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]).

It should be noted that a receiver will receive and act on data packets from a source even if it hasn't been 'discovered' yet. This means that the

number of sources communicating over multicast is completely transparent to the receiver meaning it places no limit on the number of allowed sources which allows the system to scale if required.

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.

The protocol also suffers from the same problems that many similar protocols do related to trying to maintain backwards compatibility, particularly with DMX. This imposes a number of limitations and inefficiencies. One example of this that each sACN packet sends a single universe limited to 512 channels, this is far less payload than the packet could actually hold, even if 2 universes were sent in one packet it would half the number of packets required and produce packets of size 1150 bytes (current size: 637 bytes + a universe (513 bytes)) which is less than the MTU of many common link-layers e.g. Ethernet. In addition to this the concept of universes themselves limits the protocol as problems due to devices being unable to lie across universe boundaries have been carried over into the protocol and solved e.g. universe synchronisation when actually if DMX was completely removed from the system the problems wouldn't exist - a packet could be sent for every device or group of devices with variable parameter counts meaning redundant data isn't sent. The protocol layers (UDP + sACN) also add a significant amount of over-head, for a full universe of data which takes up 513 bytes (512 DMX channels + a startcode) the packet size is 637 bytes

meaning an overhead of 124 bytes, corresponding to 19.5% and if the universe is only partially full the ratio of overhead to actual data gets worse (1 byte of data + 1 startcode leads to a packet that is 98% overhead).

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].

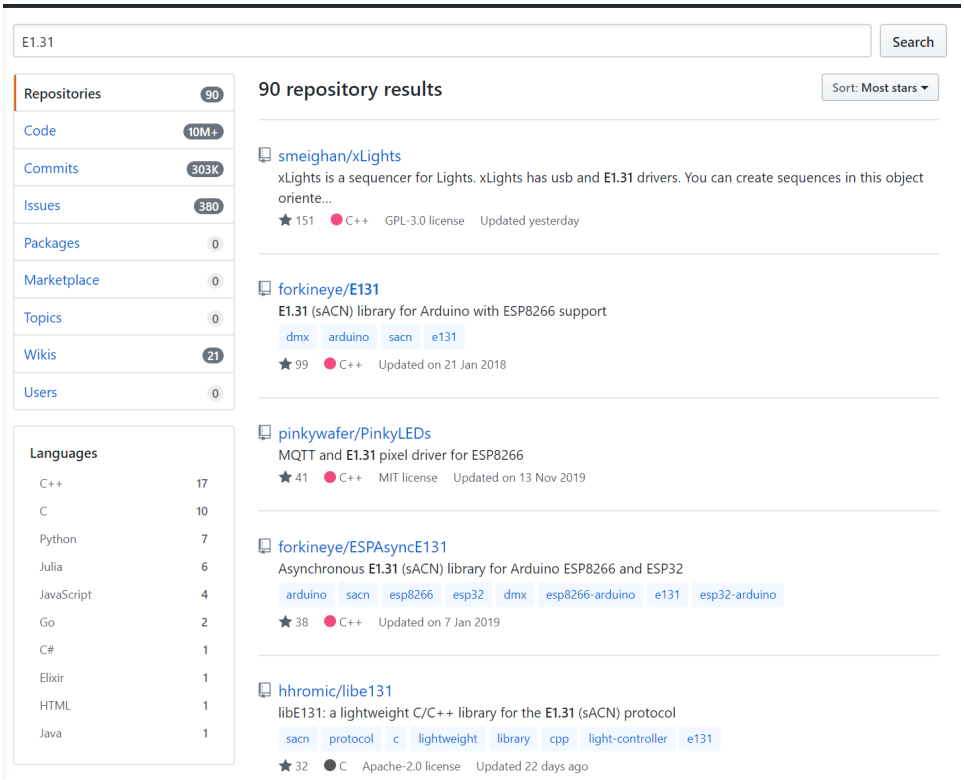

Figure 1: A search of repositories on github with the search term "E1.31" as of Jan 2020

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.

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.

The intended user for this library is a software developer developing applications that utilise the sACN protocol. It isn't designed to be used directly by an end user as it is just a library which needs to be used in code to actually perform any actions. This means it needs to be able to be understood and utilised by someone who is familiar with general software engineering and the main ideas of sACN. This makes technical documentation of the project code such as comments, API explanations and examples a vital part of the project as otherwise developers won't want or be-able to use the library.

