MEASURED DATA FORMAT FILE GENERATION FOR LONG-TERM DATA STORAGE

A Co-op Thesis written for

TESLA

and submitted to

KETTERING UNIVERSITY

in partial fulfillment
of the requirements for the
degree of

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

by

SAMUEL SALVATORE DALEO, III

2017

Student

Employer Thesis Advisor

Faculty Thesis Advisor

DISCLAIMER

This thesis is submitted as partial fulfillment of the graduation requirements of Kettering University needed to obtain a Bachelor of Science in Computer Engineering Degree.

The conclusions and opinions expressed in this thesis are those of the student author and do not necessarily represent the position of Kettering University or anyone else affiliated with this culminating undergraduate experience.

PREFACE

This thesis represents the capstone of my five years combined academic work at Kettering University and past work experiences. My Culminating Undergraduate Experience provided the opportunity for me to use the knowledge and skillset learned while at Kettering to manage a project of this magnitude.

Although this thesis represents the compilation of my own efforts, I would like to acknowledge and extend my sincere gratitude to the following persons for their valuable time and assistance, without whom the completion of this thesis would not have been possible:

1. Jared Petrie
2. Rustin Bergren
3. Jonathan McCormick
4. Elon Musk

TABLE OF CONTENTS

I. INTRODUCTION

A. The Nature and Purpose of the Project

The nature of this project was to assist in the advancement of the System Validation test automation capabilities by creating a tool that the team could use to automatically store data in a long-term storage format. Tesla's mission statement is to accelerate the world's transition to sustainable energy, and in order to accelerate this transition effectively, the company must release a product that has been thoroughly vetted to ensure high quality. In order to do this, engineers perform a careful and thorough analysis of the product by means of testing, gathering data during testing, and doing analysis on said data to determine whether or not the product is ready to be released to consumers. Unfortunately, the amount of time and manpower it takes to ensure a thoroughly vetted product is substantial. Moreover, when information needs to be exchanged throughout the organization, the company does not have a set standard and information can be lost. This information loss leads to communication issues and delays. The 'mdfwriter' repository at Tesla is a result of this thesis project, in an effort to create an easy to use Python package that engineers can use to efficiently store data as a company standard.

B. The Background & Literature Review to the Project

The time delay problem at Tesla arose in the company as a result of aggressive scheduling alongside demanding challenges with bleeding edge technology. In order to try to meet these short release deadlines, groups scramble to exchange necessary information to make professional decisions about engineering designs, validation and anything else that might arise. However, the groups often times use different methods for certain processes and therefore difficulty can arise in communication between the groups. Yet, as the company sustains itself, the people within these groups work together to create solutions that will work for everyone to ensure the interoperability within the company. The measured data file format (MDF) is a versatile solution to the problem of scattered data. This file format is currently already widely used by people within the organization. Many engineers and analysts use a software tool from Vector called CANape to view the data stored in .mdf files. However these files are currently only generated through expensive proprietary software tools. In order for one to have an MDF, one must already have access to the toolchain needed to generate one. This project aims to generate the file by means of reverse engineering the MDF specification in an open source Python software library. Open source meaning that underlying operations and architecture are available for any person to see and manipulate. The solution is to create a library in the Python programming language that test engineers may use within their automated software solutions that do not involve proprietary software or hardware setups.

C. The Methodology by Which the Project was Completed

When the problem was presented to create an .mdf file generator, I decided that what made the most sense to handle low level binary output was to use the C language. C has a very straightforward file I/O stream method that I could take advantage of. Unfortunately a lot of problems came from this design decision initially. The system validation teams use Python to write their test scripts, and having this package in a different language creates a lot of "translation" problems. In order to solve the translation problems, a tool called "Simplified Wrapper and Interface Generator" (SWIG) can be used to enable one to call functions regardless of which language they originate in. Yet, the translation problem arose in figuring out how to allow a user to input Python data types and output them in C to write to the .mdf file. So, after some difficulty and learning, a new way to write to files was discovered that provided the same functionality as the C file I/O, yet no "cross-language translation" was necessary. Python was the means by which this project was finished.

The largest sources of data for the project were from the company Vector. Vector was responsible for creating the .mdf file specification, and they released specification sheets that broke down the low level binary structure of the file. It was quite complicated, however most of the information was pieced together from this specification sheet and used to create the file generation library.

D. The Criteria by Which the Project Was Completed

The parameters for success were very clear. The package needed to generate a file that could be used with preexisting data analysis software at Tesla without bugs. If used correctly, the library should be able to generate a complete MDF file given the header and data contents of the file from user input. The file also needed to accurately store high resolution data points given as inputs. The library itself needed to generate the file with speed, as long automated tests that might store more data than the file size limit would create a new file, and if the file took too long to set up, data loss would occur.

E. An Overview of the *Substantive Written Material* in the Formal Document

The chapters in the remainder of this paper illustrate the major learning milestones over the six months the project took to complete and are listed as follows.

