WV_SUCCESS, ConnectID = 1001, HostName H054022 deviceType M540
09:05:33 WvListWaveforms  NumberOfWaveforms = 0
09:05:33 Label         Sample Min   Sample Max   Sample Rate    Scale Min      Scale Max     Units
09:05:36  WvSetFilter NumberOfWaveforms = 1
09:05:38 WvGetWaveformSamples(ECG II) ReturnCode = WV_SUCCESS
09:05:38    NumberOfSamplesReturned = 960, Timestamp = 1622761528.185, TickTime = 9566019365, Sequence number = 0
```

Figure 2.1:

2.2.3 The WvApi.h header file

The vendor supplied WvApi.h header file provides the most detailed description of the DLL functions, as well as an example sequence of function calls to be made to capture data. The header file is crifical to understanding the data structures. The developer must allocate data structures matching those in the DLL, and then pass pointers for those structures to the DLL function calls. Below is an example of a data structure defined in the vendor supplied header file WvApi.h.

```
typedef struct {
        WV_PARAMETER_ID         WvParameterID;                  // parameter id
        WCHAR                   Label[WV_LABEL_SIZE];        // label in english
        WV_NET_UNITS_OF_MEASURE Units;
        WCHAR                   Values[WV_MAX_TREND_SAMPLES][WV_VALUE_SIZE];
        WCHAR                   LowScale[WV_VALUE_SIZE];        // LowScale
        WCHAR                   HighScale[WV_VALUE_SIZE];       // HighScale
        WV_COLOR                WvColor;
} WV_TREND_PARAMETER_DATA_W;
```

2.3 Time series database

The data being collected consists of floating point or integer sample values, units of measure, parameter labels and timestamps.

Due to the high level of repetition, and the tendency for such data to be queried by time period, there is significant scope for a database to be optimised for time series data in terms of storage efficiency, write speed and query speed.

Whilst not the primary goal of this project, by comparing disk space usage between the data stored in InfluxDB, with the same data in CSV text format, an initial indication of benefit can be assessed. In the paper "Time Series Databases and InfluxDB" (Syeda Noor Zehra Naqvi 2018), a significant benefit in disk space usage and write performance was found when comparing influxDB to SQL. This project does not have output capability to SQL database so this could not be directly compared.

2.4 Physionet and the MIMIC databases

The Physionet group have been working since 1999 to build resources for the study of phsyiological signals including a repository of anonymised vitals signs datasets (Goldberger et al. 2000) and a collection of open source signal processing tools.. A sample from their dataset will be used to compare disk space usage. With further work, a tool or feature could be added to output data in a physionet compatible format to allow users to contribute to the MIMIC project.

Chapter 3

Methodology

3.1 Resource Requirements

In order to perform this project, access to the following was required

- A Dräger patient monitor with simulation mode to generate some test data
- A Dräger Infinity Gateway software Licence with Winaccess API option
- A Microsoft Windows based C++ development environment, I used the free Visual Studio 2019 Community edition
- The Winaccess API interface DLL and LIB files provided by Dräger
- The Dräger Winaccess API sample Header which lists the available function calls and defines the data structures
- Several freely available libraries developed by boost.org

3.2 Application development

The Methodology was to start with small examples and then extend capabilities until a usefully featured tool was created.

The first step was to compile a C++ project which incorporated the provided WvApi.h header, and which could call the WvStart function using the DLL. A git repository was created to track changes in the code(Blessing 2021). See Appendix "Early Progress September 2020" to see an early state of the source code which could call the WvStart WvListBeds functions and output the results to console.

As the DLL and Gateway software require windows, the project was built in Microsoft Visual Studio 2019. Several libraries were utilised to reduce the work required to reach a useful product, and also to improve maintainability.

From the boost.org project(boost.org 2021), the trival logging package was used for logging, and the property tree and ini_parser packages were used to support configuration file function. Using these tools made tasks such as creating a default config file in ini format relatively straight forward. The Boost logging package provided support for variable log levels for debugging or production use.

For writing to the database, influxdb-cxx(github 2021) was incorporated. This is a small library which handles writing to influxDB via http calls. This library also supports batch writes, which increases performances by grouping writes together before opening the http connection to the database. Using this library saved considerable time as it managed the HTTP database calls and interpreting the database response codes. Batching was also essential, as creating an http connection per sample proved far too slow.

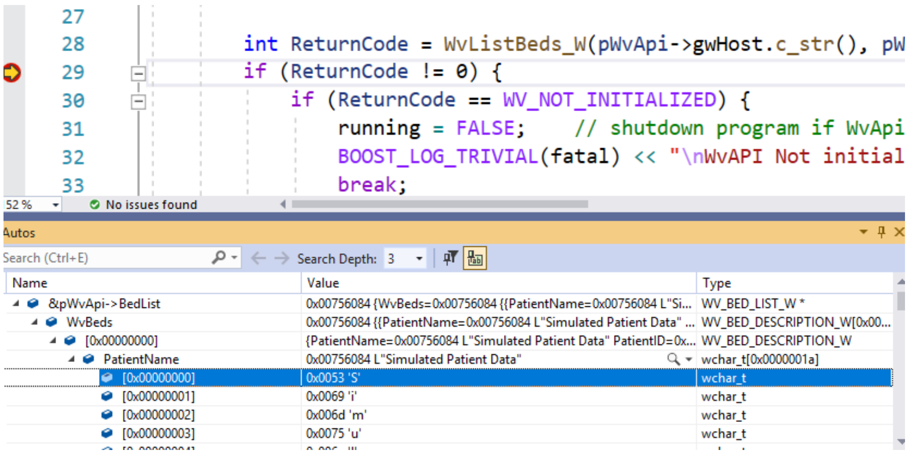
3.2.1 Data types

There were some windows specific data types involved with the header file and DLL. The API supports operation in unicode or ASCII mode. This project targets unicode mode to provide future flexibility.

The unicode implementation of the API is known as wide character UTF-16, using a fixed width 16 bit value to store a unicode character. From the debug window, these values can be seen in Figure 3.1.

As UTF-16 is not widely supported in modern software frameworks and web standards, the UTF-16 wchar arrays are converted to multibyte UTF-8 strings.

UTC Timestamps are defined in the supplied header file as time_t type. In this project, time_t was substituted for int32_t as it was found that the DLL was assuming a 32 bit time_t value, whereas the Visual Studio 2019 project was using a 64 bit time_t. This demonstrated a limitation when passing memory pointers to precompiled DLL, with the compiler unable to verify that the data structures defined in the project match those assumed when the DLL is compiled. The ticktime value is specified as INT64 and caused no issue. Emerging medical interoperability standards such as IEEE 11073 SDC(Rockstroh et al. 2017) avoid this scenario by offering SOAP (Simple Object Access Protocol) or REST (Representational State Transfer) interfaces but these



```
        int ReturnCode = WvListBeds_W(pWvApi->gwHost.c_str(), pW
        if (ReturnCode != 0) {
            if (ReturnCode == WV_NOT_INITIALIZED) {
                running = FALSE;    // shutdown program if WvApi
                BOOST_LOG_TRIVIAL(fatal) << "\nWvAPI Not initial
                break;
```

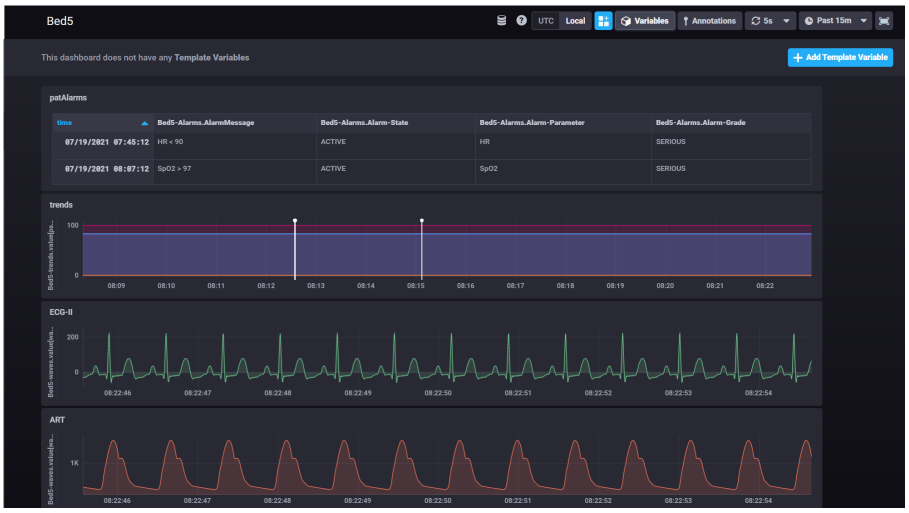
52 % No issues found

Autos

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
&pWvApi->BedList	0x00756084 {WvBeds=0x00756084 {{PatientName=0x00756084 L"Si...	WV_BED_LIST_W *
WvBeds	0x00756084 {{PatientName=0x00756084 L"Simulated Patient Data" ...	WV_BED_DESCRIPTION_W[0x00...
[0x00000000]	{PatientName=0x00756084 L"Simulated Patient Data" PatientID=0x...	WV_BED_DESCRIPTION_W
PatientName	0x00756084 L"Simulated Patient Data"	wchar_t[0x0000001a]
[0x00000000]	0x0053 'S'	wchar_t
[0x00000001]	0x0069 'i'	wchar_t
[0x00000002]	0x006d 'm'	wchar_t
[0x00000003]	0x0075 'u'	wchar_t

Figure 3.1:



Bed5

This dashboard does not have any Template Variables

patAlarms

time	Bed5-Alarms.AlarmMessage	Bed5-Alarms.Alarm-State	Bed5-Alarms.Alarm-Parameter	Bed5-Alarms.Alarm-Grade
07/19/2021 07:45:12	HR < 90	ACTIVE	HR	SERIOUS
07/19/2021 08:07:12	SpO2 > 97	ACTIVE	SpO2	SERIOUS

Figure 3.2:

are not yet in wide usage.

3.2.2 Data sources

To generate source data, a Dräger M540 standalone patient monitor in simulation was connected via ethernet to a Virtual Machine. For testing multiple bed capability, a software test tool which generates simulated data on the server loopback interface was used.

For generating alarm events, the alarm limits of the M540 standalone monitor could be adjusted to generate various alarms.

3.2.3 Multithreading

The Dräger Gateway protocol handbook states that the API supports mulitple threads, but only a single concurrent thread per connection. Early versions of this tool were single threaded, processing one bed at a time and then waiting for the next loop start time. This was effective for a small number of beds, bed would fail with a larger number of beds as the 10 second waveform buffer may overflow resulting in gaps in the waveform. In addition, due to the per bed alarm status being updated each second, a target was set of one second polling interval per bed. This would not be feasible using a single thread.

To move to a multithreaded approach, the code was refactored into an Object Oriented design, with one processing thread per object. This programming paradigm was found to be a good match, as each bed was treated as an object with local data structures, methods, timings and a single thread. It allowed each bed to be reliably polled for alarms every second, with waveforms and trends being captured every 5 seconds.

The main thread is then responsible for maintaining single map of bed labels and corrsesponding object. If a bed goes offline, the corresponding bed is dropped. This map object enforces the single thread per device rule from the API handbook.

In addition, to allow the main thread to push updated bedlist information (for example, a bed moving from discharge to monitoring state) into the bed object, Mutexes were used to prevent concurrent access to a bed object data by two threads.

To signal termination on program end, a global atomic variable was declared.

From winaccess2influx.h