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

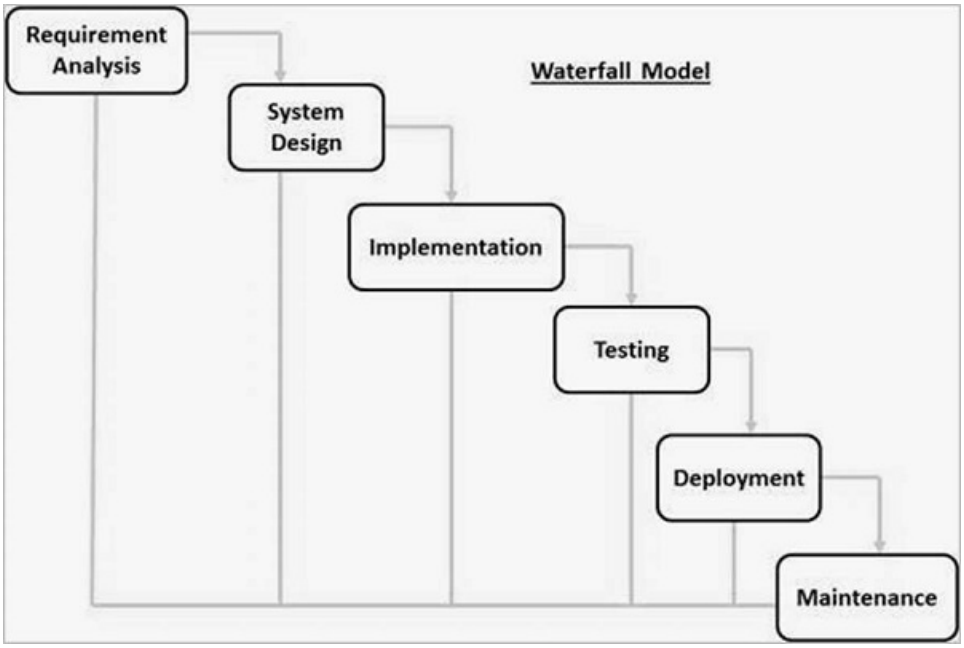

Figure 2: A diagram showing the waterfall development process, [[19]]

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.

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.

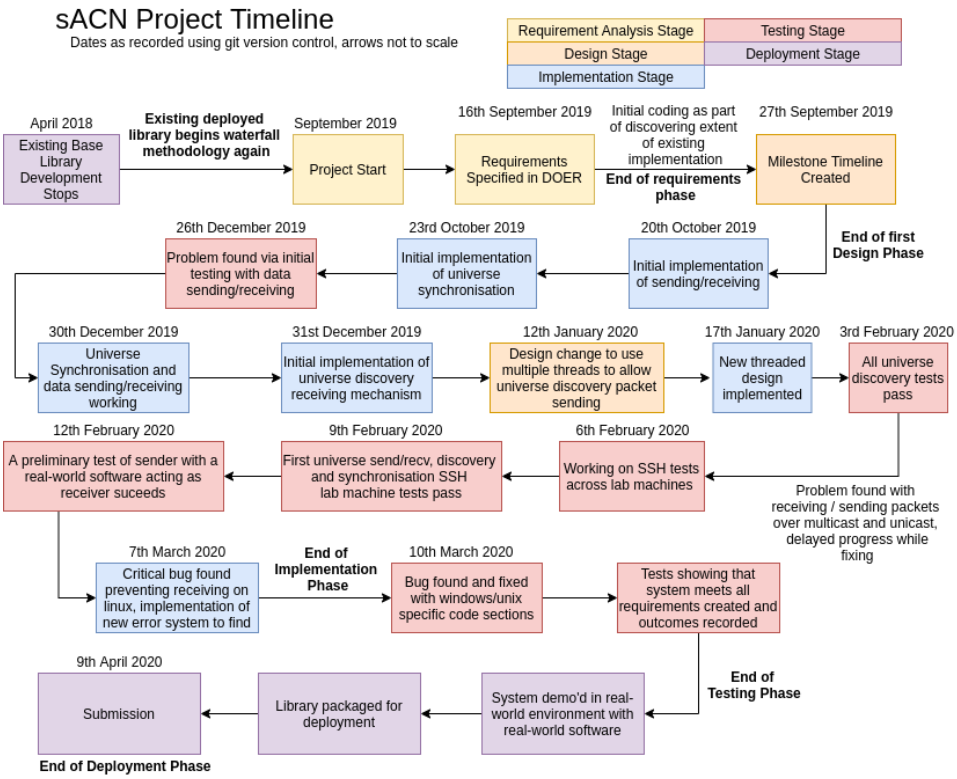

Figure 3: The development of the project over time with the water-fall methodology stages marked

