Using Formal Methods and Proof to Verify a CANDO Epilepsy Medical Device

Benjamin James Wooding
b.wooding1@newcastle.ac.uk

School of Computing Science, Newcastle University, UK

Abstract. Increased medical device complexity gives rise to more design errors found in the software of these devices and higher recalls and dangers to patients using such devices. This paper looks at an epilepsy optrode designed and created by the Controlling Abnormal Network Dynamics using Optogenetics (CANDO) team and a conducted formal verification on the device. A formal specification was made based on a verification model created in VDM from source code given. Formal proofs were completed to show correctness of generated Proof Obligations from Overture based on the formal model. Of 190 proof obligations, 144 were shown to be correct while 3 were shown to have the same counter example. Many of these proofs demonstrated issues with the Overture proof obligation tool. In the original source code, some design errors were discovered such as uncalled states, erroneous transitions and lack of termination of the finite state machine (FSM) loop.

Keywords: Formal Methods, Medical Devices, Verification, Proof

1. Introduction

Increased complexity and commonality of medical devices in both personalized and clinical settings lead to serious challenges in reliability, personal safety and security [5]. With complex systems containing many subsystems, even high-level programmers will introduce logic errors into their code. Finding and eliminating these errors prevents propagating the code with design errors, this is paramount in safety-critical medical device software where lives can be at stake [4]. Closed-loop medical devices are life-critical and in a feedback loop with the organ they effect. The software works autonomously but needs to give confidence in its ability to work correctly [6]. Certification in the form of validation and verification is imperative [7].

Software in pacemakers and defibrillators often have 80,000-100,000 lines of code and it is estimated 10,000 implanted defibrillators are installed monthly in the US, taking the predicted number of implants in 2019 to be over 1 million [6]. It is essential to secure these implanted devices, as they can potentially put a patient in a life-endangering situation. Wireless communication and the internet provide benefits and issues for the millions of patients using medical devices worldwide due to increased dependence on software. The output of these devices can lead to doctors making life or death conclusions, so these readings need to be accurate and trustworthy [8].

Manufacturers of medical devices have rules they must abide by including pre-market and post-market conditions for selling. However, software is the main cause for recall in medical devices [5]. Most design schemes use simulation and other informal techniques to test the code they make [9] but software must be verified and validated [7]. To help programmers out it is necessary to give them the right tools. Formal methods can do this by forming a formal model of the system and using mathematically

proven methods to find logical errors [4]. Formal methods model normal system behavior and significantly increase security by detecting potential vulnerabilities [8]. Techniques like formal proof can show not just the presence of errors but the absence of logical errors [4].

In this project formal methods will be used to analyse, specify, verify and prove the source code of an epilepsy sensor which is designed to be implanted in a human brain on a closed-loop system. Providing this verification will increase the reliability, safety and security of the device and give confidence to the team who are manufacturing and designing the device.

2. Background

2.1 Medical Device to Verify

This project will look specifically at the finite state machine of an epilepsy sensor with code received from the CANDO project. According to the National Health Service (NHS) website epilepsy is described as a common condition affecting the brain which can start at any age. It is lifelong and causes seizures; bursts of electrical activity in the brain that can cause a wide range of symptoms including collapsing, uncontrollable jerking (a "fit"), losing awareness, strange sensations or becoming stiff. It is possible to live a life mostly unaffected with epilepsy but there are activities which an epileptic will need to consider their condition before partaking in; swimming, driving, some jobs and pregnancy to name a few [18].

Members from Newcastle University, Imperial College London and University College London are uniting together on the CANDO project to create a closed loop treatment for focal epilepsy using optogenetics. Optogenetics is a technique to control cells and neurons using light and would be beneficial for brain implants to treat epilepsy as other techniques are known to have harmful side-effects or are ineffective [1].

The CANDO project will target the "focus", the localised area of the brain that the abnormal activity spreads from, using a cortical implant with the aim of creating the first in-human trial of the device. Of over 600,000 people in the UK with focal epilepsy, one third are unlikely to respond to traditional drug treatments and require the removal of the focus via surgery. This can lead to irreversible damage to the brain with necessary functionality being removed. The implant is designed to modulate abnormal activity, preventing development of seizures. The implant stimulates at precisely the right timing due to brain waves being constantly monitored through electrodes. Brain cells are modified to be susceptible to light using a safe virus so that light sources can modify the electrodes when needed [19].

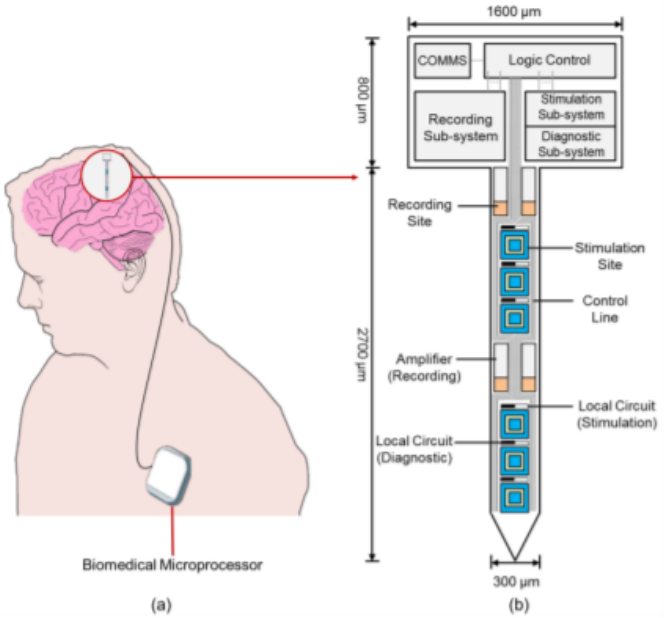

Figure 1: (a) Optrode with Communicating Chest Piece (b) Optrode Hardware [23]