```
extern std::atomic_bool running;
```

3.2.4 Winaccess API Header file

The information provided by the header file was manipulated in several ways to fit the program design.

Firstly, the header files contains several long enumeration lists, which translate a character sequence in the program source code to an integer which is passed to the DLL functions.

This enum approach did not present an easy way to match strings in the configuration file, such as selected waveforms, to an enum value. To work around this, sets of static hashmaps maps were built for matching user readable strings to the header file enums, and also translation API return codes to useful strings rather than numbers.

Some DLL functions were found to trigger a TCP connection to the bedside device, rather than relying on the multicast data. One example is requesting trend history from a device. As these calls were not required in this tool, they were removed from the header file. These calls can cause significant delay so were deemed best avoided unless necessary.

3.3 Building against the DLL

From the WvApi.h header file, we see the DLL functions are referenced using the `extern "C"` call. Given compatible data structures, a program can pass the expected values and memory pointers to the DLL funcitons, and the DLL gives a returncode and populates data into the memory locations.

```
#if !defined(WVDLLIMPLEMENTATION)
#define IMPORT_FUNCTION extern "C"

IMPORT_FUNCTION int WINAPI WvStart(int *pMajorRev, int *pMinorRev);
```

Chapter 4

Testing

4.1 Throughput

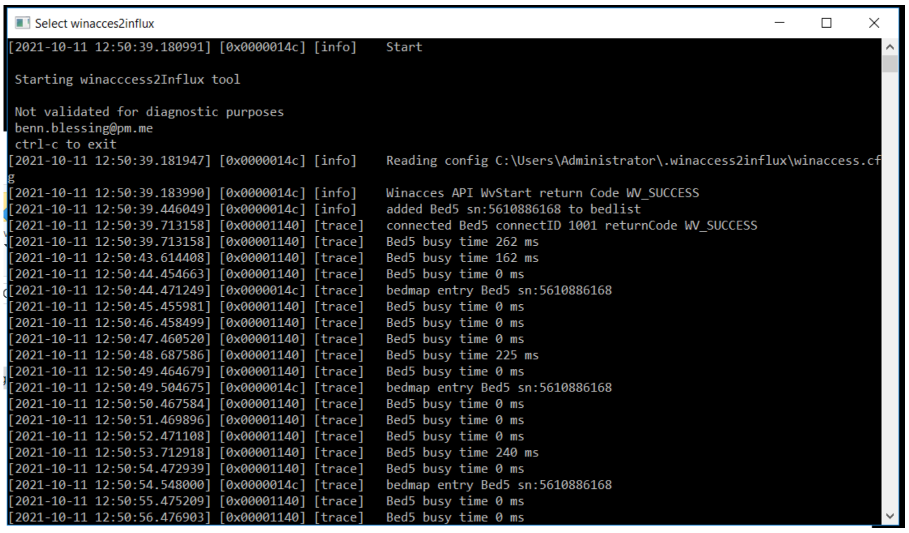
Testing was performed using the test monitor in addition to the patient emulator package. If the log level is set to trace, the developed application outputs the busy time per loop. This is the time take to perform scheduled queries to the API and also send data to the database. The shorter response times are due to the alarm query which is run once per second being a simpler task. The most data intensive task is waveform sample transfer which only runs every 5 seconds.

Capturing 3 waveforms and 11 trend parameters from a single bed, it can be seen in figure 4.1 that the busy time does not exceed one second with top of 262 ms. One second was chosen as target due to the alarm status being updated every second. By increasing to 11 waves 4.2 and 50 parameters for the same single bed, this increased to 541 ms.

An upper limit of 33 beds was chosen. This was simulated using a software tool which generates test data on the loopback interface of the server. These simulated beds had 14 waveforms and 75 trend parameters. In this scenario, the waveforms were still reliably captured, but the busy time intermittently exceeded 1 second. By limited the capture to 3 waveforms per bed, 1000 ms was only occasionally breached with 33 beds recorded.

4.2 Data validity

The values were compared between the monitor display and the chronograf gui. Care had to be taken for correct handling of floating point values, as some parameters such as temperature and gas concentrations have decimal units.

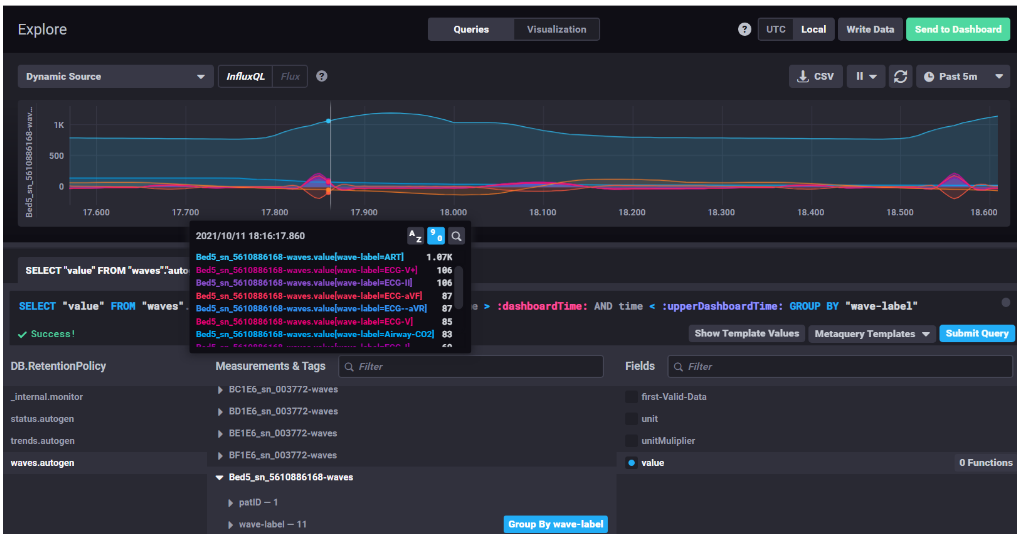


```
Select winacces2influx
[2021-10-11 12:50:39.180991] [0x0000014c] [info]    Start

 Starting winacccess2Influx tool

 Not validated for diagnostic purposes
 benn.blessing@pm.me
 ctrl-c to exit
[2021-10-11 12:50:39.181947] [0x0000014c] [info]    Reading config C:\Users\Administrator\.winaccess2influx\winaccess.cf
g
[2021-10-11 12:50:39.183990] [0x0000014c] [info]    Winacces API WvStart return Code WV_SUCCESS
[2021-10-11 12:50:39.446049] [0x0000014c] [info]    added Bed5 sn:5610886168 to bedlist
[2021-10-11 12:50:39.713158] [0x00001140] [trace]   connected Bed5 connectID 1001 returnCode WV_SUCCESS
[2021-10-11 12:50:39.713158] [0x00001140] [trace]   Bed5 busy time 262 ms
[2021-10-11 12:50:43.614408] [0x00001140] [trace]   Bed5 busy time 162 ms
[2021-10-11 12:50:44.454663] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:44.471249] [0x0000014c] [trace]   bedmap entry Bed5 sn:5610886168
[2021-10-11 12:50:45.455981] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:46.458499] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:47.460520] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:48.687586] [0x00001140] [trace]   Bed5 busy time 225 ms
[2021-10-11 12:50:49.464679] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:49.504675] [0x0000014c] [trace]   bedmap entry Bed5 sn:5610886168
[2021-10-11 12:50:50.467584] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:51.469896] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:52.471108] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:53.712918] [0x00001140] [trace]   Bed5 busy time 240 ms
[2021-10-11 12:50:54.472939] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:54.548000] [0x0000014c] [trace]   bedmap entry Bed5 sn:5610886168
[2021-10-11 12:50:55.475209] [0x00001140] [trace]   Bed5 busy time 0 ms
[2021-10-11 12:50:56.476903] [0x00001140] [trace]   Bed5 busy time 0 ms
```

Figure 4.1:



```
2021/10/11 18:16:17.860
Bed5_sn_5610886168-waves.value[wave-label=ART]        1.07K
Bed5_sn_5610886168-waves.value[wave-label=ECG-V+]       106
Bed5_sn_5610886168-waves.value[wave-label=ECG-II]       106
Bed5_sn_5610886168-waves.value[wave-label=ECG-aVF]       87
Bed5_sn_5610886168-waves.value[wave-label=ECG--aVR]      87
Bed5_sn_5610886168-waves.value[wave-label=ECG-V]         85
Bed5_sn_5610886168-waves.value[wave-label=Airway-CO2]    83
```