0.7 Tools & Technologies

0.7.1 Language: Rust

Error handling: Error-chain

– Error-chain, specific sACN, Io, Net errors.

Networking: Net2 / Socket2

0.7.2 Dependency Management: Cargo

Within rust libraries/packages are referred to as crates and the management of dependencies is handled by Cargo. This system allows fetching of dependencies as required during the build stage and includes automatic handling of sub-dependencies etc. In addition to this the Cargo system provides many commands related to testing, compiling, documenting and creating rust applications.

Run

The cargo run command allows checking of the rust code for compile time errors, fetching of dependencies, compiling/linking of the produced rust binary and finally running the produced binary all within one command. This greatly simplified development as there were no Makefiles or similar to manually manage and the code can easily be moved to a new system for development/testing as required libraries are fetched automatically.

Docs

As discussed the targeted end user of the library is a software developer. This makes good comprehensive documentation vital so that developers know what each part of the code does and as this project is expected to eventually be released open-source having good documentation allows new library developers to come in and maintain/expand the code base.

Documentation within the project is done using the rustdoc system which is included as part of the rust/cargo development package. This library is very similar to those found in other languages such as Javadoc for Java which work by having documentation embedded within the code which is then transformed into a HTML web-page to provide the documentation for the crate. As it is directly embedded into the code this makes it less likely that it will fall behind as the documentation and code are close together and a developer can change both simultaneously without having to work across multiple different documents. As this is part of the Cargo system it allows the library to packaged up along with its generated documentation automatically so that when it is distributed onto the cargo repository the documentation can be readily accessed alongside.

Test

One of the verification methods used with the library is unit tests, these are small self contained tests which can quickly run to verify that a small part of the program behaves as expected. Rust comes with a built-in form of unit testing through usage of the cargo test command. This automatically finds all tests within the code as designated by the test annotation and runs them producing a list of which tests passed and failed. This also includes other tests such as examples in the documentation which helps to prevent problems where examples are forgotten about and as the development progresses become depreciated or broken. Cargo test was therefore an important part of the project during development and remains an important part once the code is in the maintenance phase.

0.7.3 Debug Tools: Wireshark

- ACN filter

0.7.4 Compliance Testing Tools: sACN Viewer

- Checking formatting

0.7.5 Real-world Usage Tool: Visualisers: Vision

- What visualisers are

0.7.6 Version Control: Git, Gitlab, Github

0.8 Ethics

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

0.9 Design

0.9.1 Expected System Usage/Layout

– Multiple senders, multiple receivers – Packet structure

Universe Discovery

– Periodically send universe discovery packets

Universe Synchronisation

– Send data then send sync, sync can come from anywhere

0.9.2 Network Layers / Transport Modes

– Sits on top of UDP – Unicast, Broadcast Multicast

Multicast Address Assignment

– Universe -¿ Ipv4 and Ipv6. – Is in the IETF range...

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

–

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 SacnReceiver

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. This also abstracts away having to handle universe discovery packets, if a universe discovery packet is received then by default it is handled silently and added to the receivers list of discovered sources. This allows the user the flexibility to inspect the list of discovered sources if required but otherwise they can be ignored. This decision to not explicitly announce the discovered universes was made with the observation that a receiver spends most of its time receiving and processing data rather than listing discovered sources - especially as a source doesn't have to be 'discovered' to allow receiving on. This design decision comes with a few trade-offs, first it means that to discover sources a receiver must first attempt to receive data even if it doesn't act on any received data. This leads to a pattern of attempting to receive data with a short timeout and then inspecting the list of discovered sources. This is slightly more work for a library user trying to discover sources but allows the flexibility for the user to choose how they want to handle data received in this time (throw out or handle). The library does allow setting a flag on the receiver to change this behaviour, if the 'announce_source_discovery' flag is set to true (default is false) then on completely discovering a source the method will return a SourceDiscovered error. This still means any data received will need to be handled by the user but means that if no data is received it prevents the method having to wait for the entire timeout if a source is discovered. By having this off by default it keeps the method functionality simple for the expected majority of the receivers time when it is just receiving data but allowing the flexibility for use-cases that require knowing whenever a source is discovered. An error was used as oppose to a special return value for a similar reason - to keep it simple for the major use