1. Binary structure of file
2. C Language, SWIG, Failures
3. Python
    1. "Low level" functionality in a "High Level" language
    2. Algorithms
    3. Theory of Operation
    4. Data storage

CONCLUSIONS & RECOMMENDATIONS

The mdfwriter tool ultimately ended up satisfying the logging needs of the System Validation team. The validity of the file that the library generates is proven with the fact that one may use the library to store any amount of defined data, and that same data is able to be read back in an MDF analysis tool such as Vector's CANape program. A single unit test was written for the library, but not one that looks into and validates the specific binary contents of the file. Unfortunately, pass/fail criteria for unit tests were defined but not implemented in the time that I had to complete this project. Currently, the package is implemented in the back end of Tesla Energy's automated system validation test machines to store test data. However, one bug was brought up at the end of my last internship was the fact that the MDF specification had an error in it. The block definition for the CG block contains the details for the record ID property. In the specification sheet, the value is defined as a 'UINT16' meaning that it is a value that is stored with two bytes. However, later in the specification sheet when it talks about writing the record ID at the beginning of the data record, only *one* bytes worth of space is given for the record ID. In this case, only 255 Channel Groups may have data stored for them. In the application that this package was being applied to, it was noted that this may cause an issue when storing data for more than 255 Channel Groups, perhaps uploaded from an internal .dej file. Yet, the library is still functional and it is possible to use in other applications if this bug does not present issues.

Focus

The project was started because of the need for the automated test stations to be able to automatically store data. Unfortunately, towards the end of the project a large problem was revealed and as of writing this thesis, not solved. There was an error in the MDF specification sheet that displayed an inaccuracy in the data size for the Channel Group's Record ID. The definition for the CG Block says that the Record ID is a UINT16, giving two bytes worth of space to store the value in the CGBlock. However, when writing record to the data block, there is only one byte allocated for the Record ID to be written at the beginning of a data record. Because of this, only the first 255 Channel Groups in the file can be stored into the file. Currently, there is not a visible way to solve this problem. The library and file format can still be used within the company, however because the System Validation's group needs more than 255 Channel Groups for the automated testing, it might not be possible to use mdfwriter as a solution for data storage.

Integration

Since the project was not so much an exploratory project, not many new findings were discovered. The biggest discoveries were within what it takes to create a software tool that will stand the test of time within the company. Software tools need to be very clean and user friendly in order for easy integration into already existing software projects or new ones. If a package is not well documented both with in-line comments and external documentation, it will be harder for other engineers to utilize the tools that are created.

Another thing that was discovered was that Python is an extremely versatile language. It can handle low level programming projects as well as high level projects given that there is a large community of people continually developing libraries for the language. Initially there was a lot of skepticism to use the language as I noticed that the file I/O was not what I needed it to be. However, after discovering that there was a nother library I could use to format the input for the file-stream, it was clear that Python has a lot of versatility to complete various projects.

Recommendations

One recommendation for tackling a large software project such as mdfwriter is to be sure to carefully analyze the information necessary to piece together everything to complete the project. The reason why is because had I discovered that there was a discrepancy to data size problem of the Record ID, an alternative file format might have been discovered and used instead of MDF. Maybe the format might habve been stuck with, but Tesla could have worked with Vector to change the specification for the file to fix this error on their part. Nevertheless, taking the time to view all necessary information to piece things together is crucial for project integrity.

Another recommendation is to carefully analyze what already exists in the software world. Open source software has come a long way in a short amount of time, and while a library might not exist that does exactly what you need, there is a good chance that a package might contain a way of doing something, or even an entire library

of definitions that might be useful when building a new project.

Conclusions Regarding Specific Thesis Categories

At the start of the project, there was no real question for the project. The problem was that there was no accessible format for software test engineers to store data. Many different methods were approached by various groups at the company, and now the MDF file generator exists. However, upon attempting to write the library in C given that there is a lot of low level data type work done here, a question arose as to whether or not Python could handle the task. It turns out, yes- it can. Python seems to very a very versatile and open language, where if you can think of a way to do something, chances are it will work. There was concern that there would be no way for a high level scripting language to meticulously handle the specific low level data types, but using the struct library makes the control of these binary values easy.

There is still a lot that can be done with this library. Currently the library writes an unsorted MDF file, and a lot of open source software prefers to consume a sorted file. That being said, a sort method could perhaps be written for this library so that upon completion, a sorted file is output instead of an unsorted file. This would require knowledge of how much space the file will actually need for all of the data contents, however is definitely doable. Moreover, there are additional features laid out in the MDF specification sheet that are not currently implemented. The specification sheet contains many other block definitions that one might find useful depending on the use case of the library. Unfortunately, it is not clear whether or not I will return to the company upon

graduation, and therefore it is unclear if additional features and proper maintenance of this library will continue here on. Nevertheless, the library will sit in Tesla's software library for years to come and will hopefully provide a means of storing data for engineers that will be used for years to come.

