Vulcan Digital Engineering

Rev: 2020-10-08

Jacob Lehmer

# Introduction

Data sharing is a necessity of collaborative research projects. Often an afterthought is the format of the data, usually accepting the default format of whatever system is used to record the data. The first place that problems occur is with required post processing of gathered data. The format output by data gatherers is often focused on streaming speed and data density. For the purposes of recording data, the data dense format from the recorder makes sense, however the data in this format requires extensive manipulation to be able to be used by post processing applications. The modification usually takes the form of a single use script or an application that can convert the data from the format understood by the recorder. Single use applications like these abound in the research space and managing all of them without forgetting which does what can often be a daunting task.

For these reasons, another application is being developed with the goal of preventing the entropy of all the different single purpose applications. This application utilizes a framework that enables different file formats to be read and understood through plugins dynamically loaded to the main application containing the framework. The main application provides interoperability to all the different plugins, this allows each file format to convert to every other file format that has been read. There is no theoretical limit to the number of plugins that can be used by the application. Even though the plugins may be individually written and serve a purpose like single use scripts, the plugins can only be used with the main application, this forces the application and plugins to be linked. In theory this will force better data management, or at least the bundling of the main application with the plugins.

## Adding a new file format

There is currently no reliable way to see the future. To truly make software future proof all possible usages must be understood at the design phase. For practical reasons this cannot be accomplished, the best that can be accomplished is anticipating eventual use cases and providing a method for modification of the system.

For the case of a data translation system the problem that arises in the future is a lack of support for data formats. This application allows the usage of a new file format to be integrated into the overarching system through usage of a dynamic link library read at runtime. Dynamic runtime integration allows for a functioning code base to remain unmodified but additional functionality can still be added. This is accomplished through the extensive usage of unified interfaces throughout the application.

## Principles of a Dynamic Link Library

A dynamic link library (DLL) is a specific type of compiled code. This code package, a library, is compiled into the architecture of the system. DLLs have two main use cases, the first is to provide a library that can be accessed during the code writing stage, allowing the written code to utilize the functionality in the DLL. This does require the application to be modified and recompiled if more functionality from the DLL needs to be incorporated. The other way a DLL can be used is to be loaded at some point when the program executes which will then use functionality in the DLL. The Vulcan application uses the second method, allowing the main framework of the application to remain unchanged as new file reading functionality is added.

## Architecture of Vulcan Application

The Vulcan Application handles interoperability between the different plugins and handles the multithreading, bulk file handling, and user interface. The application will allow for any of the plugin systems to utilize the resources exposed by other plugins. This is accomplished by transforming the data into a common format that can be read and written by the other file plugin systems. This file format, known as the data exchange format, is detailed in the next section.

This format acts as an intermediary and does not actually need to be directly written as a file. This file format has a one to one conversion between the actual physical file and the code base that supports it. The file format describes how the physical file can be read and written to. The API description shows how to interface with the rest of the data formats to and from the data exchange format.

There are three components to read any file format, there is the executable element which handles file reading and multithreading, there is the plugin which is written by the user and there is the library which defines the API that the plugin interfaces with. The main executable and the library are provided along with two plugins, and their source code, which should be a complete set, along with this documentation, to start building your own plugins.

# Data Exchange Format

A new data format has been developed for the purpose of data interchange. This format will serve as the tools default data format. The name for this format is the data exchange format.

## Tenants

The format has been developed to meet the following design tenants

* Easy to read
* Data Dense
* Extensible

## Components

In order to accommodate the requirement for an easy to read while still being data dense file the decision was made to separate the meta data information and the data information. The metadata is stored in one of currently two formats, .xml or .json. The extended identifier for these files is .xdef and .jdef in order to differentiate between the multitude of other .xml and .json files that are in current usage. These file extensions are not necessary but do serve as an immediate way to determine what the form of the data is. The binary file contains data. The structure section of this document goes into further detail as to how the two components are related.

## Structure

An agnostic description of tags is below. These tags are the same, with the same capitalization for both .json and .xml. The differences in reading and writing .xml and .json do not have any bearing on the functionality of the file and will not be covered here. The tags can occur in any order within the document but nesting needs to be maintained.