case. This is similar to how the underlying socket will return a WouldBlock or TimedOut error if it doesn't receive within the given timeout, this isn't explicitly an error that means the program has to stop operation but it is something that the user should handle. Since errors are handled as part of the type system adding an error also doesn't add significant performance overhead.

0.10.2 SacnSource

Unlike the receiver on the sender side universe synchronisation has to be handled explicitly. Firstly when data is sent there is an optional synchronisation universe argument. The rust built in Option type makes this simple to ignore if not required as this can be set to None to indicate no synchronisation. When data is sent with synchronisation it won't be acted upon until a synchronisation packet is send using the 'send_sync_packet' initially the sender would automatically send this sync packet after the data was sent however this was removed and an explicitly split up method used instead. While slightly more involved this provides greatly increased flexibility because with the other method there was no way for a source to send data and either synchronise it later or to allow another source to generate synchronisation packet. While not a hard requirement it is advised in the standard (ANSI-E1.31-2018 Appendix B.1) that there is a small delay between sending data and sending the synchronisation packet to allow receivers time to process the data. This isn't enforced by the library as what counts as a 'small' delay will depend on the system and so this is left up to the user to decide. Similarly as specified in ANSI E1.31-2018 Section 6.6.1 the send method shouldn't be called at a higher refresh rate than specified in DMX (ANSI E1.11) unless there are no E1.31-DMX converts on the network. Since this is also something which is system dependent and the library cannot know on its own this is also left to the user.

The send method also exposes the way that the information is sent on the network through the 'dst_ip' this argument allows a sender to send information using unicast directly to a source (or broadcast by providing the broadcast IP) but also allows usage of multicast (as described in design) by providing the None argument.

The sender is based on a multi-threaded model with an internal source protected by a lock encapsulated within the SacnSource that users interact with. The reason for this was to allow abstracting the task of sending periodic universe discovery packets away from the user. Instead on creation the SacnSource spawns a thread which will send universe discovery packets at the 10s interval defined in ANSI E1.31-2018 Appendix A. This leads to

the requirement for a user to register a universe before they can send data on that universe allowing it to be reflected in the discovery packets. By abstracting this behaviour away it makes it easier for the users to create a compliant source as the correct sending interval and formatting (including splitting into multiple pages) is handled for them. In some situations it might be required that universe discovery isn't used, for example if there are devices which implement ANSI E1.31-2009 which was created before universe discovery and which don't correctly discard packets with the wrong vector. To allow compatibility with these devices the sender provides the 'is_sending_discovery' flag which defaults to true but can be set to false to prevent discovery packets being sent.

0.10.3 Std vs Non-Std

The library is implemented assuming a std environment. This means that the rust std libraries such as the network library are available. This greatly increases the amount that can be done using rust as within the standard library the inbuilt functionality is fairly limited and would require rebuilding many already implemented solutions. This differs from the library that the implementation is based on which allowed running in environments with and without std. The reason to discontinue support for no_std environments was made as the new parts of the protocol which were being implemented such as universe discovery are significantly easier and provide a better user experience when parts of the standard library such as the threads can be used.

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.

There exists no fully-implemented public-ally available implementation of sACN in rust and the most complete version was used as the base of this project, this means that there is no direct comparison possible between this project and another however there do exist many implementations sACN in other languages so these can be used for comparison.