REFERENCES

Vector Informatik GmbH [https://vector.com/vi_mdf_en.html]

Simplified Wrapper and Interface Generator [http://www.swig.org/]

SUBSTANTIVE WRITTEN MATERIAL

A: File Structure

A Measured Data Format (.mdf) file is a file format that was developed by Vector in 1991 in collaboration with Bosch. The .mdf emerged as the standard file in the automotive industry for long-term data storage. As shown in Illustration 1, the MDF file structure is complex- it is comprised of many individual “blocks” that contain relevant information that is used for the file to be tied together. These blocks are analyzed by other Vector (and third-party) software to point to where specific data is stored into the file.

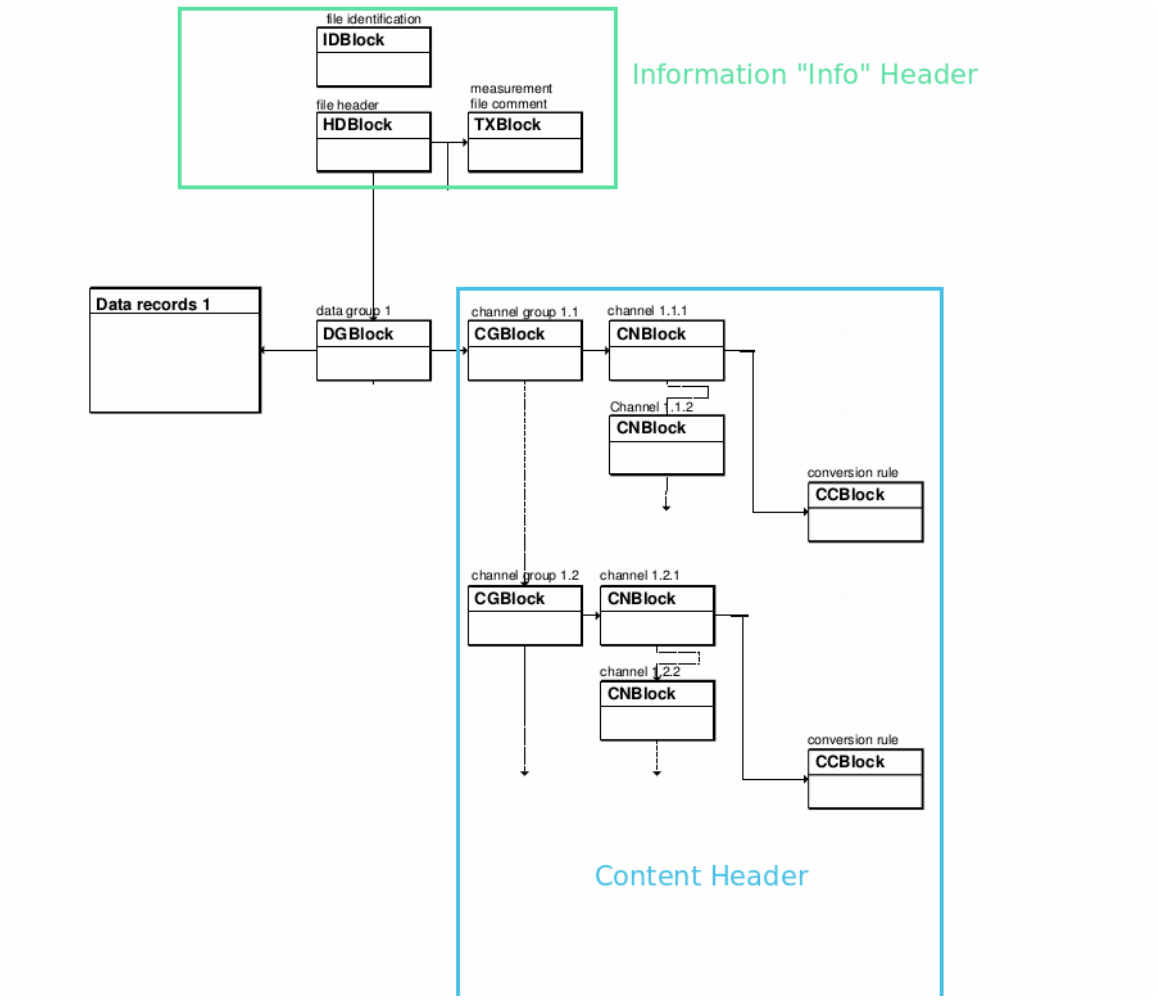

Figure A-1: MDF Block Diagram

The top two rows within Figure 1 show what is called the “information header” of the file. The ‘ID Block’, ‘HD Block’, ‘TX Block’ and the ‘DG Block’ are only written once at the beginning of the file. These blocks contain various pieces of identifying information that makes the file legitimate. After the header comes the remainder of the file overhead. ‘CG’ , ‘CN’, and ‘CC’ blocks collectively give a “Channel Group” it’s necessary identifying information. The name “Channel Group” is derived from the

specification sheet's definition of the "CG Block", as it is the highest level block that is a parent to the rest of the blocks. A "Channel Group" is simply a group in which all the data is collected at the same time. A "Channel Group" has at least one "Channel" in it corresponding to only one set of data. Each channel is a single set of data, and every channel within the channel group contains the same time points at which the data is collected. Once the blocks for the Channels and Channel Groups are created, the only thing that's left is the actual storage of the data. The theory behind the 'mdfwriter' software is that all of the information about the data is stored into the file's header upon initialization, and then the file stream is kept open for live data recording until complete.