Figure 1 shows the layout of the system when the optrode is implanted within the brain and using closed-loop processing communicates with a biomedical microprocessor as a chest piece the patient will wear. The optrode has various subsystems which communicate with one another and have different roles within the device. The optrode can complete onboard diagnostics, record local field potentials (LFPs) and control optical stimulations – this allows for real-time diagnostics during normal running of the device [23].

2.2 Aims and Objectives

The aim of the CANDO project is to produce the first in-human trial of a cortical implant using optogenetics to prevent the development of seizures. The aim of this project is to support the CANDO project with their aim by completing a formal verification of the command optrode interface. To complete the verification the following objectives will be tackled:

- Analysis of the source code to understand how it works
- Creation of a formal model which hold the properties of the source code
- Generating a formal specification for the given system
- Verifying the model against the specification and therefore the original system against it
- Conducting formal proof on the model to attempt proof of properties found in the specification

3. Implementation: Initial Analysis

3.1 Source Code Discussion

For this project two files from CANDO were acquired for analysis: Optrode_commands and Optrode_cmd_state_machine. Both files were written in the C programming language and refer to the optrode that would be implanted in the brain. The latter was mainly a finite state machine (FSM) which contained 33 rows and 21 columns. The whole FSM's state shall be referred to as the STATE with capital letters, this is the state of the entire system and environment which should not be confused with the 33 individual states which are within the FSM as the 33 rows are representative of each internal state and the FSM describes the transitions between those states. The columns represent the events which transform one state to another. For example, in figure 1 the state 0, when with the event CONT, transitions to state 1.

\Event State\	CONT	ERROR	SPI_TX_FINISH	SPI_RX_FINISH	LED_ON_E	SET_VLED_E	SET_BRE_E	LED_ALL_OFF_E	PROG_DELAY_DIAG	PROG_OP_MEM_1	RUN_MEM_E	PROG_CLK_CNT_E	RESET_ANA_E	SET_ANA_E	CONFIG_REC_E	PROG_ID_E	DUMMY_E	READ_LED	READ_DIAG	READ_LFP	GET_CMD
State 0	1	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 1	2	31	31	31	4	5	8	11	12	15	16	17	18	19	20	21	22	23	25	26	31
State 2	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 3	27	31	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 4	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 5	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 6	28	31	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 7	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 8	9	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 9	29	31	10	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 10	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 11	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 12	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 13	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 14	30	31	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 15	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 16	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 17	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 18	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 19	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 20	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 21	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 22	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 23	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 24	27	31	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 25	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 26	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 27	33	31	31	27	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 28	7	31	31	28	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 29	10	31	31	29	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 30	15	31	31	30	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 31	32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1
State 32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 33	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31

Figure 2: The Original Optrode_cmd_state_machine FSM (also Appendix A)

Each of the states from 0 to 33 has a name associated with it which was given in comments, but the C code does all transitions numerically and stores them in the STATE using their numbers. The FSM is a simple array of values so the headings for the events and states in the FSM are not included in the actual table but by code comments next to the respective column or row. The states presented in the left-hand column are displayed as it is in the code; each state is referred to by its number rather than its type in other parts of the system. The header file of Optrode_cmd_state_machine includes an enumeration of all the values associated with events stored in the STATE, this means the text version of an event can be used rather than a number, it is easier to understand this style of coding as it is more readable.

In Optrode_cmd_state_machine there is also one function called manual which runs the operations associated with the FSM. It contains a large loop which computes all transitions and calls all necessary functions which relate to the latest state in the STATE. These operations are all contained within the first file Optrode_commands. In

Optrode_commands there is an enumeration within the header file for each of the commands in the STATE. The operations inside Optrode_commands refer to all 33 states and each state has a name and an operation associated with it; see figure 2. Of the operations, the majority are very similar and do a packet manipulation before setting the current event of the STATE so that the next loop of the manual command will use the new event and not the old event.

There are some operations which are more unique. The get_cmd operation which contains a switch statement for each case of the different commands in the STATE. The send_packet operation which does a variant packet operation to other operations. The receive_packet operation which checks for a flag before receiving. The error operation which deals with errors that occur. The chip_rst operation (triggered through the error state) which resets various values before returning the GET_CMD event and the cmd_finish operation which sets the flag that the manual loop can terminate as the final FSM state transition has been made. A table of the names and operations associated with each state in the FSM is given in Appendix D.

These two C files combine to form all the data needed for state transitions inside the FSM. They include the original locations and destinations inside Optrode_cmd_state_machine and the methods to traverse these paths inside Optrode_commands. This inner system can be described fully and therefore a formal verification can be conducted. The files given were standalone so there was no formal specification to impose one will need to be formed.