|  |  |  |  |
| --- | --- | --- | --- |
| Tag | Parent Tag | Acceptable values | Description |
| MetaData |  | List of DataDescription Elements | Defines main structure of file. Needs to be the first tag in file for file to be recognized as def file |
| DataDescription | MetaData | Required DataDescription sub elements | Defines how to read and where to find actual binary data file |
| DataType | DataDescription | “Int”,”Real”,”Char” “Unsigned Int” | Defines the type of data to be read |
| BytesPerElem | DataDescription | Integer value representing number of bytes | Defines the number of 8-bit bytes that represents an individual data element in the binary data |
| Endianness | DataDescription | “Big”,”Little” | Represents the endianness of the data elements |
| NumberOfElements | DataDescription | Integer value representing the number of elements in the data channel. | Number of values in the data channel.  If the file is streamed this represents the channel chunk size, if the file is not streamed the value represents the total number of values |
| DataFilePath | DataDescription | Relative or absolute file path to the actual binary data file | Defines where the binary file for the data processing is. The relative file path is in relation to the meta data file. |
| Interleaved | DataDescription | “Yes”,”No” | [NOT REQUIRED]  [DEFAULT: “No”]  Indicates if the data defined by this meta data channel is interleaved in the binary data |
| ChunkOffset | DataDescription | Integer representing the number of bytes into a chunk the data can be found | [REQUIRED IFF Interleaved is “Yes”]  Indicates the number of bytes into a new chunk that this data channel can be found |
| ChunkSize | DataDescription | Integer representing the number of bytes in a chunk | [REQUIRED IFF Interleaved is “Yes”]  Indicates the number of bytes in a chunk in total. |
| ID | MetaData | String | [NOT REQUIRED]  [DEFAULT: metadata filename + #of entry]  This will give the ID of the current data channel, displayed. |
| Anything else | MetaData | Any | Any additional tag can be added, these tags are viewable, and will be written out in the system but do not have any intrinsic value. |

Table : Meta data tags for .def files

The binary files are references from the “DataFilePath” tag in the meta data. One meta data file can contain many entries, each pointing to a different file. Or the meta data can have multiple entries pointing to a single file, in which case the file would be an interleaved file. Or any combination therein. It is required that the meta data accurately define the structure of the data. The binary file only contains a stream of binary data. The file extension of the binary file is completely irrelevant and can be omitted. The following diagram shows the relation of the different meta data elements in the binary file.

A screenshot of a cell phone

Description automatically generated

Figure : Binary Data Format

# Hardware Requirements

All computers use DLLs, in Unix derived systems these are called shared objects, to reduce the amount of repeated code without reducing the functionality of applications. Any computer can run the Vulcan Application software suite and be able to comprehend the files. The native language is C++, which is recommended to use, but any DLL compatible language can be used to integrate the Vulcan Application.

## Visual Studio

The examples in this document are compiled and ran with Microsoft Visual Studio. Any modern compiler should work to build these elements.

# Required Integration Library

To properly interface a plugin with the main application, the application and the plugin need to implement the same interface. This interface takes the form of a traditional, compile time, dynamic link library. This dynamic link library provides the data structure base classes that can be inherited as well as the classes that use them. The name of this library is called the CodexDistributable.dll.

# API Documentation

The following sections describe the CodexDistributable.dll. These sections form the description of the API, how to use this API and examples are later in this document.

The API is composed of the following five parts

* Codex Base
* Message Handling System
* Primary Data Storage
* Current Storage Element Grouping
* Primary Data Storage Element

Only the CodexBase and primary data storage element are required to be modified to build a new codex, but the other components need to be understood for the system to operate.

## CodexBase

The CodexBase Class is exposed by the CodexDistributable library. This is a purely abstract class that needs to be inherited to be able to successfully use the system. There are four elements that need to be implemented and one that is optional.



Figure . CodexBase Class

### Identifier

The identifier of the codex is a c string that defines the name of the codex. This string is stored in the protected identifier field. The getIdentifier() method is automatically called, unless there is a need to change how the identifier is retrieved, the method itself does not need to be changed.

### processFile

This method is responsible for running the logic associated with the new file format in question.



Figure . processFile Signature

The processFile method is one of the most important for adding functionality to the system. This method is called when a new file has been found and the current codex has been selected through the codex identifier to be used. This method is meant to read enough data from the file to be able to read the data at a future point. To phrase this another way, the processFile method will read the metadata of the file, however, not every file format has formal metadata. The processFile is required to process a minimal amount of data from the file so that all of the data can be read at a future point.

The absolute file path is passed to the process file method. The method is responsible for determining if the file passed to it is correct. This is usually done through checking if the file extension is correct. However, any method can be used; for instance, many binary files contain a “magic number” as the first integer. The MessageHandlingSystem, detailed later, can be used to send a message to indicate an error. The processFile is meant to read the new format and then create a currentStorageElementGrouping of PrimaryDataStorageElements. The PrimaryDataStorage, detailed later, is used to store the data elements that have been read in by the system.