```
SELECT "value" FROM "waves".auto
SELECT "value" FROM "waves". … e > :dashboardTime: AND time < :upperDashboardTime: GROUP BY "wave-label"
✓ Success!
```

DB.RetentionPolicy
- _internal.monitor
- status.autogen
- trends.autogen
- waves.autogen

Measurements & Tags
- BC1E6_sn_003772-waves
- BD1E6_sn_003772-waves
- BE1E6_sn_003772-waves
- BF1E6_sn_003772-waves
- Bed5_sn_5610886168-waves
  - patID — 1
  - wave-label — 11

Fields
- first-Valid-Data
- unit
- unitMuliplier
- ● value

Figure 4.2:

Chapter 5

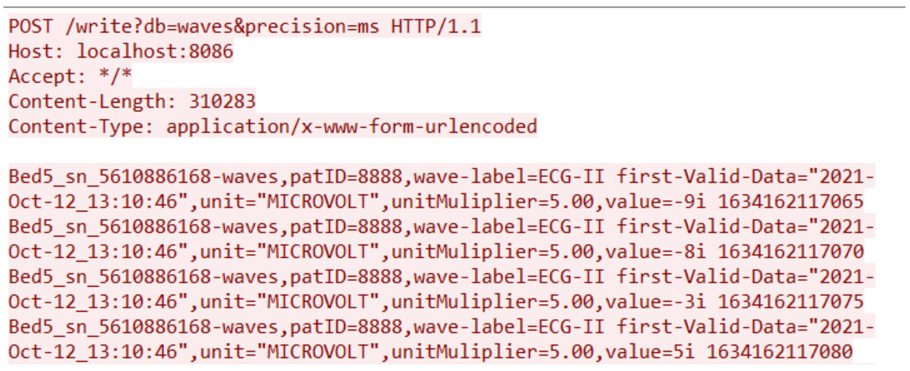
Results

5.1 Data size

For this analysis the focus will be on waveform data, as typical data size of trend values in this scenario is trivial in comparison. Figure 5.1 shows four lines from an http message which transfers sample data between the winaccess2influx tool and influxDB. It can be seen the millisecond timestamp at the end of each line incremements by 5 each time. This represenets individual samples in a 200 Hz ECG waveform. As can be seen, there is significant repetition, and to store data in this format would be highly inefficient. The expectation is that influxDB will reduce data usage by using an intelligent algorith optimised for time series data such as this.

The data usage of the influxDB database can only be approximately gauged, as the database allocates disk space in chunks, with default of 32MB. After running a single bed capture for 12 hours, the waves database had grown from an initial size of 32 MB, to 67MB.

The waveform data consisted of 3 waveforms, a 200 Hz ECG wave, a 100 Hz blood pressure wave, and a 100 Hz SPO2 waveform (blood oxygen saturation). Each sample is a 16 bit integer, with a



```
POST /write?db=waves&precision=ms HTTP/1.1
Host: localhost:8086
Accept: */*
Content-Length: 310283
Content-Type: application/x-www-form-urlencoded

Bed5_sn_5610886168-waves,patID=8888,wave-label=ECG-II first-Valid-Data="2021-
Oct-12_13:10:46",unit="MICROVOLT",unitMuliplier=5.00,value=-9i 1634162117065
Bed5_sn_5610886168-waves,patID=8888,wave-label=ECG-II first-Valid-Data="2021-
Oct-12_13:10:46",unit="MICROVOLT",unitMuliplier=5.00,value=-8i 1634162117070
Bed5_sn_5610886168-waves,patID=8888,wave-label=ECG-II first-Valid-Data="2021-
Oct-12_13:10:46",unit="MICROVOLT",unitMuliplier=5.00,value=-3i 1634162117075
Bed5_sn_5610886168-waves,patID=8888,wave-label=ECG-II first-Valid-Data="2021-
Oct-12_13:10:46",unit="MICROVOLT",unitMuliplier=5.00,value=5i 1634162117080
```

Figure 5.1:

Algorithm 5.1 csv export from influxDB

```
"time","Bed5_sn_5610886168-waves.first-Valid-Data","Bed5_sn_5610886168-waves.unit","Bed5_sn_5610886168-waves.unitMuliplier","Bed5_sn_5610886168-waves.value"
"2021-10-14T10:36:25.485+11:00","2021-Oct-12_13:10:46","MICROVOLT","5","38"
"2021-10-14T10:36:25.490+11:00","2021-Oct-12_13:10:46","MICROVOLT","5","37"
"2021-10-14T10:36:25.490+11:00","2021-Oct-12_13:10:46","MMHG","0.1","840"
```

corresponding millisecond precision timestamp.

To calculate samples per hour, given a combined total of 400 samples per second for the three waveforms

$$400\text{samples} \times 60s \times 60m = 1.4 \text{ million samples per hour}$$

With each sample being a 16 bit integer, and the timestamp being a 64 bit integer, we could roughly approximate.

$$\frac{48 \text{ bits}}{8} = \frac{6\text{bytes}}{\text{sample}}$$
$$6 \times 1.4 \exp 6 = 8.4\text{MB}$$
$$12\text{hours} \times \frac{8.4\text{MB}}{\text{hours}} = 96\text{MB}$$

Even excluding the additional data stored per sample such as unit of measure, bed label, and time of first valid sample, it can already be seen that with a database size on disk of 67MB, there is already space saving.

As another example, the disk space could be compared to plan text CSV format. This test was performed by exporting a 15 minute period of the same 3 waveforms to a CSV file.

This created a 26MB csv text file, the first few lines are shown in Algorithm 5.1. At 26 MB for 15 minutes of data, this is equivalent to 104 MB per hour, or 1.2 GB for an equivalent 12 hour period. That is 1.2 GB for CSV format compared to 64 MB in influxDB, a factor of 20 improvement.

A comparison could also be made to simple dat format files. These structures typically rely on a known start time, with an assumed consistent interval between each sample, meaning the minimum data required can be roughly approximated as number of samples × bits per sample.

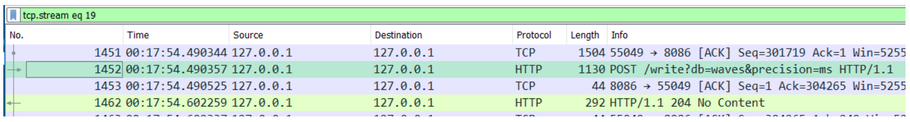
$$1.4M \times 16 \text{ bits} = 2.8\text{MB per hour}$$
$$2.8 \times 12 = 33 \text{ MB}$$

To compare this theoretical calculation to a real world example, the BIDMC Congestive Heart Failure Database from the PhysioNet project (Goldberger et al. 2000) offers a set of 20 hour, ECG recordings containing 2×250 Hz ECG waveforms with 12 bits per sample. Each of these 20 hours files is approximately 50 MB in size, or equivalent to 30MB for 12 hours.

This 30MB appears comparable to our influxDB sample, but lacks the per sample timestamp and label support, as well as flexible indexing and query language provided by the database. This is a significant benefit to favour the use of influxDB in this application.

5.2 http write time to influxDB

By measuring the write for the winaccess2influx tool writing to influxDB over http, it can be seen that most of the delay is in the API calls and processing, not in the http writes to database. A larger write receiving an acknowledgement just 110ms after the transmission was initiated. Shown in figure 5.2. This is with all tools running on the same server, hosting the database remotely would increase this delay.



```
tcp.stream eq 19
No.          Time            Source      Destination  Protocol  Length  Info
     1451 00:17:54.490344 127.0.0.1   127.0.0.1    TCP       1504 55049 → 8086 [ACK] Seq=301719 Ack=1 Win=5255
     1452 00:17:54.490357 127.0.0.1   127.0.0.1    HTTP      1130 POST /write?db=waves&precision=ms HTTP/1.1
     1453 00:17:54.490525 127.0.0.1   127.0.0.1    TCP         44 8086 → 55049 [ACK] Seq=1 Ack=304265 Win=5255
     1462 00:17:54.602259 127.0.0.1   127.0.0.1    HTTP       292 HTTP/1.1 204 No Content
```

Figure 5.2:

Chapter 6

Conclusion

By Leveraging modern C++ frameworks and the influxDB time series database, a useful data collection tool was developed.

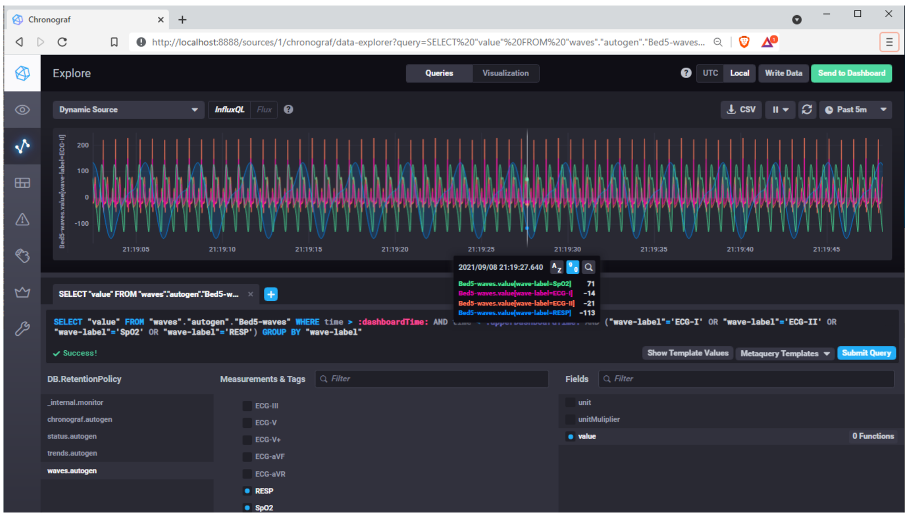
In comparison with raw CSV format, the database provides greatly reduced data size on disk. Additionally it provides the benfit of the flexible query language and web based visualisation tools. The less featured dat file formats were found to use similar disk space, but lacked the featureset of influxDB.

This project has not directly compared influxDB against other databases for the same use case. If the a single dataset were to be translated into multiple database formats, a useful direct comparison could be made, similar to the study of New York City taxi data performed in (Syeda Noor Zehra Naqvi 2018). Alternately, if the winaccess2influx tool were extended to support multiple database types, the write performance could also be compared.

Such study would be recommended prior to selection of influxDB for a larger scale research project.

For the scale tested in this project, influxDB proved appopriate, reliable and easy to use.

Looking to the future, it could be observed that this project only interfaces a single vendor API, which provides only data for the Dräger Patient monitoring platform and other devices directly connected to that platform. For a solution to provide widespread support for other vendors, it is desirable for some standardisation to occur. The HL7 protocol has had some success in providing cross vendor compatibility in the EMR space, but as of today the same can't be said of interoperability in the care area. The recently ratified IEEE 11073 SDC extends the 11073 standards family into the care area, with support for high resolution waveform data, NTP time synchronisation and modern web standard inspired security and compatibility. Were such a standard to be succesful, research tools such as winaccess2influx or ICM+ could readily be extended to support new devices and vendors without extensive drivers being built from scratch.



2021/09/08 21:19:27.640