As previously stated, the information that is contained in the "Information Header" is contained in the 'ID', 'HD', 'TX' and 'DG' blocks. It's important to note that the way the 'DG' block is written depends on whether the file will be written as a "sorted" or "unsorted" format. The 'mdfwriter' library writes the file as an unsorted file. It is possible to then later obtain a sorted file using Vector command line tools, however if the library writes a sorted format initially, the amount of specific space that the file will take will have to be known before hand. Since that is not easy to determine in most cases, it's very difficult to write a sorted file from the start. If the file is sorted, then there is one DG block for each CG block. Otherwise, in the unsorted file format that this library writes, there is only one DG block for all of the CG blocks. This makes it possible to live-stream data to the file, because the DG Block can contain all of the Channel Group blocks (CG Blocks) that retain organizational information about the stored data. Tables 1

through 4 below show the specific contents of the individual blocks that make up the Information Header.

Data Type	Numbe r	Description
CHAR	8	File identifier, always contains "MDF ". ("MDF" followed by five spaces)
CHAR	8	Format identifier, a textual representation of the format version for display, e.g. "3.00 "
CHAR	8	Program identifier, to identify the program which generated the MDF file
UINT16	1	Byte order
		0 = Little endian
UINT16	1	Floating-point format used
		0 = Floating-point format compliant with IEEE 754 standard
UINT16	1	Version number of the MDF , i.e. 300 for this version
UINT16	1	Reserved
CHAR	2	Reserved
CHAR	30	Reserved

Table A-1: ID Block data contents

Data Type	Numbe r	Description
CHAR	2	Block type identifier, always "HD"
UINT16	1	Block size of this block in bytes (entire HDBLOCK)
LINK	1	Pointer to the first file group block (DGBLOCK)
LINK	1	Pointer to the measurement file comment text (TXBLOCK) (NIL allowed)
LINK	1	Pointer to program block (PRBLOCK) (NIL allowed)
Uint16	1	Number of data groups
CHAR	10	Date at which the recording was started in "DD:MM:YYYY" format
CHAR	8	Time at which the recording was started in "HH:MM:SS" format
CHAR	32	Author's name
CHAR	32	Name of organization or department
CHAR	32	Project name
CHAR	32	Measurement object e. g. the vehicle identification

Table A-2: HD Block data contents

Data Type	Number	Description
CHAR	2	Block type identifier, always "TX"
UINT16	1	Block size of this block in bytes (entire TXBLOCK)
CHAR	variable	Text (new line indicated by CR and LF; end of text indicated by 0)

Table A-3: TX Block data contents

Data Type	Number	Description
CHAR	2	Block type identifier, always "DG"
UINT16	1	Block size of this block in bytes (entire DGBLOCK)
LINK	1	Pointer to next data group block (DGBLOCK) (NIL allowed)
LINK	1	Pointer to next channel group block (CGBLOCK) (NIL allowed)
LINK	1	Reserved
LINK	1	Pointer to the data records (see separate chapter on data storage)
UINT16	1	Number of channel groups
UINT16	1	Number of record IDs in the data block
		0 = data records without record ID 1 = record ID (UINT8) before each data record 2 = record ID (UINT8) before and after each data record
UINT32	1	Reserved

Table A-4: DG Block data contents

Once this preliminary data is written in the file's information header, the second part of the header called the "content header" needs to be written. The content header contains all of the blocks for each Channel Group within the file. The organizational structure of this half of the header will be will be discussed in a later chapter. The 'CG', 'CN', 'CC', and 'CE' blocks in Table 5 are what make up Channel Groups. The highest level block is a 'CG' (Channel Group) block. After that, a Channel Group will have at least one CN (Channel) block, and a CC (Channel Conversion) block for each CN block. Depending on the type of data that the Channel is storing, a CE (Channel Extension)

block will also be included in the Channel Group for that Channel. The only time a Channel will need a CE block is when it is storing raw hex-data CAN frames. Tables 6-10 below depict the information that is stored in each of these blocks. Note that the pointers in each block are very important to keep the integrity of the file intact. These pointers allow Vector (and third party) software to analyze an MDF file and interpret the Channel Groups accordingly to retrieve data.