Both the PrimaryDataStorage and the MessageHandlingSystem are instantiated and stored in the main Vulcan application. A pointer to both elements are passed between all of the different codexes. The provides a unified interface for all the different codexes.

### outputFile



Figure . outputFile Signature

The method outputFile will take a PrimaryDataStorageElement and output the required information in the required format to the required directory. This method can be written any way, but usually at the beginning of this method there is a call to a wrapper of convertDatatoDefFile in the the PrimaryDataStorageElement. This is because there codexes are not required to completely convert data to the DEF format upon being read. This is a requirement because many file formats require a large amount of hard drive interaction to properly convert these file formats, when the GUI developed there is a possibility that the file will never need to be output or converted, by only physically converting the data when known to be necessary the application is able to save resources by not spuriously converting data.

### convertDatatoDefFile



Figure . convertDataToDefFile Signature

This method will take whatever actual data is stored in the files and converts it to the data exchange format, this is usually takes the form of using temporary files to move the data to so that no data is overwritten. The reason this is necessary is that the processFile method is only required to read the metadata but when converting data formats the actual data has to be in the form of the Data Exchange Format. Creating a derived class from the primaryDataStorageElement will allow the elements for this method to be written to. This method needs to exist, as the CodexBase class has this as a purely virtual class, but if the data is read into the primaryDataStorageElement format, this is also the DataExchangeFormat, by default then the method does not need to do anything at all.

## MessageHandlingSystem

The MessageHandlingSystem is a structure for exposing a threadsafe method for writing any required messages. This class has few interfaces that allow for any developed plugin to be incorporated into the rest of the system without concern of race conditions that can render the message unintelligible.



Figure . Message Handler Class

There are four methods currently, October 2020, that can be used by the end user to utilize the message handling system. These are the two fatal methods, concatenateMessages and concatenateStrings.

### Fatal messages



Figure . Method Signatures For Fatal Messages

These methods allow for the passing of a message to be processed, either displayed or recorded, usually these messages are used to call out when the codex or a subsystem has failed and is erroring out. Other messages can be sent through this system though, further development will expand this to allow many kinds of messages when appropriate.



Figure . Concatenation Methods

The Concatenation methods allow for the concatenation and return of an arbitrary number of strings without any requirements other than the number of elements to be concatenated. The difference between the two methods is that concatenateMessages will put a space between all the strings passed to the method whereas the concatenateStrings will append the strings together.

## PrimaryDataStorage

The PrimaryDataStorage is the central hub of the entire Vulcan Digital Engineering application. This provides the main threadsafe storage interface for the application.



Figure . Primary Data Storage Structure

The PrimaryDataStorageElement has many different elements that do many different things but for the purposes of developing a new plugin there are only a couple that are required to be implemented. All public methods are detailed below, not all of them will be used, but the function of each will be detailed along with how they are used.

### beginReceivingData



Figure . Method Header for receiving data

This method is used to signal to the main system that a new file has been passed to a codex. This method is called automatically before the codex receives the file. This is a critical component to the threadsafe nature of the application. There is no need to interact with this method when writing a plugin, in fact calling it when inside of a codex will cause a deadlock.

### completeReceivingData



Figure .Method that completes the receiving of data

This method is used to signal to the main system that reading a new file has been completed. This method is called automatically when the system has completed reading a file. There is no need to interact with this method from inside of a codex. Doing so will cause the threadsafe precautions to be disabled and further uploading to the primary data store will cause errors.

### retrieveData



Figure . Method Signature for Data Retrieval

This method will take a pointer to a currentStorageElementGrouping pointer and assign data elements to it. The return value determines if this method succeeds. This method is called automatically and does not need to be called from the plugin being written for the system to function properly.

### addMoreData



Figure . Method Signature for Adding More Data

This method will take a pointer to a currentStorageElementGrouping and incorporate it into the complete set of data currently available. This method will need to be called by the codex. Once a currentStorageElementGrouping, detailed later, is created then it is uploaded through this method.

### exitAllWaiting



Figure . Method Signature to Exit all Waiting Threads

This method is used to gracefully stop all of the threading that is waiting on the PrimaryDataStorage. This should not ever need to be called from within a codex.

### getNextSystemID



Figure . Method signature for getting the next System ID

This method is used to retrieve the next incrementing ID number of the data that is being loaded into it. This method will need to be called when creating a new PrimaryDataStorageElement. Each primaryDataStorageElement Requires a unique ID and this is where it comes from.

## currentElementStorageGrouping

The current element storage grouping is an intermediary data structure that is meant to represent a single file, regardless of how the file is structured with data.



