Rust Implementation of the ANSI E1.31-2018 sACN Protocol

Paul Lancaster

University of St Andrews

0.1 Abstract

The project aims to create a library that is available in rust allowing usage of the ANSI E1.31-2018 sACN [3] protocol including data transfer, universe synchronisation, universe discovery and that supports Ipv4, Ipv6, Unix and Windows. This library utilises an existing implementation [**?**] as a base but does not attempt to provide backwards compatibility.

0.2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. "The main text of this project report is NN,NNN words long, including project specification and plan. "In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bonafide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

0.3 Introduction

Introduction Describe the problem you set out to solve and the extent of your success in solving it. You should include the aims and objectives of the project in order of importance and try to outline key aspects of your project for the reader to look for in the rest of your report. At the time of starting this project there did not exist a native, open-source ANSI E1.31-2018 sACN library for rust which allowed data sending, receiving, universe synchronisation and universe discovery. The library that did exist was incomplete and only supported sending data and parsing packets.

This project therefore aims to create a library for rust users which supports ANSI E1.31-2018 sACN sending, receiving, universe synchronisation and discovery. This project is based on the existing library to give a starting point. As sACN is commonly used in heterogeneous device environments with a mix of embedded systems and operating systems such as Windows and Unix to provide support for as many devices as possible a few additional non-functional requirements were made; The library must have support for both Ipv4 and Ipv6 as well as unicast, multicast and broadcast in both windows and unix environments. Backwards compatibility with the existing library was abandoned due to the incomplete nature of the library and to re-use it would require significantly forcing the implementation of the new library into confusing patterns.

0.4 Context Survey

Context survey Surveying the context, the background literature and any recent work with similar aims. The context survey describes the work already done in this area, either as described in textbooks, research papers, or in publicly available software. You may also describe potentially useful tools and technologies here but do not go into project-specific decisions.

0.4.1 DMX, SACN and ACN

DMX512

DMX512 is an protocol used in the entertainment industry for the control of lighting, effects and other devices. It works by daisy chaining devices together into distinct physical chains (called universes) and is a one way protocol. This means that the devices in the line cannot communicate their presence back to the controller so the controller must know about the devices ahead of time and their addresses so it can broadcast packets down the line which the devices then receive and use. The DMX packets are a

fixed size and contain five hundred and twelve 8-byte channel (+ a start code) which allows them to control up to 512 different devices on a singular line. A device may support the use of multiple channels to control different functionalities so for example a light with RGB colour mixing may use 3 channels to allow control of the Red, Green and Blue individually. Since there are only 512 channels available on a single universe this quickly imposes a limitation to the number of devices that can be connected together, especially as modern lighting fixtures commonly use upwards of 30 channels each for a moving light with usage of many more not uncommon. The solution to this was previously to simply have more physical lines (universes) and in this way allow more devices to be controlled simultaneously. This comes with a number of problems however as each new physical line means a new cable coming directly from the control desk.

DMX512 Problems

As the control desk is often far from the devices themselves (at the back of the venue whereas the lights/devices are above the stage) it means that many cables need to be run which can be expensive and time consuming.

The length of the cable runs can cause signal interference / degradation and DMX as a 1 way protocol does not have any error correction (bad frames if detected are thrown out).

The protocol only allowing 512 channels per physical line means that a device cannot have more channels than this. This is particularly a problem recently with the advent of complex fixtures which may have many LED's with individual colour control.

sACN

One solution to solve some of the problems with DMX is to send it using UDP over a standard IP based network and one of the protocols created to do this is sACN. This allows many DMX packets (and so many universes) to be simultaneously sent using a single network cable from the console and then to be received by the devices. Often for backwards compatibility reasons the sACN is converted back into DMX packets before being sent to the device as most devices older than a few years do not support direct sACN communication but this is rapidly increasing - particularly with higher end professional fixtures.