```
Bed5-waves.value[wave-label=SpO2]     71
Bed5-waves.value[wave-label=ECG-I]   -14
Bed5-waves.value[wave-label=ECG-II]  -21
Bed5-waves.value[wave-label=RESP]   -113
```

```
SELECT "value" FROM "waves"."autogen"."Bed5-waves" WHERE time > :dashboardTime: AND time < :upperDashboardTime: AND ("wave-label"='ECG-I' OR "wave-label"='ECG-II' OR
"wave-label"='SpO2' OR "wave-label"='RESP') GROUP BY "wave-label"
```

✓ Success!

DB.RetentionPolicy	Measurements & Tags	Fields
_internal.monitor	ECG-III	unit
chronograf.autogen	ECG-V	unitMuliplier
status.autogen	ECG-V+	● value — 0 Functions
trends.autogen	ECG-aVF	
waves.autogen	ECG-aVR	
	● RESP	
	● SpO2	

Figure 6.1:

Bibliography

ADINSTRUMENTS (2021). *LabChart Lightning.* URL: https://www.adinstruments.com/products/labchart/lightning.

adinstruments (2021). *labchart lightning SDK.* URL: https://github.com/adinstruments/LightningDeviceSDK.

Blessing, Benn (2021). *winaccess2influx git repository.* URL: https://gitlab.com/minterop/winaccess2influx.

boost.org (2021). *boost c++ libraries.* URL: https://www.boost.org/.

Cambridge, University of (2021). *ICM+.* URL: https://icmplus.neurosurg.cam.ac.uk/.

DraegerWerk (2018). *Instructions for Use Infinity Gateway Suite. Protocol Handbook VF9.n.* DraegerWerk Ag. URL: https://www.draeger.com/Products/Content/infinity-gateway-suite-protocol-handbook-sw-vf9n-ifu-3703522-en.pdf.

github, offa on (2021). *influxdb-cxx.* URL: https://github.com/offa/influxdb-cxx.

Goldberger, Ary L. et al. (2000). “PhysioBank, PhysioToolkit, and PhysioNet”. In: *Circulation* 101.23, e215–e220. DOI: 10.1161/01.CIR.101.23.e215. eprint: https://www.ahajournals.org/doi/pdf/10.1161/01.CIR.101.23.e215. URL: https://www.ahajournals.org/doi/abs/10.1161/01.CIR.101.23.e215.

Rockstroh, Max et al. (May 2017). “OR.NET: Multi-perspective qualitative evaluation of an integrated operating room based on IEEE 11073 SDC”. In: *Journal for Computer Assisted Radiology and Surgery* 12. DOI: 10.1007/s11548-017-1589-2.

Syeda Noor Zehra Naqvi, Sofia Yfantidou (2018). “Time Series Databases and InfluxDB”. thesis. Brussels, Belgium: Universite libre de Bruxelles. URL: https://cs.ulb.ac.be/public/_media/teaching/influxdb_2017.pdf.

Appendices

.1 Early Progress September 2020

```cpp
// WVAPI2Influx.cpp : This file contains the 'main' function. Program execution begins and ends there.
//
#define W_CLIENT

#include <iostream>
#include <string>
#include <windows.h>
#include "WvAPI.h"

using namespace std;

auto MajorRev = WVAPI_MAJOR_REV;
auto MinorRev = WVAPI_MINOR_REV;
int NumberOfBeds;
wchar_t pServerName[] = L"localhost";
wchar_t pUserName[] = L"gwuser";
wchar_t pPassword[] = L"Welcome1!";
wchar_t pDomain[] = L"";
WV_BED_LIST_W BedList;

void printBedList();

int main()
{
        std::cout << "Hello World!\n";

        int ReturnCode = WvStart(&MajorRev, &MinorRev);
        cout << ReturnCode;
             if (ReturnCode != WV_SUCCESS) {
                         if (ReturnCode == WV_VERSION_MISMATCH) {
                                 printf("Version mismatch:  we're using version %d.%d, DLL is using version %d.%d",
                                         WVAPI_MAJOR_REV, WVAPI_MINOR_REV, MajorRev, MinorRev);
                                 WvStop();
                                 exit(EXIT_FAILURE);
                         }
             }

        ReturnCode = WvListBeds(pServerName, pUserName, pPassword, &BedList, &NumberOfBeds, pDomain, WV_SDC_PREFERRED);
        cout << "\n        number of Beds " << NumberOfBeds;
        printBedList();

        Sleep(4000);
        WvStop();
}

void printBedList() {
        for (int i = 0; i < NumberOfBeds; i++) {
                cout << "\n";
                for (int j = 0; j < WV_PATIENT_NAME_SIZE; j++) {
                        wcout << BedList.WvBeds[i].PatientName[j];
                }
        }

}
```

.2 example sequence per Vendor supplied header file

```
TYPICAL USAGE SEQUENCE:

The sequence for using these functions should be as follows:

   1. WvStart

   2. WvListBeds  (may include therapy devices as well as bedside monitors)

   3. Repeat for all the beds you are interested in:

      3.1 WvConnect

      3.2 At this point you could do a variety of things:

         To get alarm data:

            WvGetHighestGradeAlarm
                        WvGetHighestGradeVentAlarm

         To get parameter data:

            WvListParameters

            Repeat for all the parameters you are interested in:

               WvDescribeParameter
               WvGetParameterValue

         To get waveform data (current display only):

            WvListWaveforms
                        WvSetFilter     (use the list returned in the previous call)

            Repeat for all the waveforms you are interested in:

               WvDescribeWaveform
               WvGetWaveformSamples

                     To get all available waveforms:

                        WvListAvailableWaveforms
                        WvSetFilter     (use the list returned in the previous call)

            Repeat for all the waveforms you are interested in:

               WvDescribeWaveform
               WvGetWaveformSamples

                     To get a subset of waveform data:

                        WvListAvailableWaveforms
                        WvSetFilter     (use a subset of the list returned in the previous call)

            Repeat for all the waveforms you are interested in:

               WvDescribeWaveform
```

```
                  WvGetWaveformSamples


            To get vital signs data:

               WvGetVitalSignsReport

                        To get trend data (for a subset of all available trend data):
                                WVGetAvailableTrends(mConnectID, &Signals)
                                WvGetTrendData(mConnectID,pData,numHours, &Signals);

                        To get trend data (for all available trends):
                                WvGetTrendData(mConnectID,pData);

      4. WvStop
```

.3 Main.cpp from project

```
#pragma once
// WVAPI2Influx.cpp
//
// Building in VS 2019, using vcpkg for dependencies
// from vcpkg package influxdb-cxx, lineprotocol.cxx has been modified to use milliseconds
// C++ 17
//
#define W_CLIENT
#define WVDLLVERSION    // Set unicode for Draeger Winacces API
#include "winaccess2influx.h"
#include "WvApiObj.h"
#include "WvBed.h"

std::atomic_bool running;   // flag to end primary while loop

int main(int argc, char** argv) {
    windowsAppInit();       // handle ctrl-c etc, ensure only single instance of this program
    BOOST_LOG_TRIVIAL(info) << "Start\n\n Starting winacccess2Influx tool\n\n Not validated for diagnostic
    ↪ purposes\n benn.blessing@pm.me\n ctrl-c to exit" ;
    static WvApi* pWvApi = new WvApi(argv);             // reads in config and initialises WvAPI and list for bed
    ↪ objects
    setBoostLogLevel();

    while (running) {
        auto loopStartTime = std::chrono::steady_clock::now();
        auto nextLoopStart = loopStartTime + std::chrono::seconds(5);
        int NumberOfBeds{};
        pWvApi->deleteExpiredBeds();

        int ReturnCode = WvListBeds_W(pWvApi->gwHost.c_str(), pWvApi->gwUser.c_str(), pWvApi->gwPass.c_str(),
        ↪ &pWvApi->BedList, &NumberOfBeds, pWvApi->pDomain.c_str(), WV_INFINITY_ONLY);
        if (ReturnCode != 0) {
            if (ReturnCode == WV_NOT_INITIALIZED) {
                running = FALSE;    // shutdown program if WvApi has stopped
                BOOST_LOG_TRIVIAL(fatal) << "\nWvAPI Not initialised, terminating\n";
                break;
            }
```

```cpp
            BOOST_LOG_TRIVIAL(error) << "WvListBeds failed, ReturnCode " << pWvApi->MapIntRetCodes.at(ReturnCode);
            std::this_thread::sleep_for(std::chrono::seconds(2));
            continue;
        }
        if (NumberOfBeds == 0) {
            BOOST_LOG_TRIVIAL(trace) << "WvListBeds API call to " << utf8_encode(pWvApi->gwHost) << " successful, but 0 beds returned";
        }