Initial analysis of the source code produced multiple findings, firstly the common patterns in the system will be discussed, afterward potential anomalous values shall be discussed. The findings are based off the code provided but also the comments associated with these, for values considered to be erroneous the impact will be discussed which will also come from known information in the code or comments.

3.2 Initial Findings

Firstly, some events have a higher weighting than other events due to the number of states impacted varying dramatically. For every state under every event a transition does occur, but most of these transitions will be to the error state which can be equated to a default state or a default transition. Only the loop in the receive_packet operation calls the ERROR event which transitions everything to the error state. Since all state-event combinations default to the error state, the default case will be ignored in the analysis and deliberate transitions (all transitions not to error state or chip_rst state) will be looked at primarily.

The CONT event impacts the most states with only state 33 (cmd_finish) and state 32 (chip_rst) using the default transition under this event. The former state obviously transitioning to error in all event cases since the finished flag that was set would not have been detected. The error state only transitions away from the error state under GET_CMD, the only state this event causes a transition to occur in. chip_rst calls GET_CMD in the code but this transitions back to the error state under for this event.

The SPI_TX_FINISH and SPI_RX_FINISH events act in a similar way. The first event works with send_packet states and transitions from a send_packet state to the same send_packet state while the other event works with receive_packet states and once again transitions from a receive_packet state to the same receive packet state.

All other events only affect one state, but always the same state; the get_cmd state. The get_cmd state can transition to one of multiple states under different events.

Usually the transition will be to a state with the same name as the event that it has transitioned under. For example, state 1 (get_cmd) under event LED_ON_E goes to state 4 (LED_on) or event RUN_MEM_E foes to state 16 (run_mem). The exception to this is the event CONT which can be called on the get_cmd state and transitions to state 2 (LED_off).

3.3 Patterns

Inside the FSM there are 3 distinct transition patterns that can be seen, transitions are usually under the CONT event except for ones applied to get_cmd state since there are many different state transitions under the different events – these will be named more generally using x to represent the set of different events/states:

$$1.\quad 0 -> 1 -> x -> 3 -> 27 -> 33$$
$$Where\ x\ is\ in\ the\ set\ of\ states: \{2,4,11,12,16-22\}.$$

$$2.\quad 0 -> 1 -> x -> send\_packet ->$$
$$receive\_packet -> y -> 3 -> 27 -> 33$$
$$Where\ (x,y)\ can\ be\ (5,7)\ or\ (8,10)$$

$$3.\quad 0 -> 1 -> x -> 24 -> 27 -> 33$$
$$Where\ x\ is\ in\ the\ set\ of\ states: \{23,25,26\}$$

These three are varying forms of an even more general pattern that can be shown within the FSM. There are 5 states which are send_packet states (states 3, 6, 9, 14, 24) and 4 states that are receive_packet states (states 27, 28, 29, 30). State 0 (start), State 1(get_cmd) and State 33 (cmd_finish) appear in all three patterns in the same locations first two states and last state respectively. Since all send_packet states call the same operation and all receive_packet states call another, the pattern can be generalized so that pattern 1 and pattern 3 of the above are the same:

$$start \rightarrow get\_cmd \rightarrow x \rightarrow send\_packet \rightarrow receive\_packet \rightarrow cmd\_finish$$

Where x can now be any value within the set {2,4,11,12,16-23, 25,26}. The second pattern above can also be simplified in this way giving:

$$start \rightarrow get\_cmd \rightarrow$$
$$x \rightarrow send\_packet \rightarrow receive\_packet \rightarrow$$
$$y \rightarrow send\_packet \rightarrow receive\_packet \rightarrow$$
$$cmd\_finish$$

There is a clear loop of the operations where x and y both have similar send and receive state transitions before the cmd_finish state is reached.

The SPI_TX_FINISH and SPI_RX_FINISH have already been described as transitioning from a state to itself and only affect either the send or receive state. This can be shown in the following:

$$send\_packet \xrightarrow{SPI\_TX\_FINISH} send\_packet$$

$$receive\_packet \xrightarrow{SPI\_TX\_FINISH} receive\_packet$$

A diagram of the major transitions is included below as figure 2, it does not include state 31 (error) or state 32 (chip_rst) since the error state is the default for all state-event transitions not included in the diagram and chip_rst state is only called from the error state.

Figure 3: State Transition Diagram (also appendix E)

3.4 Potential Anomalies

Independent analysis of the FSM lead to corroboration of potential anomalies found in the source code. Some may not have any consequences while others could be harmful to the system if not detected. These anomalies could then be taken up with the CANDO design team and checked to see if there was an issue.

Firstly, state 0 (start) is never called except for the very initial run of the FSM loop, the operation for the start state only moves the current event to CONT to cause a transition to state 1 (get_cmd). The anomaly here is that it appears to not impact the code. Possible consequences arrive if the start state is supposed to be revisited as this is not possible in the code.

State 13 (prog_op_mem_1) is never called in the FSM table – no state-event pairs transition to it. This triggers large consequences as states which are only visited by state 13 are then themselves never reached. State 14 (send_packet) and state 30 (receive_packet) only are called through state 13 and so become unreachable themselves. The desired method of prog_op_mem_1 will not occur; the update of the MEMSize register in the Program Memory state. This method determines the length of the memory to be programmed.