sACN - Universe Synchronisation

A potential problem with multiplexing multiple universes down a single network line is that two universes of data cannot be sent simultaneously, this

is often not a problem but for receiving devices that span multiple universes receiving one packet before the another may put the device into an inconsistent state. A similar problem arises if two different devices on different universes want to be controlled simultaneously. ANSI E1.31-2018 provides a solution to this problem in the form of the universe synchronisation feature. This works by data packets containing a synchronisation universe field which can be set to a specific universe. On receipt of a sACN packet with a non-zero universe synchronisation field a compliant receiver won't act on the packet immediately and instead will hold the data for that universe. This data will then be acted upon on receipt of a universe synchronisation packet with the corresponding universe. As data packets for multiple different universes can specify a single synchronisation universe this allows data for multiple universes to be acted upon simultaneously on receipt of a universe synchronisation packet.

sACN - Universe Discovery

sACN allows sending on upto 63998 universes with each universe having a unique multicast address. Any of the universes can be used by any source and so in initial versions of the protocol such as ANSI E1.31-2009 [10] the only way to learn which universes were in use were either to have prior knowledge or to scan every single possible address and listen for packets. This is very inefficient and impractical in a real-system especially as in the time that a universe was last scanned another source might have joined and started transmitting. Universe discovery solves this problem through the universe discovery mechanism. This mechanism works by having a reserved universe of 64214 (as defined in ANSI E1.31-2018 Appendix A) on which sources send universe discovery advert packets. These packets contain a list of universes that the source sends which is referred to as a universe page. Each page can hold 512 universes and so therefore a source may send multiple discovery packets each with a different page that the receiver can then put together to build up a complete list of universes that the source is sending. To allow a receiver to know when all the pages have been received for a given source each universe discovery page has a numbering which increases sequentially with the number of the last page expected included. By having multiple pages it prevents the protocol being required to send large packets on the network (size limited by page size not by the much larger number of possible universes). This is advantageous as it prevents problems with sending large packets such as causing a-lot of fragmentation at the link layer which will fragment packets into frames that are the size of link-layers maximum transmission unit (e.g. 1500 bytes for ethernet [20]).

0.4.2 Critical Analysis of the sACN protocol

ANSI E1.31-2018 sACN over a purely DMX network provides a solution to a number of problems as discussed above but also introduces its own flaws.

One flaw comes with the concept of universe synchronisation as it relies on all receivers to receive a universe synchronisation packet simultaneously. In real-world networks however this may not be the case and depending on the complexity of the network varying transmission latencies between devices may mean that even with synchronisation multiple sources act on data at different times. Another potential issue with the protocol is that it provides no protection from malicious or malfunctioning sources taking control of the system. This makes isolation, preferably physical, of the network vital and so ANSI E1.31 sACN is commonly used on networks dedicated to lighting protocols. This also helps reduce the issue of variable transmission latencies as these networks are likely to be fairly simple. Even with isolation from malicious users the sACN protocol is still vulnerable to problems related to byzantine failures where devices fail but rather than doing so cleanly instead produce random values which are interpreted by devices on the network as intentional and can cause the system to act unpredictably. These failures are not-uncommon in networks using cheap, knock-off devices which might not be fully compliant with the protocol even if they work most of the time.

0.4.3 Rust

Rust [6] is a compiled memory safe language with no garbage collector. It is extremely fast with near C/C++ like performance [17] but with a much stricter compiler that guarantees memory safety. As Rust has no runtime due to no garbage collector it is applicable to embedded devices and high performance applications making it an ideal language for an ANSI E1.31-2018 sACN device which are often embedded (e.g. in lighting fixtures) or controlling multiple steams of data in real-time where minimal latency is vitally important (e.g. lighting controllers).

0.4.4 Related Work

The ANSI E1.31 sACN protocol was originally specified in the document ANSI E1.31-2009 [10]. This represented the base version of the protocol without any universe synchronisation, universe discovery or discussion of operation with Ipv6. Since then it has been revised in 2016 (universe sync and discovery) [11] and again to its current latest version in 2018 (Ipv6). The future of ANSI E1.31 is still being actively developed and discussed [12] with the direction of the ACN eco-system of lighting control data over IP being focused on supporting communication from receivers back to sources. Within traditional DMX systems this is supported using the remote device

management protocol (RDM) as described in ANSI E1.20-2010 [15]. An IP version of the RDM protocol was then created (RDMnet) [13] which is ACN based and allows discovery and control of receivers over a network. RDMnet as a fairly new protocol is still in the process of being taken up by vendors but has strong support from ETC (a large lighting company [16]) in the form of a maintained open source implementation of RDMnet in C++ [14].