        // check each current bedMap entry for presence in bedList, mark false if no match
        for (auto bed : pWvApi->bedMap) {
            BOOST_LOG_TRIVIAL(trace) << "bedmap entry " << bed.first;
            bool match = FALSE;
            for (int i = 0; i < NumberOfBeds; i++) {
                if (bed.first == uniqueBedID(pWvApi->BedList.WvBeds[i])) {
                    match = TRUE;
                }
            }
            // If none of the bedlist entries match sn of an existing bedmap entry, mark for removal
            // Will be removed at start of next loop. The flag will also instruct bed object bedLoop thread to terminate
            if (match == FALSE) {
                bed.second->InBedList_Atomic = FALSE;       // mark for removal from bedmap
            }
        }

        // For each bed returned in bedlist. check if already in bed map
        // If already in bed map, call update function to provide bed object with latest bedList info
        // This will provide bed objec with changes to MRN and device Status, eg bed coming out of standby
        for (int i = 0; i < NumberOfBeds; i++) {
            std::string bedID = uniqueBedID(pWvApi->BedList.WvBeds[i]);
            auto bedIter = pWvApi->bedMap.find(bedID);
            if (bedIter != pWvApi->bedMap.end()) {
                bedIter->second->update(pWvApi->BedList.WvBeds[i]);
            }
            // If BedList entry is not found in the bed object map, inset into map as new bed object
            else if (bedIter == pWvApi->bedMap.end()) {
                auto ret = pWvApi->bedMap.try_emplace(bedID, new WvBed(pWvApi->BedList.WvBeds[i]));
                if (ret.second == TRUE) { BOOST_LOG_TRIVIAL(info) << "added " << bedID << " to bedlist";  }
                else { BOOST_LOG_TRIVIAL(error) << "failed to add " << bedID << " to bedlist"; }
            }
        }
        // Check every 500ms to make terminate more responsive
        while (std::chrono::steady_clock::now() < nextLoopStart && running) {
            std::this_thread::sleep_for(std::chrono::milliseconds(500));
        }
    }
    // clean up and call WvStop()
    delete pWvApi;
    return 0;
}
```

.4 WvBed.cpp Bed object implementation from project

```cpp
#pragma once
#include "WvBed.h"

void WvBed::processCfg() {
    try {
        includeMRN = bedCfg.get_child("includeMRN").get_value<bool>();
        alarmQueryInterval = std::chrono::seconds(bedCfg.get_child("timing.alarmInterval").get_value<int>());
        trendQueryInterval = std::chrono::seconds(bedCfg.get_child("timing.trendInterval").get_value<int>());
        waveQueryInterval = std::chrono::seconds(bedCfg.get_child("timing.waveInterval").get_value<int>());
        captureWaves = bedCfg.get_child("parameters.captureWaves").get_value<bool>();
        captureTrends = bedCfg.get_child("parameters.captureTrends").get_value<bool>();
        cfgTrendParamString = bedCfg.get_child("parameters.trends").get_value("");
        cfgWaveParamString = bedCfg.get_child("parameters.waves").get_value("");
        cfgTargetBeds = bedCfg.get_child("source.targetBeds").get_value("");
        cfgTargetCU = bedCfg.get_child("source.targetCU").get_value("");
        trendURL = bedCfg.get_child("influxdb.trendURL").get_value("");
        waveURL = bedCfg.get_child("influxdb.waveURL").get_value("");
        statusURL = bedCfg.get_child("influxdb.statusURL").get_value("");
        paramTextWanted = bedCfg.get_child("parameters.create_txt_of_avail_param").get_value<bool>();
        addParamOnAlarm = bedCfg.get_child("parameters.addParamToBedIfAlarming").get_value<bool>();

        boost::to_upper(cfgTrendParamString);
        boost::to_upper(cfgWaveParamString);
        boost::to_upper(cfgTargetBeds);
        boost::to_upper(cfgTargetCU);
        boost::replace_all(cfgTrendParamString, " ", "");
        boost::replace_all(cfgWaveParamString, " ", "");
        boost::replace_all(cfgTargetBeds, " ", "");
        boost::replace_all(cfgTargetCU, " ", "");
        boost::replace_all(trendURL, " ", "");
        boost::replace_all(waveURL, " ", "");
        boost::replace_all(statusURL, " ", "");
    }
    catch (std::exception& e) {
        BOOST_LOG_TRIVIAL(fatal) << "Failed to process bed config " << e.what() << "\n";
        running = false;
        return;
    }
    // apply minimum 1 second intervals
    if (alarmQueryInterval < std::chrono::seconds(1)) { alarmQueryInterval = std::chrono::seconds(1); }
    if (trendQueryInterval < alarmQueryInterval) { trendQueryInterval = alarmQueryInterval; }
    if (waveQueryInterval < alarmQueryInterval) { waveQueryInterval = alarmQueryInterval; }
}

void WvBed::tokenizeParam() {
    allTrends = FALSE;
    allWaves = FALSE;
    boost::char_separator<char> sep(",");
    boost::tokenizer<boost::char_separator<char>> trendTok(cfgTrendParamString, sep);
    for (boost::tokenizer<boost::char_separator<char>>::iterator beg = trendTok.begin(); beg != trendTok.end();
    ↪  ++beg) {
        std::string trendLabel = *beg;
        auto it = WvApi::MapTrend.find(trendLabel); // Match string against enum, store parameter ID to config
        ↪  struct
        if (it != WvApi::MapTrend.end()) {
            trendParamIDs.push_back(it->second);
        }
        else if (trendLabel == "ALL") {
            allTrends = TRUE;   // don't push "ALL" to cfg.trendParamIDs
        }
        else {
            BOOST_LOG_TRIVIAL(warning) << "invalid trend parameter in config !! " << *beg;
        }
    }
    boost::tokenizer<boost::char_separator<char>> waveTok(cfgWaveParamString, sep);
    for (boost::tokenizer<boost::char_separator<char>>::iterator beg = waveTok.begin(); beg != waveTok.end();
    ↪   ++beg) {
        std::string waveLabel = *beg;
        auto it = WvApi::MapWvf.find(waveLabel);    // Match string against enum, store parameter ID to config
        ↪   struct
        if (it != WvApi::MapWvf.end()) {
            waveParamIDs.push_back(it->second);
        }
        else if (waveLabel == "ALL") {
            allWaves = TRUE;
        }
        else {
            BOOST_LOG_TRIVIAL(warning) << "invalid Wave parameter in config !! " << *beg;
        }
    }
}

void WvBed::checkIfBedSelected() {
    bool careUnitMatch = FALSE;
    bool bedMatch = FALSE;
    std::string careUnit = utf8_encode(bedDesc.CareUnit);
    std::string bedLabel = utf8_encode(bedDesc.BedLabel);
    boost::to_upper(careUnit);
    boost::to_upper(bedLabel);
    boost::erase_all(careUnit, " ");
    boost::erase_all(bedLabel, " ");

    boost::char_separator<char> sep(",");       // split string of beds from config file, check against this bed
    ↪   label
    boost::tokenizer< boost::char_separator<char> > bedTok(cfgTargetBeds, sep);
    for (boost::tokenizer< boost::char_separator<char> >::iterator beg = bedTok.begin(); beg != bedTok.end();
    ↪   ++beg)
    {
        std::string tempBedLabel = *beg;
        if (tempBedLabel == bedLabel || tempBedLabel == "ALL") {
            bedMatch = TRUE;
            break;
        }
    }
    boost::tokenizer< boost::char_separator<char> > cuTok(cfgTargetCU, sep);
    for (boost::tokenizer< boost::char_separator<char> >::iterator beg = cuTok.begin(); beg != cuTok.end(); ++beg)
    {
        std::string tempCuLabel = *beg;
        if (tempCuLabel == careUnit || tempCuLabel == "ALL") {
            careUnitMatch = TRUE;
            break;
        }
    }

    if (bedMatch && careUnitMatch) {
        ignoreBed = FALSE;
    }
    else {
        ignoreBed = TRUE;
        BOOST_LOG_TRIVIAL(info) << bedLabel << " online but not selected in config so ignoring";
    }
}

// Connect if monitoring and not already connected
void WvBed::connectBed() {
    if (bedDesc.DeviceStatus == WV_OPERATING_MODE_MONITORING && bedDesc.ConnectID == 0) {
        WV_REQUEST_TYPE connectMode;
        if (captureWaves) { connectMode = WV_REQUEST_ALARMS_PARAMS_AND_WAVES; }
        else { connectMode = WV_REQUEST_ALARMS_AND_PARAMS; }

        int returnCode = WvConnectEx(bedDesc.dirEntryId, &bedDesc.ConnectID, WV_INFINITY_ONLY, connectMode);
        if (returnCode == WV_SUCCESS) {
            connectTime = boost::posix_time::second_clock::local_time();
            //BOOST_LOG_TRIVIAL(info) << "connectBed succeeded " << bedLabel << " connectID " <<
            ↪  bedDesc.ConnectID;
            throw (returnCode);  // Throw to delay first calls to bed
        }
        else {
            BOOST_LOG_TRIVIAL(error) << "connectBed failed for " << bedLabel;
            throw (returnCode);
        }
    }
}

// Build wave filter as subset of avaiable waves per config
void WvBed::buildWaveFilter() {
    numFilteredWaves = 0;
    for (int i = 0; i < numWavesAvailable; i++) {
        if (allWaves || count(waveParamIDs.begin(), waveParamIDs.end(), wavesAvailable.WvWaveforms[i])) {
            waveListFiltered.WvWaveforms[numFilteredWaves] = wavesAvailable.WvWaveforms[i];
            numFilteredWaves += 1;
        }
    }
}

void WvBed::applyWaveFilter() {
    int returnCode = WvSetFilter(bedDesc.ConnectID, &waveListFiltered, &numFilteredWaves);
    if (returnCode != 0) {
        BOOST_LOG_TRIVIAL(error) << "WvSetFilter failed on bed " << bedLabel;
        throw (returnCode);
    }
}