State 16 (run_mem) relies on prog_op_mem_1 for updating values. Run_mem uses both prog_op_mem_1 and prog_op_mem_2 states (states 13 and 15 respectively) to operate. By missing the data from state 13 the method can run but could use incorrect data leading to errors down the line – this could go undetected.

The solution to State 13 not being called is possible to deduce from the patterns described in section 4.2. The second pattern described fits the flow of states 13, 14, and 30 to state 15:

$$x \rightarrow send\_state \rightarrow receive\_state -> y$$

Where state 14 is the send_state and state 30 the receive_state. It also appears that state 15 is the x value in its own transition flow as state 1 (get_cmd) under PROG_OP_MEM_E transitions to state 15 (prog_op_mem_2) directly. By switching the transition to go to state 13 (prog_op_mem_1) from state 1 then prog_op_mem_1 will be reachable and state 15 will slot into the second pattern from section 3.3 as described before.

State 1 (get_cmd) is only reached by state 0 (start) and state 31 (error). The consequences of this are that to complete two operations the program must fully finish executing, then exit and restart to do the second operation, in case of an error the first operation will be repeated and can't be changed mid-run. The chip_rst operation calls the GET_CMD event but the FSM table shows this transition going to the error state. From the error state the get_cmd state can be reached through the GET_CMD event.

The Optrode_commands file mentions an idle state at state 1 in the comments of the code, however this does not exist as state 1 is the get_cmd state. On top of this it describes the idle state being entered once transitions and operations have ended and until a new command is requested. This does not fit with the code as we discussed state 1 (get_cmd) can only be reach through the start state or error state, neither of which can occur when the operation has ended.

In the FSM SPI_TX_FINISH appears as an event which causes a self-loop transition. It is called on send_packet states but for state 9 (send_packet) the event transition goes to state 10, which is not a self-loop. Since all other send_packets under SPI_TX_FINISH loop back on themselves this is likely to be an erroneous value. Related to this the SPI_TX_FINISH event is never called within Optrode_commands the consequences of this are potentially minor as the FSM can still finish and the event triggers a self-loop so the transition path would remain almost the same. However, the functionality associated with the transition will be missing. SPI_TX_FINISH is linked with send_packets as they are the only states which transition normally under the event. However, the send_packets operation does not call an event it only completes packet operations. The consequences of this could be that the event is not updated as it should be leading to errors. The lack of event in the send_packet operation and the SPI_TX_FINISH event not being called in the code were later reasoned as being inside code that was inaccessible for this project. The issue mentioned of the incorrect transition between state 9 and 10 however is relevant. Since the FSM can still transition despite the internal fault that occurs it would be difficult to detect during runtime.

The operation get_cmd is a large switch statement with cases for each of the potential commands to be held by the STATE. The last case on the switch statement is a default case which calls break and no other operations. This seems anomalous since if a command is called which does not appear within the STATE then an error should be thrown, by calling break the previous stored event will still be set and the program will continue transitioning as before except it is likely to take transitions which were

not desired. The two scenarios which can occur are that the command is wrong from the beginning of the run and so the event CONT will remain stored from the transition to get_cmd from start. This will then lead to state 2 (LED_off) even if another event was intended but the command was wrong. The other scenario is the get_cmd state is accessed from the error state under the GET_CMD event. This would mean the GET_CMD event remained which leads to get_cmd under event GET_CMD which goes back to the error state causing a possible infinite loop.

Finally, there is a possible single point of failure where state 33 (cmd_finish) can only be reached by state 27 (receive_packet) under the CONT event. Any loops or halts in the program which do not lead to state 27 and then the cmd_finish state will prevent the FSM resolving safely. The program will then require the FSM to finish after a timeout operation.

After discussion with the CANDO team most anomalies could be explained from a different area of the code that this project didn't have access too. State 13 and state 9 errors were genuine issues. Although it is possible the major errors were just typos, the analysis presented on the source code led to the logic behind the values being understood and therefore the illumination of those typos. Figure 4 below shows the FSM but highlighting the positions which appear to hold an incorrect value and then also in red text marking the states which are not accessed. These were the major issues found within the code and were corrected as explained previously, the FSM with all data corrected can be found in Appendix C.

\Event State\	CONT	ERROR	SPI_TX_FINISH	SPI_RX_FINISH	LED_ON_E	SET_VLED_E	SET_BRE_E	LED_ALL_OFF_E	PROG_DEL_AY_DIAG_	PROG_OP_MEM_1	RUN_MEM_E	PROG_CLK_CNT_E	RESET_ANA_E	SET_ANA_E	CONFIG_REC_E	PROG_ID_E	DUMMY_E	READ_LED	READ_DIAG	READ_LFP	GET_CMD
State 0	1	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 1	2	31	31	31	4	5	8	11	12	15	16	17	18	19	20	21	22	23	25	26	31
State 2	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 3	27	31	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 4	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 5	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 6	28	31	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 7	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 8	9	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 9	29	31	10	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 10	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 11	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 12	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 13	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 14	30	31	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 15	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 16	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 17	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 18	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 19	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 20	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 21	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 22	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 23	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 24	27	31	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 25	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 26	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 27	33	31	31	27	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 28	7	31	31	28	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 29	10	31	31	29	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 30	15	31	31	30	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 31	32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1
State 32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 33	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31