The ACN based family of lighting control protocols aren't the only protocols that allow sending DMX data over an IP network. Another widely adopted protocol is ArtNet which at time of writing is in its 4th version. Unlike sACN on its own ArtNet allows discovery of receivers, remote configuration and transporting RDM data [**?**] in addition to sending data. ArtNet therefore has taken the strategy of being a larger protocol which covers many use-cases as opposed to the ACN strategy of many protocols each doing a specific area that inter-operate. While they are developed independently the ArtNet v4 standard does allow managing sACN devices which means that it can be used to configure/control sACN devices but with the data still sent over sACN [7, Pg. 3].
https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html]

There are a number of existing implementations of sACN in rust however none are fully compliant with the protocol as specified in ANSI E1.31-2018. One of the most complete is [2] which was used as the base for this project. As this is hosted on github it can be seen that while there are a number of forks (6 at time of writing) no public fork has any further progress which leads to the conclusion that this is the most complete open source rust implementation available. Note that this implementation appears in a number of places such as [8] but this is still the same implementation.

Implementations of sACN exist in multiple languages, at the time of writing (Jan 2020) a cursory search for E1.31 repositories on github reveals repositories in multiple languages with the most prevalent being C++ and C as shown by Figure: 0.4.4. An example of one of these projects is **??** which allows both sending and receiving of sACN packets.

0.5 Requirement specification

Requirements specification. Capturing the properties the software solution must have in the form of requirements specification. You may wish to specify different types of requirements and given them priorities if applicable.

The project was split into the following list of primary and secondary functional requirements
Primary

Allow sending and receiving DMX data over sACN.

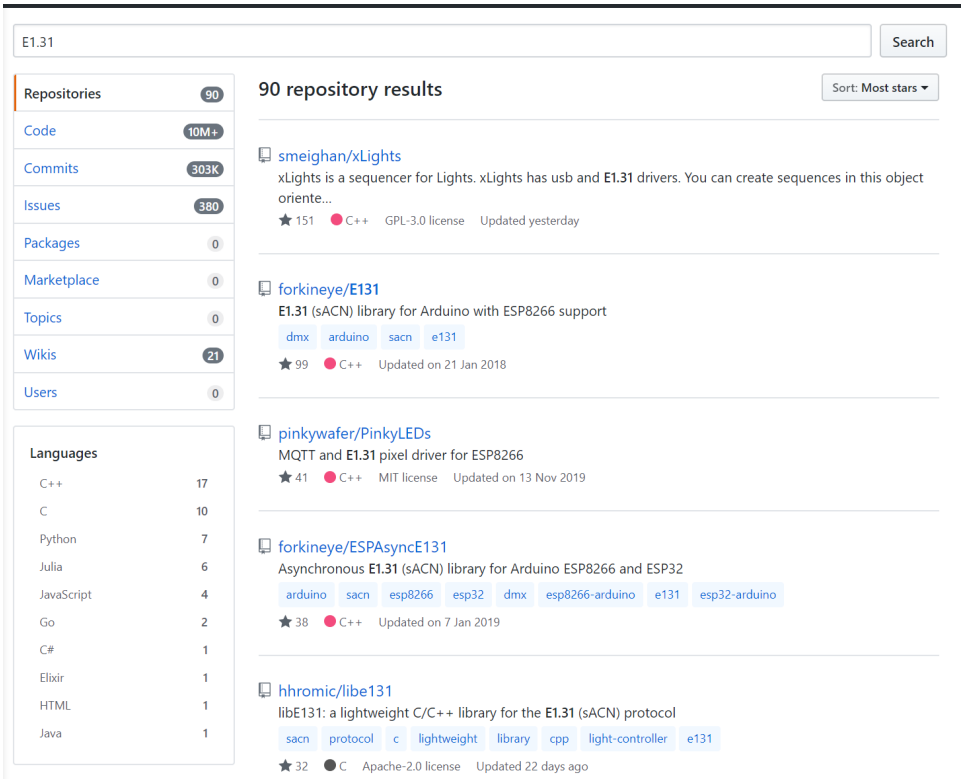

Figure 1: A search of repositories on github with the search term "E1.31" as of Jan 2020

Support the sending and receiving of cross universe DMX data through the universe synchronisation feature.