Data Type	Numbe r	Description
CHAR	2	Block type identifier, always "CG"
UINT16	1	Block size of this block in bytes (entire CGBLOCK)
LINK	1	Pointer to next data Channel group block (CGBLOCK) (NIL allowed)
LINK	1	Pointer to first channel block (CNBLOCK) (NIL allowed)
LINK	1	Pointer to channel group comment text (TXBLOCK) (NIL allowed)
UINT16	1	Record ID
UINT16	1	Number of channels
UINT16	1	Data record size in bytes (without the record ID), i.e. data size of the channel group for each sample
UINT32	1	Number of records

Table A-5: CG Data block contents

The important pieces to note in the CG block are the three "LINK" type values. They are pointers to the next CGBlock (to keep all of the CGBlocks tied together in the file), The first CNBlock within this specific Channel Group, and an option TXBlock for adding text comments to the Channel Group. The text block was not implemented in to the currently existing version of the mdfwriter library due to time constraints and lack of use. Another important thing about the CG block is the UINT16 data-type value called "Record ID". This number is important because it is the only thing that separates all of the actual data records within the data block underneath the header. Every time a piece of data is stored into the file during live-streaming, the record is preceded with a number called the

“Record ID”. Once the file is complete and interpreted by additional software, the “Record ID” determines which Channel Group the data belongs to and therefore tells the software how to interpret the data based on the additional CN, CC, (and perhaps) CE blocks that the CG block points to.

In figure 7, the important values to note are the pointers to the “next CN block”, which will continually be used within a Channel Group to link all of the channels it contains. The pointer to the conversion formula is also important because it allows a piece of software to determine how to interpret the raw data that the channel group is storing. The rest of the pointers in the CN block are simply written as NIL values because they are not implemented in this version of the library.

Data Type	Number	Description
CHAR	2	Block type identifier, always "CN"
UINT16	1	Block size of this block in bytes (entire CNBLOCK)
LINK	1	Pointer to next channel block (CNBLOCK) of this channel group (NIL allowed)
LINK	1	Pointer to the conversion formula (CCBLOCK) of this signal (NIL allowed).
LINK	1	Reserved
LINK	1	Reserved
LINK	1	Pointer to the channel comment (TXBLOCK) of this signal (NIL allowed)
UINT16	1	Channel type
		0 = data channel 1 = time channel for all signals of this group (in each channel group, exactly one channel must be defined as time channel)
CHAR	32	Signal name, i.e. the first 32 characters of the ASAM-MCD unique name
CHAR	128	Signal description
UINT16	1	Number of the first bits [0..n] (bit position within a byte: bit 0 is the least significant bit, bit 7 is the most significant bit)
UINT16	1	Number of bits
UINT16	1	Signal data type
		0 = unsigned integer 1 = signed integer (two's complement) 2,3 = IEEE 754 floating-point format 7 = String (NULL terminated) 8 = Byte Array
BOOL	1	Value range – known implementation value
REAL	1	Value range – minimum implementation value
REAL	1	Value range – maximum implementation value
REAL	1	Rate in which the variable was sampled. Unit [s]
LINK	1	Pointer to the ASAM-MCD unique name (TXBLOCK) (NIL allowed)
LINK	1	Pointer to TXBLOCK that contains the signal's display identifier (default: NIL; NIL allowed)
UINT16	1	Byte offset of the signal in the data record in addition to bit offset (default value: 0) *note: this fields shall only be used if the CGBLOCK record size and the actual offset is larger than 8192 Bytes to ensure compatibility; it enables to write data blocks larger than 8kBytes*

Table A-6: CN Block data contents

Data Type	Number	Description
CHAR	2	Block type identifier, always "CC"
UINT16	1	Block size of this block in bytes (entire CCBLOCK)
BOOL	1	Value range – known physical value
REAL	1	Value range – minimum physical value
REAL	1	Value range – maximum physical value
CHAR	20	Physical unit
UINT16	1	Conversion formula identifier
		0 = parametric, linear 1 = tabular with interpolation 2 = tabular 6 = polynomial function 7 = exponential function 8 = logarithmic function 9 = ASAP2 Rational conversion formula 10 = ASAM-MCD2 Text formula 11 = ASAM-MCD2 Text Table, (COMPU_VTAB) 12 = ASAM-MCD2 Text Range Table (COMPU_VTAB_RANGE) 132 = Date (Based on 7 Byte Date data structure) 133 = time (Based on 6 Byte Time data structure) 65535 = 1:1 conversion formula (Int = Phys)
UINT16	1	Number of value pairs for conversion formulas 1, 2, 11 and 12 or number of parameters
...		Parameter (for type 0, 6,7,8, 9) or table (for type 1, 2, 11, or 12) or text (for type 10), depending on the conversion formula identifier. See formula-specific block supplement.

Table A-7: CC Block data contents

Data type	Count	Description
CHAR	2	**Block type identifier**, always "CE"
UINT16	1	**Block size** of this block in bytes (entire CEBLOCK)
UINT16	1	**Extension type** identifier 2 = DIM (DIM block supplement) 19 = Vector CAN (Vector CAN block supplement)
...	variable	**Additional fields** depending on the extension type. See extension-specific block supplements.

Note:
- for CAN signals, type Vector CAN (19) should be used
- for ECU signals (CCP, XCP), type DIM (2) should be used

Table A-8: CE Block data contents

Not much information in Table 8 and 9 are critical for the operation of the library.

The values for the type of conversion of the raw data is important, and if a CE block is

included it has basic information specifying that the information is “Vector CAN” data. Other than that, there are no pieces of information to worry about that keep the integrity of the file structure together.