Figure 4: The Optrode_cmd_state_machine FSM with erroneous values marked (also Appendix B)

4. Implementation: VDM Model

4.1 Why VDM? What is VDM?

VDM is a tool designed to model computer-based systems using formal methods, an acronym for the Vienna Development Method. Used mainly in the development stage before expensive implementation commitments are made, VDM allows for the analysis

and creation of early stage design models. Safety and security properties are verified within a VDM model, but incompleteness and ambiguity of a design are also made apparent to a designer. VDM is used in industry and is thought to reduce or eradicate costs caused by reworking logical errors discovered late in a design process by exposing them earlier on [20].

A technique used in VDM is abstraction; to abstract away from the fine details of a system and instead model the general properties which are underlying it. A VDM model has the STATE (like mentioned previously – capital letter STATE includes the current state, finish flag, current command, etc.) which is the environment or playing field for our model. The STATE can be constrained using invariants (rules applied to values to enforce properties onto the model). Invariants will throw errors whenever they are broken – and must always be held within the system. Within invariants there are a subset known as pre and post conditions. A precondition is like an invariant since it is also a rule, but it has difference that it only needs to hold before an action takes place – in the same way a postcondition is an invariant which only needs to hold after an action takes place. Pre and postconditions are used to make sure operations work as planned with only the expected values being put in and the expected outcomes pushed out.

```
Bytes = nat          -- Bytes sent or received
  inv b == b <= PACKET_LENGTH;
```

Above is an example invariant, for the created type Bytes. It must always be true that Bytes never holds a value greater than the value of the constant "PACKET_LENGTH". Below is an example of a precondition and a postcondition. The precondition states that the current state must be LED_off before the operation LED_off can be executed while the postcondition states that after the operation the current event must be the CONT event.

```
LED_off() ==          -- state 2
    (
      s_packet := mk_Packet(<Optrode_addr>, <LED_OFF_C>, <LED_addr>); --assembles packet
      currentEvt := <CONT>;
    )
  ext
    rd currentSt
    wr currentEvt, s_packet
  pre currentSt = <LED_off>
  post currentEvt = <CONT>;
```

As well as the STATE which tracks all the current values of the environment, there is an initial STATE which is the STATE that the model will begin in when it is run the first time. For this project the initial STATE will include that state 0 (start) must be the current state when the program begins and the first event to be called must be CONT since this is the only event that can be called on state 0, the finish flag should start as false and not change until the FSM program safely completes.

4.2 Designing the Model

The initial decision taken was how to design the FSM which all the code was based upon. It was within the FSM that the properties of the model would be located so finding a way to tap into these properties was essential. At first it was proposed that a

map of state to event to state would be best since this was the format given within the source code. However, this technique struggled against the map of event to state to state which could enforce properties based around the events. Submaps such as State Maps could be formed with their own properties and then strengthened by events and event only properties when used in the complete map (Total State Map) where every permutation of event and state was included in the FSM.

Using smaller functions to build up the Total State Map, a default submap could be inserted as an umbrella transition from a state to the error state. covering all transitions not otherwise directly described. This meant it was possible to have State Maps with specific properties being included within the Total State Map by design before all remaining default transitions were added in bulk. By using this building block style technique, invariants can be imposed on specific sections of code in a general way rather than relying on a complex invariant being imposed on a very general FSM which would have been the case with mapping state to event to state.

Abstraction was key in the design of the system. Within the source code lots of fine details are included, such as the data used in the packets and the number format of tracking states. This can make it difficult to work out which transition is being checked which is more likely to lead to errors. To deal with this in the model an enumeration of states, events and commands was made so that the name of the event, state or command could be used and abstract away from the underlying value associated with it. With regards to the packets, abstraction was used to give general names as a summary of sections and values within the packet.

$$packet = ((((Optrode0.Optrode\_addr \,\&\, BITS\_6) << 18) \mid ((LED\_OFF \,\&\, BITS\_6) << 12) \mid (Optrode0.LED\_addr \,\&\, BITS\_5)) \,\&\, BITS\_24);$$

Could now be written as:

```
s_packet := mk_Packet(<Optrode_addr>, <LED_OFF_C>, <LED_addr>); --assembles packet
```

This is a much simpler way of portraying the same data, keeping the properties but stepping away from the fine details of the code.

Within the states it is also possible to distinguish similar states from one another by separating them into groups. Such group were formed consisting of send states, receive states, packet creator states (any state which can be substituted for x or y in section 4.2 about patterns in the FSM), and error states (error and chip_rst). Packet creator states can be further broken into two subsets. Stage 1 packet creator states are ones which can be substituted for x in the pattern and transition directly from the get_cmd state while stage 2 packet creator states transition from receive packet states to complete the second loop which some transitions undergo with a double send and receive occurring before the transition to cmd_finish.