Support universe discovery with adverts for sources and discovery for receivers.

Secondary

Demonstrate a deployment of the library into a real-world system to show its compliance with the protocol by showing interoperability with other compliant devices.

Provide support for Windows 10 and Fedora Linux systems.

Support multiple IP transmission modes - Unicast, Multicast and Broadcast.

Support multiple IP versions - Ipv4 and Ipv6.

0.6 Software Engineering Process

Software engineering process. The development approach taken and justification for its adoption.

A waterfall based process model was used for the development of the program. In the waterfall method there are several distinct phases of the project as shown in figure: 0.6 which follow on from each other with loops back possible if a problem is found at a later stage. This development approach was chosen as it has a very clear structure which allows easy to manage distinct milestones so progress through the project can be more easily tracked. This process method has a number of disadvantages as-well with the main one being the inflexibility - if something major needed to change it would be difficult to adapt the project. As this project is based on a clearly defined specification provided by the protocol specification and the domains were clearly defined at the start it means that this inflexibility isn't a major issue and so therefore choosing the waterfall method for its advantages makes sense.

The waterfall model can be clearly seen throughout the development of the program. The first phase of 'requirement analysis' is the protocol specification itself as it clearly lays out the goals of the protocol and what it is required to do. On top of this there is the project goals which were defined around the protocol specifically for how much of the protocol this specification should implement for example universe-synchronisation, IPv4/IPv6 support, Unix/Windows support etc. When take together this gives a clear list of requirements as so allows moving onto the 'system design' phase.

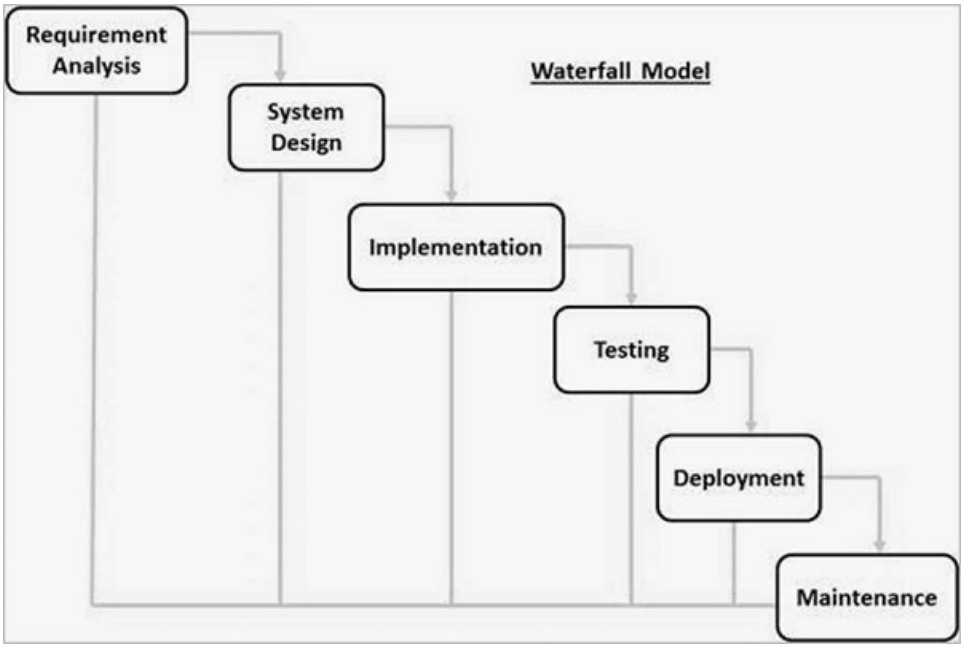

Figure 2: A diagram showing the waterfall development process, [[19]]

The system design phase is where the requirements are turned into a technical plan for how they will be implemented. Most of this comes from the protocol specification itself as it describes how each bit of a compliant implementation should behave and so therefore the design can be based of this. This combined with the existing base incomplete implementation that was used meant that the general system design was built around this. In general the system was designed around there being distinct receivers and senders with all communication being one-way with no expected replies. This meant that the two different sides could be developed in relative isolation as all their communication must be done in a way that is compliant with the protocol which provides the interface between them.