The algorithms and operation of the library will be discussed in a later chapter, however it’s important to note the pointer values in the various tables above as they play an important role in keeping the structure of the file intact. Once all of the blocks are written to the file, the data block begins and the file stream is left open for the user to write data to the file as long as necessary. Once there is no more data that should be stored to the file, the file is closed.

B: C Language and SWIG

Once the scope of the project was realized, it was clear that a library needed to be created that would write specific binary contents (the contents of which are laid out in the previous chapter). Throughout the years of low level programming at Kettering University, it was obvious that the C language libraries contained exactly what was needed to write binary contents to the file-stream at a user's command. The first edition of the library contained definitions for all of the pieces of information contained in Chapter 1 for each block. After the header file was written, the definitions were used to define write methods that used an opened file-stream to write the specific binary contents to. Each write method took the various content from the struct and wrote the information to the file stream with no problem. However, once the header was written, a problem arose with the approach of using C.

The initial use case for this library was to take live data from automated tests and store it for later analysis. Most of these automated tests are written in a scripting language such as Python. Because Python creates data types differently than C does, the data types that might be input by a user could not easily be consumed by a method in C and written to a file. For example, a Python "int" data type is a C "long" type. Because of so many differences in binary sizes for input types, it is very difficult to accurately take python input data types and output them to a file stream that is opened in C to match the specified data lengths shown in Chapter 1.

Because of this critical difference, it was brought up by a co-worker that the

Python input data types can be translated to corresponding C types. Although, in order to do something like this, a new tool would need to utilized.

A co-worker suggested a tool called SWIG. SWIG stands for Simplified Wrapper and Interface Generator. This is a tool that allows a programmer to write code in one language and then use it with any language for which SWIG can generate. By using SWIG, it was possible to write the initial code in C and then translate it to Python. Once the initial write functions were written in C, it was possible to automatically generate python code using SWIG. However, the problem still remained that the python input data needed to be translated correctly. This is where SWIG's advanced techniques came into play. A feature called "typemapping" allows an engineer to define which Python data types should be converted into any C data type (or any other language supported by SWIG). A typemap is a piece of code that SWIG automatically injects into it's generated output code during compilation. A sample of a SWIG typemap might look something

```
PyObject *wrap_gcd(PyObject
*self, PyObject *args) {
   int arg1;
   int arg2;
   int result;
   PyObject *obj1;
   PyObject *obj2;
   PyObject *resultobj;

   if
(!PyArg_ParseTuple("OO:gcd",
&obj1, &obj2)) return NULL;

   /* "in" typemap, argument 1
*/
   {
      arg1 =
PyInt_AsLong(obj1);
   }

   /* "in" typemap, argument 2
*/
   {
      arg2 =
PyInt_AsLong(obj2);
   }

   result = gcd(arg1,arg2);

   /* "out" typemap, return
value */
   {
      resultobj =
PyInt_FromLong(result);
   }

   return resultobj;
}
```

like:

Figure B-1: Sample SWIG typemap code

It is unnecessary to go into explicit detail about the contents of the typemap code. However, it is important to understand the functionality of such code. The code takes a variable number of input parameters designated by a coder and converts each parameter into the desired output format that the typemap is set up for. The typemap is essentially a function that returns these newly "translated" input parameters and passes them into the automatically generated SWIG code.

After about a week of work time spent on understanding how to write the templates for SWIG's typemaps, there was still no functional product using the C library. As time continued to go forward, it seemed as though the project would become extremely complicated and unlikely to achieve a result that was free of bugs and usable within the software infrastructure within the company. So, it was determined that there might be another way to achieve the desired end result without as much complication.

C: Python

Part 1: “Low-level” functionality in a “High Level” language

There was a lot of skepticism on my behalf when it came to turning to Python as the language that this project would be written in. It was already determined that Python’s native file-stream I/O operations did not provide as concise of a result as what was offered in C, which is where most of the skepticism arose from. After digging through the internet, a piece of open source software called “mdfreader” from Github shined a light into what would be the path leading to success. The “mdfreader” library provided functionality to load a .mdf file into memory and manipulate the contents of the data within the file as dictionaries. Yet, there was another important function that provided insight to how our “mdfwriter” library could be created. The “mdfreader” library allowed a programmer to add additional contents to an already existing file. The way they wrote the new data to the binary structure was by using a Python package called “struct”. Struct allows a programmer to take any data type from Python and ‘pack’ it into a binary structure as specified. Therefore, it was possible to take any Python data type as input and format it in the same fashion that the C language did to write it to a filestream. After discovering this, it was clear that this could be done in Python entirely.

The write methods and block structures that were previously defined in the C library were transferred over to Python with a little bit of work, and it was onto the next task.

Part 2: Algorithms

Once the contents from the C library were rewritten in Python, the next piece of the puzzle needed to be solved. It was now possible to write all possible blocks to an open file stream using Python, but it was useless unless they could be tied together properly. Back in Chapter 1, during the discussion of the binary contents of the blocks in the file, it was noted that the "LINK" data types defined by Vector called Pointers would play a large role in the functionality of this file. In order to allow this library to generate a file given any number of Channel Groups that needed to be stored, the pointers must be calculated mathematically. It was determined that the file should group like blocks together to make the calculation of these dynamic pointer variables as simple as possible. The figure below shows an overview of the way the "content header" of the file is laid out.