Figure . Current Storage Element Grouping

This struct is meant to be mostly self contained. All that is required is creating one and calling the addToCurrentStorageElementGrouping for each of the PrimaryDataStorageElements that have been found in the file.

### addToCurrentStorageElementGrouping



Figure . Method signature for adding data to grouping

This method receives a pointer to a PrimaryDataStorageElement and then adds it to its internal object. This method just needs the proper parameters and the rest is taken care of.

## PrimaryDataStorageElement

The primary data storage element is the other element that may be required to be derived to successfully create a codex for the new file format. The default version of this structure contains all the information required to output a data exchange format file. This is a vitally important structure to understand, but preliminary information is required to understand how to use the structure.



Figure . Preliminary Information for Data Storage Element

This preliminary information is required to properly understand the PrimaryDataStorageElement. The enumerations at the top of the document identify the data types and endianness of the data proper. The data description struct contains the same information as the data description in the data exchange format script. The earlier section describing the data exchange format should provide the necessary information to understand the data description.

The KeyValueProperty is a convenient method for storing all of the different kinds of metadata that can be stored in various other files. There may be requirements to add in additional information for the data to be understood and processed. This can take many different forms, such as the need to define units or locations that the data was taken from. These can also be used in the codex as different file formats often require different metadata to be ran properly. The goal is that the simple string key value pair property will be sufficient to meet the needs.



The primary data storage element is one of the important elements that will need to be utilized to create a custom file reading codex. When a new file format reading codex is read and understood there may be a situation where the structure of the DEF file is insufficient for the purposes at hand. When this happens, the best idea is to create a derived class from this one. The use case for this functionality is when the metadata for the source file is embedded with the data in such a way that without completely rereading the file. Various streaming file formats are structured around storing pertinent meta data close to the actual data. These types of data formats may be necessary to keep a list of pointers to the structures where the data is stored. An overarching way of thinking about this is how to be able to transform the data when required without completely rereading the file. However, the metadata in the file will eventually need to be transformed into a DEF format. The next section defines when the need to have the format transformed will take place.

## Call Structure

The plugin codices need to be in a directory. The directory will be passed to the application through a command line parameter. The system will then scan the entire directory of plugins. The plugins each are required to have an identification. The identification for the different codices are used in the command line as written in the codex. It is a simple error for these to not be mapped together. Furthermore the identifier for the codex is the same for the input type and the output type should the codex need to be used.

Once a codex is found, the function that will create the working code base is the constructCodex() function which is in the dllmain.cpp file. This function will call the constructor of the class and produce the necessary pointer to use. This means that any setup required for the codex should take place in the constructor.

The directory selected as the input directory will be the starting point for inputting data into the system. This directory will be read by the system and return a list of names for each of the different files in the system. The files will then be passed to the selected input codex for reading. The specific method that is called is processFile in the codex base. The codex will need to implement this file in whichever way the data format would be required to handle it. The actual selection for the correct file format also occurs in this method. Any required function such as file extension checking or magic number checking occurs here.

From this point anything can happen and the codex takes over to process the file however it needs to. This is the point where the file format that is being read is being processed. There is complete freedom to do whatever is necessary to read the file.

Once the system has completed reading the files, the next step is to take the data and load it into the system. This is accomplished through a structure that is called currentStorageElementGrouping. This structure stores the elements that are read from the file. Each of the individual PrimaryDataStorageElements that are found in the file are loaded into the currentStorageElementGrouping to expedite the process so that multiple different calls to upload information is not made to the storage system. The reason the system behaves this way is so that multithreading can occur and many different parallel file reading codexes can be used at the same time. It is necessary for the codex to add multiple different elements to the currentStorageElementGrouping, each of the different data storage elements represent a single array of data to be processed. At this point the data has been successfully incorporated into the system.

After the files have been successfully read the system will do much the opposite and output the data that has been stored in the system. The output format specifier passed into the system as a command line argument will specify how the data will be output. The data stored in the main data store will be iterated through and the codex that was used to create the data record will be used to actually convert the data, not just the metadata, into the DEF file format. Once that is completed the record is passed to whatever codex is selected to output it. This will output all the selected data and then exit the program.

## System Flexibilities

By using the plugin system there is no limit to the type of data that can be passed into the system. This is in no way limited to data that is in a normal numeric format. This can be bizarre things such as image, sound, or video files. If the file format can be logically separated into numeric arrays there is no limitation to what can be done. For instance a plugin may contain a completely built in encrypting system that is entirely self-contained in the plugin. There is also full capability for the plugin to access network resources or even custom hardware. This provides extensive customization capability to fill any perspective use case.