The pattern of the start state transitioning to the get_cmd state which transitions to a packet creator state onto a send packet state before a receive packet state is very useful in the modelling. Patterns are simple to impose as they apply generally over a system and will occur on multiple occasions. The below two patterns will be imposed in the model as they have been deduced and refined from the source code (also section 3.3):

$$start \rightarrow packet\ creator\ state \rightarrow send\ packet \rightarrow receive\ packet \rightarrow cmd\ finish$$

$$start \rightarrow stage\ one\ PCS \rightarrow send\ packet \rightarrow receive\ packet \rightarrow stage\ two\ PCS \rightarrow send\ packet \rightarrow receive\ packet \rightarrow cmd\ finish$$

Further design decisions, ones relating to the formal specification of the system are described in the next sections.

4.3 State Maps

A State Map is a map from state to state. The most basic form a State Map has eight invariants:

1. No states can map to the start state
2. The start state can only map to get_cmd or error states
3. The error state can only map to error, get_cmd or chip_rst states
4. The cmd_finish state can only map to the error state
5. The chip_rst state can only map to the error state
6. All Packet creator states can only map to send states or error.
7. All Receive states map to either a stage two packet creator state or cmd_finish state
8. The state get_cmd maps to either the error state or a stage one packet creator state only

All forms of the State Map must hold to the above eight invariants, the State Map is the primary building blocks of the FSM and other maps in this model; the code for them

```
StateMap = map State to State
      inv s ==
            --@doc states cannot map to the initial state "Start"
            not <start> in set rng s
            and
            --@doc start state can only map to <get_cmd> or <error>
            (<start> in set dom s => s(<start>) in set { <get_cmd>, <error_> })
            and
            --@doc error state can only go to error_, get_cmd, or chip_rst
            (<error_> in set dom s => s(<error_>) in set { <get_cmd>,
            <chip_rst>, <error_>})
            and
            --@doc cmd_finish can only lead to error
            (<cmd_finish> in set dom s => s(<cmd_finish>) = <error_>)
            and
            --@doc chip_reset can only lead to error
            (<chip_rst> in set dom s => s(<chip_rst>) = <error_>)
            and
            --@doc packet_creator_states can only go to send packet states or
            error, prevents packets being overwritten
            (forall p in set packet_creator_states & p in set dom s => s(p) in
            set ({<error_>} union send_states))
            and
            --receive states map to either a stage 2 packet creator state or the
            finish state
             (forall r in set receive_states & r in set dom s => s(r) in set
            {<cmd_finish>, r, <error_>} union stage_two_packet_creator_states)
            and
            --@doc get_cmd maps to either error or a stage 1 packet state
            --@doc get_cmd cannot lead to either send or receive states or stage
            (<get_cmd> in set dom s =>
                        s(<get_cmd>) in set ({<error_>} union
```

Figure 5: State Map Invariants

can be seen in Figure 5. The eight invariants are taken from the FSM directly by analysis of the rows and highlighting states which have minimal transitions or can be generalized simply.

A Transition Map is a form of State Map, so contains the same primary invariants, but imposes an additional invariant that it only maps send states to receive states. If any state within the domain of the Transition Map is not in the set send states an error is thrown, the same is true if the range of the Transition Map contains a state other than from the set receive states. Figure 5 shows the VDM code for these invariants.

```
--@doc transmission map ensures that send_states are mapped only to receive_states, if any
TXMap = StateMap
inv m ==
  dom m subset send_states
  and
  rng m subset receive_states;
```

Figure 6: Transition Map Invariant

An Identity Map is another form of State Map. In an Identity Map all states map to themselves. This will be a useful map to have as it specifies what occurs in events SPI_TX_FINISH and SPI_RX_FINISH. The VDM code is shown in Figure 6.

```
--@doc identity state map
IdMap = StateMap
inv m ==
  forall s in set dom m & m(s) = s;
```

Figure 7: Identity Map Invariant

An Error Map is a State Map where all states map to the error state. This will be used for the error event but also can be used to describe the default case to be imposed.

```
--@doc maps every state to <error_>
ErrorMap = StateMap
inv em ==
  forall s in set dom em & em(s) = <error_>;
```

Figure 8: Error Map Invariant

A Packet Map is a State Map where every packet creator state maps to a send state. This maps the transition in the pattern of the FSM with packet creator states transitioning to send states.

```
--@doc maps every packet creator state to a send state so no packets are overwritten
PacketMap = StateMap
inv pm ==
  dom pm subset packet_creator_states
  and
  rng pm subset send_states;
```

Figure 9: Packet Map Invariant

A Receive Map is a State Map where the domain consists only of receive states and they map either to the stage two packet creator states or the cmd_finish state.

```
--@doc maps every receive state maps to a stage 2 packet state or the command finish state
ReceiveMap = StateMap
inv rm ==
  dom rm subset receive_states
  and
  rng rm subset stage_two_packet_creator_states union {<cmd_finish>};
```

Figure 10: Receive Map Invariant

A Total State Map (TSM) is a State Map, however, the domain of the State Map must contain all states. This is essential otherwise the FSM will have gaps in the data table which would not be accurately modelling the source code.

```
--@doc total statemap (or FSM row) must contain all states in the domain of the set
TStateMap = StateMap
inv sm == dom sm = ALL_STATES;
```

Figure 11: Total State Map Invariant

A simple function, sm2tsm, was created to convert a State Map to a Total State Map by the union of all map forms together and any remaining states not currently in the domain are mapped to the error state. Sm2tsm is crucial to the creation of the Total FSM (TFSM) which will be analysed in this project since otherwise there would be missing information in the modelled FSM table which could lead to errors and logical conclusions being formed which are false.