CG1
CG2
…
CGn
CN1
CN2
…
CNn
CC1
CC2
…
CCn
CE1
CE2
…
CEn

Table C-1: Binary structure for content header

Once this structure was determined, it would be easy to simply calculate the position of the necessary block the pointer needed to reference based on the size of each block and how many blocks of this type existed the file already. The _write_header() method in the ‘mdfwriter’ library contains 5 sections of code that correspond to five pointers that are criticial to the operation of the file. The CG Block has a pointer to the first CN Block it contains, and a pointer to the next CG Block in the file. The CN Block contains three pointers. It has a pointer to the next CN Block in the CG it belongs to, a CC Block pointer for the data for that specific CN Block, and if necessary, a CE Block to indicate that the CN contains CAN data.

a. CG Block Pointer to Next CG Block

The code in this section is as follows:

```
# cgPointerPointer
for i in range(len(self.cgBlockList)):
    location = self.cgPointers['cg' + str(i + 1)]['cgPointerPointer']
    if len(self.cgBlockList) == 1:
        pointer = 0
    elif len(self.cgBlockList) > 1:
        if i + 1 is not len(self.cgBlockList):
            pointer = size_offset_of_information_header + self.TXBlock.blocksize + size_of_cgblock * (i + 1)
        elif i + 1 is len(self.cgBlockList):
            pointer = 0
    self.file.seek(location)
    self.file.write(STRUCT_TYPE['LINK'].pack(pointer))
```

b. CG Block pointer to first CN Block

```
# cnPointerPointer #
for j in range(len(self.cgBlockList)):
    location = self.cgPointers['cg' + str(j + 1)]['cnPointerPointer']
    if j > 0:
        if j == 1:
            cn_offset = self.cgPointers['cg' + str(j)]['cnCount']*size_of_cnblock
        else:
            cn_offset += size_of_cnblock * self.cgPointers['cg' + str(j)]['cnCount']
    pointer = size_offset_of_information_header + self.TXBlock.blocksize \
              + size_of_cgblock * len(self.cgBlockList) + cn_offset
    self.file.seek(location)
    self.file.write(STRUCT_TYPE['LINK'].pack(pointer))
```

c. CN Block pointer to next CN Block

```
# nextCNPointer
location = self.cnPointers['cn1']['cnPointerPointer']
for m in range(len(self.cgBlockList)):
    cn_number = self.cgPointers['cg' + str(m + 1)]['cnCount']
    for n in range(cn_number):
        if n + 1 is not cn_number:
            pointer = location + cn_block_offset_minus_pointer_location
        elif n + 1 is cn_number:
            pointer = 0
        self.file.seek(location)
        self.file.write(STRUCT_TYPE['LINK'].pack(pointer))
        location += size_of_cnblock
```

d. CN Block pointer to CC Block

```
# CNBlock Pointers
# ccPointer
for l in range(len(self.cc_blockList)):
    if l > 0:
        ccoffset = ccblock_size_list[l-1]
    else:
        ccoffset = 0
    location = self.cnPointers['cn' + str(l + 1)]['ccPointerPointer']
    pointer = size_offset_of_information_header + self.TXBlock.blocksize + \
              size_of_cgblock*len(self.cgBlockList) + size_of_cnblock*len(self.cnBlockList) + ccoffset
    self.file.seek(location)
    self.file.write(STRUCT_TYPE['LINK'].pack(pointer))
```

e. CN Block pointer to CE Block

```
s = 0
if len(self.ceBlockList) > 0:
    for t in range(len(self.cnBlockList)):
        location = self.cnPointers['cn' + str(t + 1)]['cePointerPointer']
        if self.cnBlockList[t].channelTitle == "CAN":
            pointer = size_offset_of_information_header + self.TXBlock.blocksize + \
                      size_of_cgblock*len(self.cgBlockList) + size_of_cnblock*len(self.cnBlockList) + \
                      ccblock_size + size_of_ceblock*s
            s += 1
        elif self.cnTypeList[t] == 1:
            pointer = 0
        self.file.seek(location)
        self.file.write(STRUCT_TYPE['LINK'].pack(pointer))
self.file.seek(0, 2)
```

These algorithms simply calculate the number of each type of block available and are executed in turn to write the pointers for each block in the correct position. The

locations are recorded during the execution of the "_write_header()" method. Both the "_write_header()" method and the "_write_pointers()" method work together to create the entire first half of the file. After this content is saved to disk, all that is left is to write data to the file, which will be discussed in the next section.

Part 3: Theory of Operation

At this point, the structure of the file header should be written. Yet, one large piece of information is still missing. How does the program know what contents to populate the header of the file with? In order for programmers to use this program, some additional features needed to be implemented. The previous methods for _write_header and _write_pointers are private methods located in the larger "MDF" class in the library. The MDF class can be instantiated to create an MDF object. Upon instantiation of this object, a few important things happen.