The decision not to pass up data packets awaiting synchronisation means that the packets must be temporarily stored within the receiver and this is done using a Vec data-structure which is a dynamically sized structure. This

means that the memory allocated to the program will continue to increase as packets are received which can be problematic for embedded devices with limited memory capacity. To limit this problem the implementation relies on the limited number of possible universes in the protocol and only stores a single universe of data for each waiting universe. This limits the maximum required space for this storage to 31.3MB + overhead which is not a problem for any modern PC but is potentially a significant amount for an embedded device. This means that the library is at risk of running out of memory for some devices such as arduino [22] which are commonly used for creating simple DIY embedded systems. This is only a risk on systems which have a large number of universes being synchronised at once so the for majority of usage cases where most universes aren't synchronised and only a few are synchronised at any one time this isn't a problem.

$$\textit{max possible universes} \times \textit{universe capacity} = 63999 \times 513 bytes = 32831487 bytes = 31.3 megabytes.$$

Similarly to this as the library is based on the std-environment it means the size of the produced binary will include the std libraries used, this means that the binary is bigger than it otherwise might be and isn't as tuned to the specific application from the perspective of performance. This also limits the usage of the library with certain disk-space constrained embedded devices. Support for embedded devices isn't an objective of this project so solutions to these problems such as not using std and putting smaller limits on the memory usage were not implemented as both solutions comes with associated trade-offs.

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.

– Finish full compliance, performance analysis vs C, see how the library does and if worth using in actual devices.

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

Throughout the project a priority was put on reproducibility and automation when it comes to testing. The reasoning for this is simple, once a

framework is setup it takes approximately the same time to run a test manually once or twice as it does to write the test in a way that it can be run multiple times automatically. This means that there is only a small penalty to setting up a test so that it can run automatically but once it is setup it can be run frequently allowing confidence that the code continues to work and that any change such as a bug fix for another test hasn't broken something else. Easily reproducible automated tests also provide a significant advantage to a project once it reaches the deployment/maintenance stage as they act as their own documentation of the code and a source of examples for new developers to use when learning. These examples are particularly good as they can be run to verify that they still perform as expected which can be used to flag up areas where the documentation and code have diverged.

0.14.1 Unit Testing

Unit testing is an vital part of many projects including this one. The reason for this is because each test is fairly small and quick to run many of them can be created to each cover a specific case. This means that once in place a developer can be confident that any change they make which breaks a part of the code will be quickly detected and as long as the unit tests are run frequently (aided by their quick running) the specific breakage can be found by following the tests. This relies on the unit tests sufficiently covering all the various functionality and expected outcomes and one measure for this is code-coverage. This doesn't verify that the code is of a high quality or that there is absolutely no bugs but does show that at least every part of the code is run as part of a test and coupled with good testing in general this gives confidence that the system behaves as expected.

https://martinfowler.com/bliki/TestCoverage.html

As described in the tools section unit tests created using the in-built rust/cargo unit testing framework. In addition to this the code coverage of these unit tests was checked using ***. The outcome of the unit tests as-well as details of what each test shows is included in the attached unit-test-outcomes.pdf. The specific tests themselves can be found in the *** file which shows the source-code for each test and allows them to be re-run to re-produce the presented results.

https://github.com/mozilla/grcov

0.14.2 Integration Testing

Unit testing can only be used so far and wasn't suitable for testing this project once pieces started being brought together. For this integration testing was required which are tests focused on how the system behaves once

the various parts are brought together. This includes testing the sender and receiver implementations with each-other but also compliance testing where the program is tested against other pieces of software that claim compliance with the specification and in-doing so this adds evidence that the library is compliant with the protocol.

The bulk of the integration tests were performed using the 'demo_src' and 'demo_rcv' programs which provide an implementation of the library for the sender and receiving sides and allow functionality to test specific areas of the library. These programs were then run remotely on different machines connected within a network using a specific combination of input commands to setup various test scenarios with the output from each program being recorded. This output was then compared against the expected output and used to detect if the test passed or failed. The abstract layout which shows how the components are logically connected as-well as the various actions and ordering is show in Figure 0.14.2. This also shows the actual physical test set-up to allow recreating the tests. All of these tests were performed between machines running the Fedora 30 operating system.