0.6.1 Implementation, Testing and Deployment Phases

As most of the design is already provided in the form of the protocol specifications and the requirement to make the project in rust it means that the project is mainly focused on the Implementation, Testing and Deployment phases. The implementation phase is one of the biggest in this project and represents the actual creation of the code as discussed in more detail in the Implementation section. As part of the engineering process there was an amount of looping between the implementation and testing phases. This was done as each part of the code was implemented (for example adding universe synchronisation) which was then tested by creating some initial tests

to check that the design for that section has been implemented correctly. Then the implementation phase was revisited to either fix a discovered bug or to implement the next section. This looping is similar to the way that a test-driven-development methodology might work however the waterfall methodology described here is distinct as the implementation is written before the tests.

The implementation is known to be complete when all the functionality specified in the design has been implemented. In this project this is represented by data sending, universe sync and universe discovery all being implemented on both the sender and receiver. At this point the project moves into the testing phase. The focus now becomes on verifying that the implementation is correct with respect to the design (compliant) using a holistic view with all parts put together as-well as ensuring the documentation matches the actual behaviour. During this stage it is possible that bugs or areas where the implementation isn't compliant with the protocol specification may be discovered. In this case the focus will move briefly back to the implementation stage to fix the problems before progressing back through to the testing phase. It is possible at this phase that a design problem is encountered, for example if it was found that the structure of the program didn't support a functional or non-functional requirement. If this happens then at that point the engineering focus would move back to the design stage and as per the waterfall model the focus would then continue through the process of the implementation and testing phases. The testing phase is signalled as complete when there is sufficient tests that verify that all functional and non-functional requirements have been met. What counts as sufficient is discussed in more detail in the testing section.

The next phase if the deployment phase, within the project this is where the finished and tested code is given to users to use. In this project this is shown by the real-world and acceptance tests. These tests fall across the boundary of the testing and deployment phases as they both verify the system works but also show that it is sufficiently mature that it can be used by an intended end user. For this project the intended end user is a software developer creating a program which allows usage of the E131 protocol. Having an actual developer use the library is beyond the scope of the project however the demo sender and receivers act as an example of a possible deployment. Since this example is then demoed by interacting with a real-system and this is shown to someone who actually works in the field it shows that the project has reached the stage of being deployed. As part of this stage it also includes the packaging of the project so that it can be used by developers including the finalisation of documentation and a list of dependencies, once this is complete and the demo programs have been packaged the deployment stage is complete. This is the point at which the

scope of the project ends as the final 'deployment' is marked by the final submission.

The final stage is the maintenance stage, this falls out with the scope of this limited time-period project however in a real-world project this represents the process of users reporting bugs, problems, feedback and developers looping back to one of the various stages such as design, implementation or testing to verify the problem and implement a fix. While not part of the project directly it is hoped that the library will be able to be contributed back to the community e.g. through the rust cargo repository and GitHub and by doing so the maintenance stage can begin.

0.6.2 Reflection on Methodology Used

The approach fit the project well as it made it clear which stages the project was focused on (implementation, testing, deployment) with the previous stages (analysis, design) clearly shown by the protocol specification. The methodology did require increased up-front work as implementation could not begin until the analysis and design states were complete. This up-front work came in the form of the initial documentation for the project such as the DOER list of objectives and as this is required anyway this isn't a problem for this project. The methodology also meant that there was the risk that too long could be spent on one stage which delays further stages and therefore the entire project doesn't reach the deployment stage by the fixed deadline. This meant that a time-line had to be created early on to mark when various parts of the project would be complete so that progress could be tracked. This was attached as part of the project in the 'Objectives with times.txt' file and its creation and modifications are shown by the git-version control which shows how it changed. This was later superseded by minute notes at weekly meetings where the project and its progress were discussed. Taken together these show how the project has developed and how the requirements have been changed from those originally proposed due to time-constraints.