1. Various containers are created and associated with this new MDF object. These containers are holders for the blocks in which the file header will be populated with. There are also dictionaries created to record all of the pointers that the program will later reference while executing the _write_pointers() method previously discussed.
2. A filestream is opened. A programmer can determine the name of the file that the stream will open by defining it as one of the input parameters in the instantiation of the MDF object.

To ease a programmer's burden of creating all of the correct blocks for the MDF object to contain, the high level Channel Group object plays a major role. The name Channel Group was taken from the MDF specification sheet and turned into a high level object to be created. The relationship is that it is still representative of a low level "CG" Block, but abstracted enough to relieve a programmer of the burden to define every piece

that goes with it. Just like the low level CG Block, there are Channels that are associated with the Channel Group. The high level object Channel is abstracted away from the Vector definition of a CN block to contain the necessary information that a CN Block contains. Both the Channel Group and the Channel objects are consumed by the MDF object in order to populate the MDF object's containers with the necessary "block" objects. There is another object that is a child class of Channel Group, called "CANmsg". Similarly, there is a high level "CANsignal" object that is a child class of the "Channel" class. However, the CANmsg object is specifically for creating a Channel Group that will contain raw hexadecimal CAN data from a CAN bus.

To start using the 'mdfwriter' library, a user must create all necessary Channel Group/CANmsg objects with their associated Channels/CANsignals. The Channel Group object's container stores all relevant Channel objects. Once a user has all Channel Group objects created, an MDF object may then be created. There is a specific method in the MDF class called "add_channel_group()" which contains one input parameter "channelgroup" which is the high level Channel Group object that the programmer has previously created. The "add_channel_group()" method is the key to adding substance to the empty MDF object. This method consumes all high level Channel Group objects. Once all relevant Channel Group objects are added to the MDF object, the programmer may run the "start_file()" method which goes through the MDF object's containers and created the low level block objects to be written to the binary stream.

It must be noted that the MDF file was originally created to hold CAN data. During this abstraction process, the capability of storing *any* data to the file was created.

When a “Channel Group” object is consumed by the MDF, the necessary low level structures are automatically created to allow any data to be stored and later recalled. This is because of the way the CC Block is created when the MDF object consumes a Channel Group.

The CC Block indicates the type of conversion equation that a piece of analysis software should use to analyze the stored data. In the case of a regular Channel Group being consumed, a CC Block is automatically created to use a linear conversion equation to interpret stored data.

The equation is:

$$Physical\ Value = Int * P2 + P1 \qquad \textbf{(C-1)}$$

where “Physical Value” is the physical data value from anywhere data is collected. Int is the implementation value stored in the MDF file. By setting P2 = 1 and P1 = 0, the physical value and the Int value are the same number, and any number can be stored into the file’s data block.

When a CANmsg object is consumed by an MDF object, however, more information is needed to allow analysis software to interpret the hexadecimal data. The CCBlock exists but will not always use the linear conversion equation. If, for example, the CAN message that this CANmsg object is representing is enumerated, then the CC Block will point the software to a Key:Value table that will pair individual signal values with signal names. To ease the programmer from the burden of individually creating CANmsg and CANsignal objects for each desired CAN message to be stored, the MDF object contains a method called “import_DEJ” which parses a Tesla internal .dej file

containing all information about a specific set of CAN messages. This “import_DEJ” method creates all necessary high level CANsignal and CANmsg objects that will be consumed when the file is ready to be written.

Once the MDF object holds all necessary objects in it’s containers, the “start_file” method is ready to be called. The “start_file” method uses the filestream that was opened upon instantiation of the MDF object and executes the “_write_header” and “_write_pointers” methods. Once the file is started, all that is left is storing data to the file.

Part 4: Data Storage

Storing data to the file is what the file is all about. The MDF file exists as a long-term data storage solution within the automotive industry. The MDF object within the 'mdfwriter' library contains a method called "write", which takes three input parameters. The write method first refers to the name of the Channel Group that it wishes to store data for. The second input parameter is a floating point value called "time_offset". The time_offset is a numeric value of the number of seconds that have passed since the time located in the header. By default, the timestamp is set to whenever the MDF object is created. However, a user may use the MDF object's method "define_start_time" to set the timestamp to anything. The third input parameter to the "write" method is either one of two things. In the case that the Channel Group that's being accessed is a CANmsg, the third parameter should be no more than an integer corresponding to the hexadecimal value for the specified CAN message. Otherwise, the input parameter shall be a list of data containing one data point per channel in the Channel Group. For example, if a Channel Group has three Channels, then the list that is passed in the "write" method should contain "[1,2,3]" points of data corresponding to the order the channels are stored in the Channel Group.

This third parameter was difficult to define when designing the "write" method. It is very confusing to a third party to have an input parameter than could ideally be one of two different data types. Moreover, a user must also remember the order in which Channels are stored in a Channel Group, otherwise data can end up mis-represented. The more ideal format for this input parameter is a dictionary that contains the name of each

Channel in the data group and a corresponding data point as the value for said key.