// send waveform samples for all waveforms in waveSamples vector to influxDB
void WvBed::sendWaveSamples() {
    unsigned long int localSampleCount = 0;
    try {
        auto influxdb = influxdb::InfluxDBFactory::Get(waveURL);
        influxdb::Point::floatsPrecision = 2;
        influxdb->batchOf(5000);    // optimal influxdb batch size
        std::string bedLabWaves = influxBedLabel + "-waves";
        for (auto wave : waveSamples) {
            localSampleCount += wave.numSamplesReturned;
            for (int i = 0; i < wave.numSamplesReturned; i++) {
                if (firstValidSample.is_not_a_date_time()) {
                    firstValidSample = boost::posix_time::second_clock::local_time();
                    firstValidTimeStr = boost::posix_time::to_simple_string(firstValidSample);
                    boost::replace_all(firstValidTimeStr, " ", "_");
                }

                influxdb->write(influxdb::Point{ bedLabWaves }
                    .addTag("patID", patID)
                    .addTag("wave-label", wave.Label)
                    .addField("first-Valid-Data", firstValidTimeStr)
                    .addField("unit", wave.UOM)
                    .addField("unitMuliplier", wave.unitMultiplier)
                    .addField("value", wave.samples[i])
                    .setTimestamp(wave.firstSampleUTC + i * wave.sampleInterval)
                );
            }
        }
        influxdb->flushBatch();
        waveSamplesSent = localSampleCount;
    }

    catch (influxdb::InfluxDBException& e) {
        BOOST_LOG_TRIVIAL(error) << "Influx error in sendWaveSamples " << bedLabel << " " << e.what();
    }
    waveSamples.clear();    //Reset the vector of wave samples
}

// Query waveform samples for all waveforms in waveListFiltered
void WvBed::getWaveSamples() {
    if (bedDesc.DeviceStatus == WV_OPERATING_MODE_MONITORING && captureWaves && bedDesc.ConnectID) {
        for (int i = 0; i < numFilteredWaves; i++) {
            waveSampleData              waveData;
            WV_WAVEFORM_DESCRIPTION_W   waveDescriptionFromApi;
            int64_t                     tickTimeFirstSampFromApi{};
            int32_t                     FirstSampleTimestamp_fromAPI;
            unsigned int                FirstSampleSequenceNumber;
            int                         msecDiff;

            // scalemax divided by sample max
            int returnCode = WvDescribeWaveform_W(bedDesc.ConnectID, waveListFiltered.WvWaveforms[i],
            ↪  &waveDescriptionFromApi);
            if (returnCode != 0) {
                throw (returnCode);
            }
            try {
                waveData.Label = utf8_encode(waveDescriptionFromApi.Label);
                boost::replace_all(waveData.Label, " ", "-");           // Remove spaces for writing to influxDB
                waveData.UOM = WvApi::MapUOM.find(waveDescriptionFromApi.Units)->second;
                waveData.sampleRate = std::stoi(waveDescriptionFromApi.SampleRate);
                waveData.sampleInterval = std::chrono::milliseconds(1000 / waveData.sampleRate);
                waveData.WvWaveformID = waveDescriptionFromApi.WvWaveformID;
                if (waveData.WvWaveformID < WV_WAVE_ECG_LEAD_V6) {
                    waveData.unitMultiplier = 5;    // All ECG waves, 5 uV per unit from API
                }
                else {
                    waveData.unitMultiplier = std::stof(waveDescriptionFromApi.ScaleMax) /
                    ↪  std::stof(waveDescriptionFromApi.SampleMax);
```

```cpp
                }
            }
            catch (std::exception& e) {
                BOOST_LOG_TRIVIAL(error) << "unexpected error processing wave description" << e.what();
                throw;
            }

            returnCode = WvGetWaveformSamples(bedDesc.ConnectID, waveListFiltered.WvWaveforms[i],
            ↪  waveData.samples, WV_MAX_SAMPLES,
                &waveData.numSamplesReturned, &FirstSampleTimestamp_fromAPI, &FirstSampleSequenceNumber,
                &msecDiff, &tickTimeFirstSampFromApi);
            if (returnCode != WV_SUCCESS) {
                throw (returnCode);
            }
            auto serverTimeFirstSample = std::chrono::system_clock::now() - (waveData.numSamplesReturned *
            ↪  waveData.sampleInterval);
            waveData.firstSampleUTC =
            ↪  std::chrono::system_clock::time_point(std::chrono::milliseconds(tickTimeFirstSampFromApi) +
            ↪  tickDiffUTC);
            if (waveData.numSamplesReturned > 200) {
                serverBedTdiff = std::chrono::duration_cast<std::chrono::milliseconds>(serverTimeFirstSample -
                ↪  waveData.firstSampleUTC);
            }

            if (serverBedTdiff > std::chrono::seconds(120)) {
                BOOST_LOG_TRIVIAL(warning) << "Difference between bed clock and server time " <<
                ↪  serverBedTdiff.count() << "ms " << bedLabel;
            }
            waveSamples.push_back(waveData);
        }
    }
}

// Query list of all available waveforms for this bed
void WvBed::updateAvailableWaves() {
    int returnCode = WvListAvailableWaveforms(bedDesc.ConnectID, &wavesAvailable, &numWavesAvailable);
    if (returnCode != 0) {
        numWavesAvailable = 0;
        throw (returnCode);
    }
}

// Queries current alarm status and uses that data to calculate offset between bed tickCount and bed Localtime
// This offset will persist until disconnection of bed to prevent waveform distortion if bedside realtime clock
↪  shifts.
// ticktime is a continuous millisecond counter in the bedside monitor
void WvBed::updateTickOffset() {
    if (bedDesc.ConnectID != 0 && tickDiffUTC.count() == 0) {
        WV_ALARM_INFO_W     AlarmInfo;
        int returnCode = WvGetHighestGradeAlarm_W(bedDesc.ConnectID, &AlarmInfo);
        if ( returnCode == WV_SUCCESS) {
            long long timeFromAlarm = AlarmInfo.AlarmTimeStamp;
            tickDiffUTC = std::chrono::milliseconds(timeFromAlarm * 1000 - AlarmInfo.AlarmTickTimeStamp);
        }
    }
}

// Function to filter SPO2 wave if all zeros, or other invalid wave samples value '-32768' from API
```

```
void WvBed::filterWaveSamples() {
    for (auto& wave : waveSamples) {
        // Remove SPO2 wave if all zeros, eg no probe attached
        bool noValidSamples = TRUE;
        // If all samples are either -32768 or 0, set numSamplesReturned to 0
        for (int i = 0; i < wave.numSamplesReturned; i++) {
            if (wave.samples[i] == -32768) {
                wave.samples[i] = 0;
            }
            if (wave.samples[i] != 0) {
                noValidSamples = FALSE;
            }
        }
        if (noValidSamples) {
            wave.numSamplesReturned = 0;    // cheaper than erasing from vector
            BOOST_LOG_TRIVIAL(debug) << "No valid samples in " << wave.Label << " " << bedLabel;
        }
    }
}

void WvBed::grabHighestAlarm(alarmMode mode) {
    WV_ALARM_INFO_W     AlarmInfo;
    if (mode == monitor) {
        int returnCode = WvGetHighestGradeAlarm_W(bedDesc.ConnectID, &AlarmInfo);
        if (returnCode != WV_SUCCESS) {
            BOOST_LOG_TRIVIAL(trace) << "Failed to get Alarm info from " << bedLabel;
            throw (returnCode);
        }
    }
    else if ( mode == vent ){
        int returnCode = WvGetHighestGradeVentAlarm_W(bedDesc.ConnectID, &AlarmInfo);
        if ( returnCode != WV_SUCCESS  ) {
           // BOOST_LOG_TRIVIAL(trace) << "No vent Alarm data for " << bedLabel <<
             //   " returnCode " << WvApi::MapIntRetCodes.find(returnCode)->second;
            return; // Don't throw on Vent data failure as maybe no vent connected
        }
    }

    if (AlarmInfo.AlarmState != WV_ALARM_STATE_NOT_ACTIVE) {
        try {
            auto influxdb = influxdb::InfluxDBFactory::Get(trendURL);
            influxdb::Point::floatsPrecision = 2;
            std::chrono::system_clock::time_point timeStamp{};
            timeStamp += std::chrono::milliseconds(AlarmInfo.AlarmTickTimeStamp) + tickDiffUTC;
            std::string alarmState = WvApi::MapAlarmState.find(AlarmInfo.AlarmState)->second;
            std::string alarmMessage = utf8_encode(AlarmInfo.AlarmMessage);
            std::string alarmGrade = WvApi::MapAlarmGrade.find(AlarmInfo.AlarmGrade)->second;
            std::string alarmParam = "none";
            if (AlarmInfo.WvParameterID != WV_PARAM_INVALID) {
                int returnCode = WvDescribeParameter_W(bedDesc.ConnectID, AlarmInfo.WvParameterID,
                ↪  &trend.paramDesc);
                if (returnCode == WV_SUCCESS) {
                    alarmParam = utf8_encode(trend.paramDesc.Label);
                }
                // Check if alarming parameter is configured for trend recording, if not add it
                if (addParamOnAlarm && count(trendParamIDs.begin(), trendParamIDs.end(), AlarmInfo.WvParameterID)
                ↪  == 0) {
                    trendParamIDs.push_back(AlarmInfo.WvParameterID);
```