A high level view of the development of the project over time is shown in Figure 0.6.2. This shows that the waterfall methodology was followed starting with the deployed existing library which is moved into the requirement analysis stage as new requirements are set as discussed in the requirements section. The requirements are then turned into a design using some of the structure provided by the existing base implementation and a set of milestones created. As discussed above the project then enters its main stages of implementation and testing which loop around as sections of the program are created, tested and debugged. It can be seen that at on the 12th of

January the project had to loop back 2 steps to the design stage and this was due to the non-threaded structure of the program being insufficient to allow the periodic universe discovery adverts and so the design had to be changed to a threaded structure to allow this. The implementation of this new design and testing was then performed as part of the next 2 steps as per the waterfall model. This was then followed by a stage of further testing indicating the start of the test phase. This phase also included instances of steps back being required such as the bug found on March 7th which required implementing a new OS specific socket handling mechanism to fix. The testing phase then continued up until the code was ready to be demoed in the real-world environment which marks the transition from the testing phase into the documentation phase. The documentation phase then continued right up until the final deployment marked by the final submission.

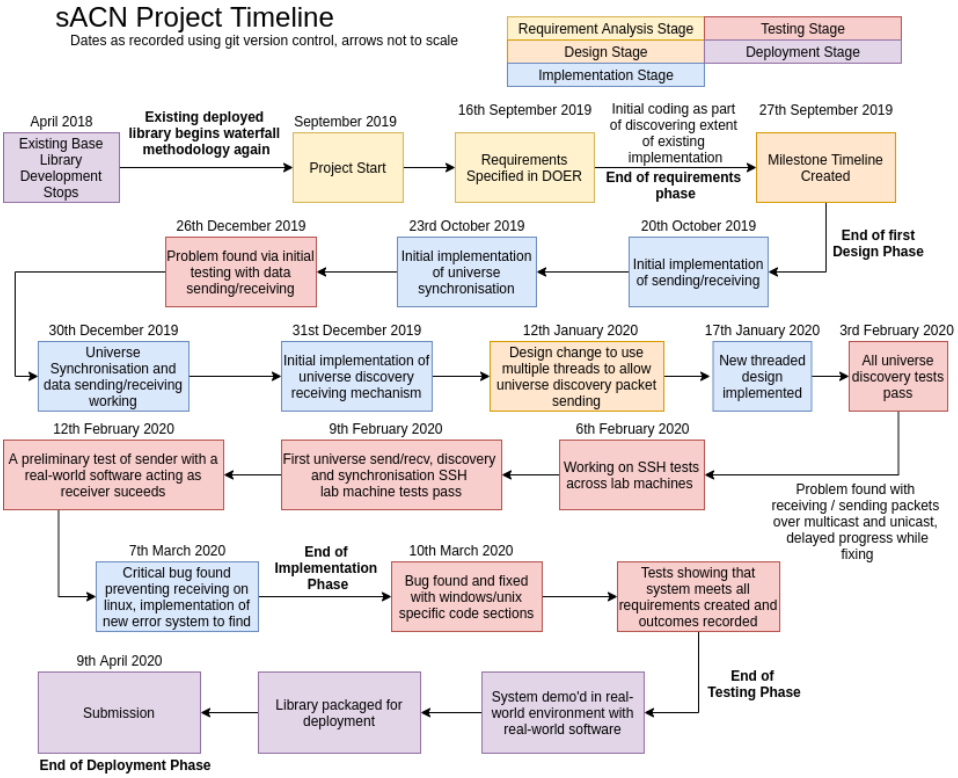

Figure 3: The development of the project over time with the water-fall methodology stages marked

0.7 Tools & Technologies

- Rust - Error-chain - Net2/Socket2 - Cargo - Run - Docs - Test - Wireshark - ACN filter - Vision - What visualisers are - sACN Viewer - Checking formatting - Git, Github, Gitlab - Version control

0.8 Ethics

This project has no ethical considerations that require notification in this section.