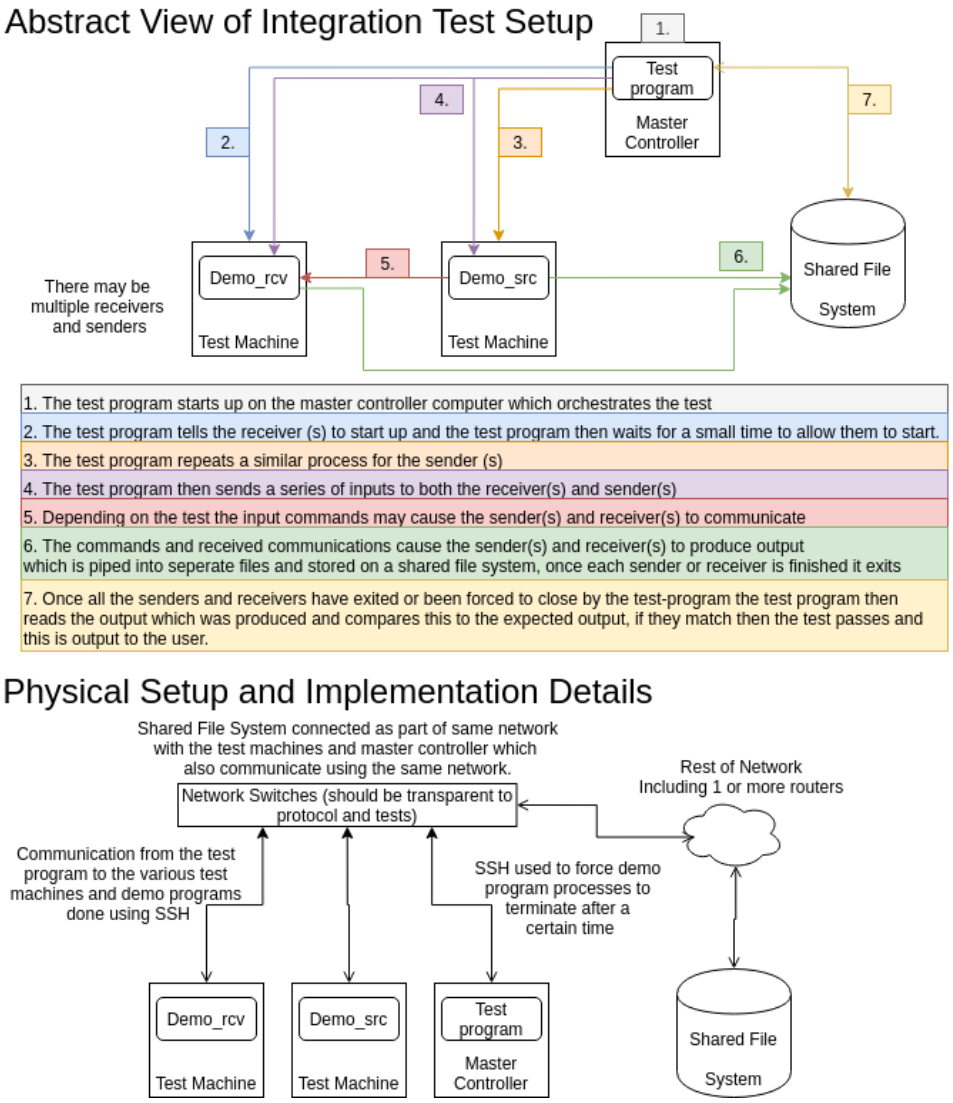

Figure 4: The abstract setup of the integration tests along with the actual implementation

The test-scripts used as-well as the expected output files and given inputs are included in the integration-testing folder. An explanation of what each file does is as follows...

Compliance and Conformance Testing

0.14.3 Real-world Testing

The final part of the testing acts as the bridge between the testing and deployment phases. This is when the program is put into a real-world scenario

and is actually used. This can highlight problems with the program which are hidden until this point such as gaps in the specification where the library behaviour is implementation defined and may not be compatible with other systems. It can also highlight parts of the system which perform slightly differently than described in the abstract specification due to the introduction of real-world factors such as real-equipment limitations like processing speeds. An example of this might be if the library absolutely relied on universe discovery packets being sent at exactly the interval as defined by the specification. In-real systems network delays as well as varying workloads on the devices might cause packets to be received at slightly variable intervals. Real-world tests therefore help find some of these problems and allow fixes to be made before the program is sent to users.

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)

[21] https://doc.rust-lang.org/1.7.0/book/no-stdlib.html (11/03/2020)

[22] https://www.arduino.cc/en/tutorial/memory (11/03/2020)