```cpp
                }
            }

            if (firstValidSample.is_not_a_date_time()) {
                firstValidSample = boost::posix_time::second_clock::local_time();
                firstValidTimeStr = boost::posix_time::to_simple_string(firstValidSample);
                boost::replace_all(firstValidTimeStr, " ", "_");
            }
            if (mode == monitor) {
                influxdb->write(influxdb::Point{ influxBedLabel + "-Alarms" }
                    .addTag("First-Valid", firstValidTimeStr)
                    .addTag("patID", patID)
                    .addField("Alarm-Grade", alarmGrade)
                    .addField("Alarm-Message", alarmMessage)
                    .addField("Alarm-Parameter", alarmParam)
                    .addField("Alarm-State", alarmState)
                    .setTimestamp(timeStamp));
                influxdb->flushBatch();
            }
            if (mode == vent) {
                influxdb->write(influxdb::Point{ influxBedLabel + "-Alarms" }
                    .addTag("patID", patID)
                    .addTag("First-Valid", firstValidTimeStr)
                    .addField("VentAlarm-Grade", alarmGrade)
                    .addField("VentAlarm-Message", alarmMessage)
                    .addField("VentAlarm-Parameter", alarmParam)
                    .addField("VentAlarm-State", alarmState)
                    .setTimestamp(timeStamp));
                influxdb->flushBatch();
            }
        }
        catch (influxdb::InfluxDBException& E) {
            BOOST_LOG_TRIVIAL(error) << " Influx error in AlarmGrabber " << E.what();
        }
    }
}

void WvBed::makeParamTextFile() {
    if ( paramTextWanted && numWavesAvailable ) {
        paramTextWanted = FALSE;    // only run once for life of ConnectID
        auto path = getenv("USERPROFILE") + std::string("\\.winaccess2influx\\paramRef");
        if (!std::filesystem::exists(path)) {
            std::filesystem::create_directories(path);
        }
        std::string input;
        input += "First Valid data since connection at " + firstValidTimeStr + "\n\n";
        input += "Valid loglevel values = fatal, error, warning, info, debug, trace\n\n";
        input += "Available Waveform parameters for " + bedLabel + "\n";
        for (int i = 0; i < numWavesAvailable; i++) {
            for (auto it : WvApi::MapWvf) {
                if (it.second == wavesAvailable.WvWaveforms[i]) {
                    input += (it.first + std::string(", "));
                    break;
                }
            }
        }

        input += "\n\nAvailalable trend parameters for bed " + bedLabel + "\n";
        for (int i = 0; i < trend.numParamReturned; i++) {
            for (auto it : WvApi::MapTrend) {
                if (it.second == trend.unfilteredParamList.WvParameters[i]) {
                    input += (it.first + std::string(", "));
                    break;
                }
            }
        }

        input += "\n\nComplete list of valid Waveform parameters\n";
        for (auto waveParam : WvApi::MapWvf) {
            input += waveParam.first + std::string(",\n");
        }

        input += "\n\nComplete list of valid trend parameters\n";
        for (auto trendParam : WvApi::MapTrend){
            input += trendParam.first + std::string(",\n");
            }

        auto filename = path + std::string("\\avail-Param-") + bedLabel + std::string(".txt");
        try {
            std::ofstream out(filename);
            out << input;
            out.close();
        }
        catch (std::exception& e) {
            BOOST_LOG_TRIVIAL(warning) << "Error created available trends txt for bed " << bedLabel << e.what();
        }
    }
}

void WvBed::bedLoop() {
    while (InBedList_Atomic && !ignoreBed && running ) {
        auto loopStart = std::chrono::steady_clock::now();
        auto nextLoopStart = loopStart + alarmQueryInterval;
        if ( bedDesc.ConnectID == 0) { nextLoopStart += std::chrono::seconds(3); }    // Soft start on first
        ↪  connect
        if (bedDesc.DeviceStatus == WV_OPERATING_MODE_MONITORING) {
            /// START MUTEX //////////////
            bedInfoMutex.lock();    // Prevent main calling update until loop completes
            try {
                connectBed();                   // throws on fail and first connect
                updateTickOffset();             // Must run for timestamp calculation, throw on fail
                getDemographics();              // checks for demographics enabled status in config
                grabHighestAlarm(monitor);      //
                grabHighestAlarm(vent);         //

                ////////////////// WAVES ////////////////////////////////
                if (std::chrono::steady_clock::now() > nextWaveQuery) {
                    nextWaveQuery = std::chrono::steady_clock::now() + waveQueryInterval;
                    if (bedDesc.ConnectID != 0 && captureWaves) {
                        updateAvailableWaves();
                        buildWaveFilter();
                        applyWaveFilter();
                        getWaveSamples();
                        filterWaveSamples();    // delete blocks of all invalid or SPO2 all 0
                        sendWaveSamples();
                    }
```

```cpp
                }
                /////////////////// TRENDS ///////////////////////////
                if (std::chrono::steady_clock::now() > nextTrendQuery) {
                    nextTrendQuery = std::chrono::steady_clock::now() + trendQueryInterval;
                    if (bedDesc.ConnectID != 0 && captureTrends) {
                        trendGrabber();
                        makeParamTextFile();    // run once for life of ConnectID
                    }

                }
            }
            catch (int returnCode) {
                // ConnectBed throws on first connection with WV_SUCCESS if succesful
                if (returnCode == WV_SUCCESS) {
                    BOOST_LOG_TRIVIAL(trace) << "connected " << bedLabel << " connectID " << bedDesc.ConnectID <<
                    ↪  " " << "returnCode WV_SUCCESS";
                }
                else if (returnCode == WV_PATIENT_DISCHARGED) {
                    disconnectBed();
                    // Delay next loop start beyond next WvListBeds in main loop
                    nextLoopStart += std::chrono::seconds(4);
                }
                else {
                    BOOST_LOG_TRIVIAL(debug) << "Data capture error " << bedLabel << " returnCode " <<
                    ↪  WvApi::MapIntRetCodes.at(returnCode);
                    nextLoopStart += std::chrono::seconds(4);   // delay next loop start on error to wait for
                    ↪  another bedlist update
                }

            }
            /////////////////////////  END MUTEX   ///////////////
            bedInfoMutex.unlock();
        }

        auto busyTime = std::chrono::steady_clock::now() - loopStart;
        BOOST_LOG_TRIVIAL(trace) << bedLabel << " busy time " <<
        ↪  std::chrono::duration_cast<std::chrono::milliseconds>(busyTime).count() << " ms";
        std::this_thread::sleep_until(nextLoopStart);
    }
}

void WvBed::disconnectBed() {
    BOOST_LOG_TRIVIAL(trace) << "Disconnecting discharged " << bedLabel;
    int returnCode = WvDisconnect(bedDesc.ConnectID);
    BOOST_LOG_TRIVIAL(trace) << "WvDisconnect returnCode " << WvApi::MapIntRetCodes.find(returnCode)->second;
    if (returnCode == WV_SUCCESS) {
        bedDesc.ConnectID = 0; // don't wait for next bedlist
        bedDesc.DeviceStatus = WV_OPERATING_MODE_DISCHARGE; // Set now to prevent reconnect attempt before bedlist
        ↪  updates
    }
    firstValidSample = boost::posix_time::ptime();  // Reset first sample timestamps
    firstValidTimeStr = "";
    serverBedTdiff = std::chrono::milliseconds(0);
}

// This function called by main to copy latest global bedList data into each bed object
// If the bed is busy might be locked, but busy means working so don't care if this is skipped.
void WvBed::update(WV_BED_DESCRIPTION_W updatedBedInfo) {
    if (bedInfoMutex.try_lock()) {
        bedDesc = updatedBedInfo;
        bedLabel = utf8_encode(bedDesc.BedLabel);
        if (bedDesc.DeviceStatus == WV_OPERATING_MODE_DISCHARGE && bedDesc.ConnectID) {
            disconnectBed();
        }
        // Ensure first sample timestamp is reset for disconnected bed
        if (!firstValidSample.is_not_a_date_time() && bedDesc.ConnectID == 0) {
            firstValidSample = boost::posix_time::ptime();
            firstValidTimeStr = "";
        }
        influxBedLabel = bedLabel + +"_sn_" + utf8_encode(bedDesc.SerialNumber);
        std::string bedStatus = WvApi::MapOpMode.find(bedDesc.DeviceStatus)->second;
        if (bedDesc.Wireless) {
            influxBedLabel += "-wireless";
        }
        std::string bedLabStatus = influxBedLabel + "-status";
        std::string devType = utf8_encode(bedDesc.DeviceType);
        std::string MRN = patID;
        long long currentWaveSampCount = waveSamplesSent;
        bool bedIgnored = ignoreBed;
        int conID = bedDesc.ConnectID;
        int tdiffms = serverBedTdiff.count();

        // Safe to unlock now before influx write if we only access local variables
        bedInfoMutex.unlock();

        try {
            auto influxdb = influxdb::InfluxDBFactory::Get(statusURL);
            influxdb->write(influxdb::Point{ bedLabStatus }
                .addTag("MRN", MRN)
                .addField("Bed-Ignored", bedIgnored)
                .addField("bed-Status", bedStatus)
                .addField("ConnectID", conID)
                .addField("device-type", devType)
                .addField("ms_diff_bed_to_server", tdiffms)
                .addField("Wave-Samples-Sent", currentWaveSampCount)
            );
            influxdb->flushBatch();
        }
        catch (influxdb::InfluxDBException& E) {
            BOOST_LOG_TRIVIAL(error) << " Influx error in update status " << bedLabel << E.what();
        }
        waveSamplesSent = 0;    // reset count
    }
    else {
        BOOST_LOG_TRIVIAL(trace) << "bed info update blocked by Mutex " << bedLabel;
    }
}

WvBed::WvBed(WV_BED_DESCRIPTION_W bedDescIn) {
    InBedList_Atomic = TRUE;
    bedCfg = WvApi::cfgTree;
    bedDesc = bedDescIn;        // Later bedDesc updates called from main via update function
    bedLabel = utf8_encode(bedDesc.BedLabel);
    processCfg();
    tokenizeParam();
    checkIfBedSelected();   // sets ignore bed flag if bed not selected in config

    // If bed is selected per care unit and bed, launch thread in this object to gather and export data
    if (!ignoreBed) {
        update(bedDescIn);
        bed_Thread = std::thread(&WvBed::bedLoop, this);
    }
}