0.9 Design

The project was based around creating a library that is high-level enough that a user can easily use it for its primary functionality. This meant providing a clear set of public facing functions/methods which don't rely on the user understanding all aspects of the protocol and instead just the bits they are interested in. For example the SacnReceiver.recv(timeout) method which takes just a timeout (using None to indicate no timeout) and returns DMX data. This completely abstracts away many aspects of the protocol from a user who most of the time will want to just get data and perform actions with it. For example this function abstracts the concept of universe synchronisation with data only retuned when it is ready to be acted upon without users having to know when they can use the data.

Design Indicating the structure of the system, with particular focus on main ideas of the design, unusual design features, etc.

– Intended end user documentation vital -¿ maintenance -¿ developers will use the library test-¿ compliance -¿ conformance -¿ what it means for there to be 'sufficient' testing

0.10 Implementation

Implementation How the implementation was done and tested, with particular focus on important / novel algorithms and/or data structures, unusual implementation decisions, novel user interface features, etc.

0.10.1 Std vs Non-Std

0.10.2 Implementation dependent specifics

0.11 Evaluation and Critical Appraisal

Evaluation and critical appraisal You should evaluate your own work with respect to your original objectives. You should also critically evaluate your

work with respect to related work done by others. You should compare and contrast the project to similar work in the public domain, for example as written about in published papers, or as distributed in software available to you.

0.12 Conclusions

Conclusions You should summarise your project, emphasising your key achievements and significant drawbacks to your work, and discuss future directions your work could be taken in.

0.13 Appendices

The appendices to your report will normally be as follows. Testing summary This should describe the steps taken to debug, test, verify or otherwise confirm the correctness of the various modules and their combination.

0.14 Testing

0.14.1 Automated Testing

0.14.2 Real-world Testing

0.15 User Manual

User manual Instructions on installing, executing and using the system where appropriate.

Building documentation from source: cargo doc –document-private-items –no-deps –open

0.16 Other Appendices

Other appendices If appropriate, you may include other material in appendices which are not suitable for inclusion in the main body of your report, such as the ethical approval document. You should not include software listings in your project report, unless it is appropriate to discuss small sections in the main body of your report. Instead, you will submit via MMS your code and associated material such as JavaDoc documentation and detailed UML diagrams

Bibliography

[1] ANSI E1.17 - 2015 Entertainment Technology?Architecture for Control Networks

[2] https://github.com/lschmierer/sacn (September 2019)

[3] ANSI E1.31 ? 2018 Entertainment Technology Lightweight streaming protocol for transport of DMX512 using ACN

[4] https://www.element14.com/community/groups/open-source-hardware/blog/2017/08/24/dmx-explained-dmx512-and-rs-485-protocol-detail-for-lighting-applications (17/09/2019)

[5] https://github.com/hhromic/libe131 (17/09/2019)

[6] https://www.rust-lang.org/ (17/09/2019)

[7] http://artisticlicence.com/WebSiteMaster/User%20Guides/art-net.pdf (17/09/2019)

[8] https://docs.rs/sacn/0.4.4/sacn/index.html (26/01/2020)

[9] https://github.com/hhromic/libe131 (26/01/2020)

[10] https://tsp.esta.org/tsp/documents/docs/E1-31_2009.pdf (26/01/2020)

[11] https://tsp.esta.org/tsp/documents/docs/E1-31-2016.pdf (26/01/2020)

[12] http://www.rdmprotocol.org/files/What_Comes_After_Streaming_DMX_over_ACN_%20%284%2 (26/01/2020)

[13] RDM-NET

[14] https://github.com/ETCLabs/RDMnet (26/01/2020)

[15] RDM

[16] https://www.etcconnect.com/About/ (26/01/2020)

[17] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html (28/01/2020)

[18] https://www.techrepublic.com/article/rust-programming-language-seven-reasons-why-you-should-learn-it-in-2019/

[19] https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm (01/01/2020)

[20] https://tools.ietf.org/html/rfc894 (10/03/2020)