4.4 CandoFSM

The aim of the building blocks formed in section 4.3 was to build up to the FSM that is to be analysed in the source code. This FSM shall be named CandoFSM so that FSM can be used to more generally refer to all mappings of event to state to state. Since a map of state to state has been created the definition of an FSM can be simplified to a map of event to Total State Map. All the invariants of the Total State Map are included within the FSM definition now.

A Total FSM is a type of FSM which has the following invariant; A TFSM must contain all events in its domain and for every event in the domain that event must map to a Total State Map. Therefore, a TFSM can be used to specify the entirety of CandoFSM.

CandoFSM is a TFSM with invariants. CandoFSM comprises of all the building blocks that have come before it and can impose invariants which use these blocks to specify properties which must be held within the code. The following are all properties held within the FSM of the source code and use the building blocks to create succinct invariants – event specific ones particularly:

1. Under event CONT, all packet creator states are a Packet Map
2. Under event CONT, all send states are a Transition Map
3. Under event CONT, all receive states are a Receive Map
4. Under event SPI_TX_FINISH, all send states are an Identity Map
5. Under event SPI_RX_FINISH, all receive states are an Identity Map
6. Under event CONT, all error states are an Error Map
7. The start state under event CONT transitions to the get_cmd state. Under all other events the start state will transition to error.
8. The error state under event CONT transitions to the chip_rst state

9. The error state under event GET_CMD transitions to the get_cmd state
10. Under all other events the error state transitions to the error state (this is both an Identity Map and an Error Map!)

4.5 Operations

For all operations in the source code, the process of modelling is straightforward. The abstraction of the packet types simplifies the operations. Complex packet calculations are no longer needed. The main additions to the operations within the code are the preconditions and postconditions. For the operations the precondition check will be that the operation is being called only when the current state in the STATE matches correctly with what is expected. It is not wanted that a send state can activate the get_cmd operation for example. The postcondition considers more the event which will affect the next transition and checks the current event in the STATE has been updated appropriately by the operation.

On top of this, VDM provides further checks which can be imposed on the operations. Operations change variables inside the STATE some are only read, others written and so it is possible to specify the variables which the operation can write over and those which can only be read. Therefore, if the operation attempts to write over a variable in the STATE which it has not been given the permissions then an error will be thrown like the invariants. These hardcoded limitations are nearly identical to the actual C code, so a trace can occur to see how the program runs. The manual function was also modelled across, so it is possible to run various simulations of the model and test the properties or values within the model.

4.6 Formal Specification and Verification

The invariants discussed in the previous two sections make up the formal specification for this model. These are the rules which the system abides by and which can be formally verified to determine if they hold within the model or they do not! VDM provides the option of this as it can run the model and any cases where invariants, preconditions or postconditions fail the specification can be shown not to verify correctly.

Running the model with the initial data from the source code FSM (appendix A) throws errors immediately. This is because the initial FSM to be used in the STATE contains values which trigger the invariants of CandoFSM to throw errors. This is to be expected as state 9 (send_packet) does not follow the Identity Map that it should. For state 15 (prog_op_mem_2) it is currently classified as a stage two packet creator state and so would trigger an invariant error in the original data since get_cmd state can only transition to stage one packet creator states. If it had also been misclassified as a stage one packet creator state, then state 15 would still throw an error as state 30 (receive_packet) tries to transition to it. Receive Packets only transition to stage two packet creator states or cmd_finish.

For the data corrected after the erroneous values were found (appendix C), the model runs and can complete a trace of the system – the most recent trace run for this project completed in 0.09 seconds with no issues. VDM can verify if a specification is imposed on a system or not, but it cannot prove the absence of logical errors – for that formal proof would need to be done.

5. Implementation: Isabelle Model and Proof

5.1 Conversion to Isabelle

Isabelle is a proof assistant tool which can help with formal verification proving correctness of computer software or hardware. Mathematical formulae are expressed in formal language which allows for proof in logical calculus [22]. Therefore, principles in a VDM model can be shown correct through mathematical proof using logical calculus. The formal specification which was created and imposed on the VDM model can be tested using proof to see if it always holds and prove that fact. In order to do this the VDM model will need to be converted to Isabelle and the new Isabelle model proved.

Converting to Isabelle is a tricky task, for one example map comprehension in VDM uses sets but for map comprehension in Isabelle lists are better since they do not require as complex proofs down the line. Other difficulties come with syntax – Isabelle syntax requires very specific uses of brackets for some parts of code and missing those brackets can make the meaning completely change while the same line of code in VDM is more readable and less precise syntax dependent. Converting the VDM to Isabelle was a slow process in this project as Isabelle was an unfamiliar language but in the end the task was fully completed. An existing toolkit "VDMToolkit" was considered which specified some VDM in an Isabelle format such as VDMNat which describes the natural numbers that VDM uses in Isabelle Syntax or VDMMap which does the same for maps. In the end these were mostly done away with but having a basis to use and a knowledge bank to rely on was helpful.