WvBed::~WvBed() {
    InBedList_Atomic = false;
    int returnCode = WvDisconnect(bedDesc.ConnectID);
    if (bed_Thread.joinable()) {
        bed_Thread.join();
    }
}

void WvBed::getDemographics() {
    if (includeMRN) {
        patID = sanitiseForInflux(utf8_encode(bedDesc.PatientID));
    }
    if (includeMRN == FALSE || patID.empty()) {
        patID = "no-MRN";
    }
}

void WvBed::trendGrabber() {
    try{
        auto influxdb = influxdb::InfluxDBFactory::Get(trendURL);
        influxdb->batchOf(WV_MAX_PARAMETERS_PER_BED);   //must flush before destructor called
        influxdb::Point::floatsPrecision = 2;

        int returnCode = WvListParameters(bedDesc.ConnectID, &trend.unfilteredParamList, &trend.numParamReturned);
        if (returnCode != 0) {
            BOOST_LOG_TRIVIAL(debug) << "WvListParameters Failed for " << bedLabel;
            throw (returnCode);
        }
        for (int i = 0; i < trend.numParamReturned; i++) {
            if (allTrends || count(trendParamIDs.begin(), trendParamIDs.end(),
            ↪  trend.unfilteredParamList.WvParameters[i])) {
                returnCode = WvDescribeParameter_W(bedDesc.ConnectID, trend.unfilteredParamList.WvParameters[i],
                ↪  &trend.paramDesc);
                if (returnCode != WV_SUCCESS) {
                    BOOST_LOG_TRIVIAL(error) << "WvDescribe parameter Failed, bed " << bedLabel;
                    throw (returnCode);
                }
                std::string paramLabel = sanitiseForInflux(utf8_encode(trend.paramDesc.Label));

                if (trend.paramDesc.isSetting) {
                    paramLabel += "-setting";
                }
                std::string UOM = WvApi::MapUOM.find(trend.paramDesc.Units)->second;
                float value{};
                try {
                    value = std::stof(trend.paramDesc.Value);
                }
                // Expect exception on null value, break loop on NULL
                catch (const std::invalid_argument& e) {
                    BOOST_LOG_TRIVIAL(debug) << e.what() << " maybe null param value " << bedLabel << " " <<
                    ↪  paramLabel;
                    continue;
                }
                catch (const std::out_of_range& e) {
                    BOOST_LOG_TRIVIAL(debug) << e.what() << " maybe null param value" << bedLabel << " " <<
                    ↪  paramLabel;
                    continue;
                }
                if (firstValidSample.is_not_a_date_time()) {
                    firstValidSample = boost::posix_time::second_clock::local_time();
                    firstValidTimeStr = boost::posix_time::to_simple_string(firstValidSample);
                    boost::replace_all(firstValidTimeStr, " ", "_");
                }

                std::chrono::system_clock::time_point timeStamp{};
                timeStamp += (std::chrono::milliseconds(trend.paramDesc.ValueTickTimeStamp) + tickDiffUTC);

                influxdb->write(influxdb::Point{ influxBedLabel + "-trends" }
                    .addTag("First-Valid-Data", firstValidTimeStr)
                    .addTag("paramLabel", paramLabel)
                    .addTag("patID", patID)
                    .addField("unit", UOM)
                    .addField("value", value)
                    .addField("WvParamID", trend.paramDesc.WvParameterID)
                    .setTimestamp(timeStamp)
                );
            }
        }
        influxdb->flushBatch();
    }
    catch (influxdb::InfluxDBException& E) {
        BOOST_LOG_TRIVIAL(error) << " Influx error in trendGrabber " << E.what();
    }
}

// Remove spaces, '=', commas for use with influxDB
std::string WvBed::sanitiseForInflux(std::string inString) {
    boost::replace_all(inString, " ", "-");
    boost::replace_all(inString, ",", "");
    boost::replace_all(inString, "=", "");
    return inString;
}
```

.5 WvApiObj.cpp general implemenation functions from project

```
#pragma once
#include "WvApiObj.h"
#include "WvBed.h"

boost::property_tree::ptree WvApi::cfgTree{};

void WvApi::wvapi2influxinit(int MajorRev, int MinorRev) {
    int returnCode = WvStart(&MajorRev, &MinorRev);
    BOOST_LOG_TRIVIAL(info) << "Winacces API WvStart return Code " << WvApi::MapIntRetCodes.at(returnCode);
    if (returnCode != WV_SUCCESS) {
        BOOST_LOG_TRIVIAL(fatal) << "Failed to initialise Winaccess API, terminating";
        if (returnCode == WV_VERSION_MISMATCH) {
            BOOST_LOG_TRIVIAL(fatal) << "we're using API version " << WVAPI_MAJOR_REV << "." << WVAPI_MINOR_REV
                                     << "API DLL returned " << MajorRev << "." << MinorRev;
        }
        returnCode = WvStop();
        BOOST_LOG_TRIVIAL(info) << "WvStop return Code " << WvApi::MapIntRetCodes.at(returnCode);
        running = FALSE;
    }
}

void WvApi::deleteExpiredBeds() {
    for (auto bed = bedMap.cbegin(); bed != bedMap.cend();/*no increment*/)
    {
        if (bed->second->InBedList_Atomic == FALSE)
        {
            BOOST_LOG_TRIVIAL(info) << "dropping inactive bed " << bed->second->bedLabel << " From bedMap";
            bed = bedMap.erase(bed);
        }
        else
        {
            ++bed;
        }
    }
}

void WvApi::readConfig() {
    namespace fs = std::filesystem;
    auto dirPath = getenv("USERPROFILE") + std::string("\\.winaccess2influx");
    auto cfgFilePath = dirPath + std::string("\\winaccess.cfg");
    try {
        if (!fs::exists(dirPath)) {
            fs::create_directory(dirPath);
        }
        if (!fs::exists(cfgFilePath)) {
            BOOST_LOG_TRIVIAL(info) << "config file not found at " << cfgFilePath << " creating config from
            ↪  defaults";
            boost::property_tree::write_ini(cfgFilePath, cfgTree);
        }
    }
    catch (std::exception& e) {
        BOOST_LOG_TRIVIAL(fatal) << "Failed to write cfg file " << e.what();
        running = FALSE;
        return;
    }
    if (fs::exists(cfgFilePath)) {
        try {
            BOOST_LOG_TRIVIAL(info) << "Reading config " << cfgFilePath;
            boost::property_tree::read_ini(cfgFilePath, cfgTree);
        }
        catch (std::exception& e) {
            BOOST_LOG_TRIVIAL(fatal) << "Error reading in config file from " << cfgFilePath << e.what();
            running = FALSE;
            return;
        }
    }

    try {
        std::string host = cfgTree.get_child("Gateway.gwHost").get_value("");
        std::string user = cfgTree.get_child("Gateway.gwUser").get_value("");
        std::string pass = cfgTree.get_child("Gateway.gwPass").get_value("");

        gwHost = std::wstring(host.begin(), host.end());
        gwUser = std::wstring(user.begin(), user.end());
        gwPass = std::wstring(pass.begin(), pass.end());
    }
    catch (std::exception& e) {
        BOOST_LOG_TRIVIAL(fatal) << "Failed to process Gateway host details " << e.what();
        running = FALSE;
        return;
    }
}

void WvApi::buildDefaultIni() {
    std::string defaultTrendParam = " ECG_HR, SPO2_SAT, SPO2_PR, ART_D, ART_S, ART_M, MBUSX_RESP_ETCO2,
    ↪  MBUSX_RESP_FIO2, ";
    defaultTrendParam += "MBUSX_ETO2, MBUSX_ETN20, MBUSX_ETSEV, MBUSX_ETDES, MIB_BIS, MIB_SQI, MBUSX_RESP_VT,
    ↪  MBUSX_RESP_PIP, ";
    defaultTrendParam += "MBUSX_RESP_PEEP, MBUSX_RESP_MV, MIB_BIS, MIB_SQI, NIBP_S, NIBP_D, NIBP_M, TEMP_BASIC_A,
    ↪  TEMP_BASIC_B";
    cfgTree.put<std::string>("comment", "Config for winaccess2influx tool. This file will be recreated if
    ↪  deleted");
    cfgTree.put<std::string>("influxdb.waveURL", "http://localhost:8086/?db=waves&precision=ms");
    cfgTree.put<std::string>("influxdb.trendURL", "http://localhost:8086/?db=trends&precision=ms");
    cfgTree.put<std::string>("influxdb.statusURL", "http://localhost:8086/?db=status&precision=ms");
    cfgTree.put<std::string>("Gateway.gwHost", "localhost");
    cfgTree.put<std::string>("Gateway.gwUser", "gwuser");
    cfgTree.put<std::string>("Gateway.gwPass", "Welcome1!");
    cfgTree.put<std::string>("Gateway.pDomain", "localhost");
    cfgTree.put<std::string>("parameters.trends", defaultTrendParam);
    cfgTree.put<std::string>("parameters.waves", "ECG_LEAD_II, SPO2, ART, CVP");
    cfgTree.put<int>("timing.trendInterval", 5);
    cfgTree.put<int>("timing.alarmInterval", 1);
    cfgTree.put<int>("timing.waveInterval", 5);
    cfgTree.put<std::string>("source.targetBeds", "ALL");
    cfgTree.put<std::string>("source.targetCU", "ALL");
    cfgTree.put<bool>("includeMRN", TRUE);
    cfgTree.put<bool>("parameters.captureWaves", TRUE);
    cfgTree.put<bool>("parameters.captureTrends", TRUE);
    cfgTree.put<bool>("parameters.addParamToBedIfAlarming", TRUE);
    cfgTree.put<bool>("parameters.create_txt_of_avail_param", TRUE);
    cfgTree.put<std::string>("logLevel", "info");
}

WvApi::WvApi(char** argv) {
    buildDefaultIni();
    readConfig();
    wvapi2influxinit(WVAPI_MAJOR_REV, WVAPI_MINOR_REV);
}
WvApi::~WvApi() {
    int returnCode = WvStop();
    BOOST_LOG_TRIVIAL(trace) << "WvStop returnCode = " << returnCode;
}
```