5.2 VDM Proof Obligations

For a VDM model which has been fully specified it is possible to generate proof obligations (POs) in Overture, leading to a list of conjectures which can show properties will always hold for a system if true. For the VDM model created in this project the number of proof obligations was 190. One conjecture, proof obligation 187, was unprovable from the beginning since the proof given was "…", it is assumed this meant that the proof was either too complex or too long to be printed with the rest of the POs. Another group which appear to be poorly specified proofs occur from the execute command within the VDM. There are 33 proof obligations for the execute command, but the proofs appear to not be complete as they have obvious counter examples. Proof Obligation 153 asks for a proof to this:

$$currentSt = < start >$$

There are obvious counter examples to this such that currentSt = <get_cmd> or any of the other states. On further inspection it could be seen that the above is the precondition for the start operation. The next proof obligation was the post condition for the start operation and the proofs continue like that. Underlying these conjectures is the question of it being true when the operation is called. This is not something easy to describe. These 33 proofs are difficult to interpret and so they have been skipped from the proofs; this left 156 proof obligations.

Of the 156 proof obligations remaining in this project 154 were successfully translated to lemmas which could then have a proof attempt made of them. The two proof obligations which were not translated both use the map override function which

was difficult to translate as the symbol required for the translation was unknown and no information could be found online about it.

82 of the proof obligations were discovered to be identical to one another. They were each called independently inside each of the operations and were referring to the STATE invariant. Since these 82 could be considered as one proof then the number of proofs required to translate was 75 (including the one proof and the two untranslated). Of these 75; 73 were translated correctly and 63 were proved to be true or true with respect to another lemma including the 82 in one lemma. Nine were translated but not proven and three were shown to include a counter example.

5.3 Flaws in Overture

From what has been discussed in section 8.1. It becomes clear that the Overture proof obligation tool is flawed. On the one hand there are proofs which cannot be completed due to poor specification this leads to spurious proofs and on the other hand proofs which are solved trivially where the Isabelle tool can complete them automatically and not much is learned of the previous model. For the spurious proofs where the proof appears incomplete or is written in a way that poorly specifies the property that is trying to be proved the tool leaves a user lost for understanding on that area of their program. If the proof cannot be understood and so cannot be completed the user does not know if the property being tested holds in the model or if it fails. This could lead to issues further along the development cycle of the program since that area of code has not been verified properly and any errors will likely go unnoticed.

For the trivial proofs the property being proved appears irrelevant, providing no meaningful information. Some proof obligations are incredibly complicated and specific, and a user cannot learn about the model or the system they are looking at from it. In the program these also appear in groups where each proof is the same as the last proof with a slight addition to the end of it. This creates chains of proof where one proof can prove the next one down the chain. These proofs appear to be a trace through the system attempting each one in turn, but a better definition could merge many of these proofs into one general proof which shows the properties of all the traces. This is assumed to be a design error of the POs in not providing a well-defined scope of the proof. In Figure 13 below the proof being derived appears to be a long complex trace through the FSM without providing the underlying property needing to be proved – with around 10 other nearly identical proofs adding a little suffix to the proof they could be simplified into one proof. In Figure 14, the opposite is shown in that the PO is meaningless since it proves that true is true. Many of the proofs for the operations and functions appear to show properties like this where there is no real point creating a proof. In 14, true will always be true no matter was you let x be within it. Neither of these really capture a proof which a user would find interesting and learn a large amount from. As a tool Overture could improve its features by refining the proof obligations.

```
(forall fsm:FSM3`TFSM & (is_((send_states <: fsm(<CONT>)), TXMap) => (is_((send_states <: fsm(<SPI_TX_FINISH>)), IdMap) => (is_((receive_states <:
fsm(<SPI_RX_FINISH>)), IdMap) => (is_((packet_creator_states <: fsm(<CONT>)), PacketMap) => (is_((error_states <: (packet_creator_states <-: fsm(<CONT>))), ErrorMap)
=> ((fsm(<CONT>)(<start>) = <get_cmd>) => ((forall x in set (dom ({<CONT>} <-: fsm)) & (fsm(x)(<start>) = <error_>)) => ((fsm(<CONT>)(<error_>) = <chip_rst>) =>
(<GET_CMD_E> in set (dom fsm)))))))))))
```

Figure 12: A Complex Proof Obligation that Could be Generalised

```
let mk_FSM3(-, s, -, -, -, -, p, -, -, -) = FSM3 in true
```

Figure 13: A Meaninglessly Trivial Proof Obligation

Due to the trivial nature of some of the proofs like Figure 14 (the repeated proof obligation), it seemed clear that some POs were attempting to touch on an important proof but got it wrong. This would imply that there are still some POs missing that Overture's system has not declared. This could be misleading if all proofs were shown to be true as an unevaluated proof could notify of a problem within the model.

However, it is still possible to prove some key properties from the proof obligations and to learn meaningful information from them. Operations and functions appear to struggle more when proof obligations are formed. The types, if well-defined, show proof obligations being parallel to the invariants of those types. It is then possible to attempt a proof that a type should not break it's invariant by its definition. This is a useful proof to have evaluated.

5.4 Overview of Translation and Proof

Of the 190 proof obligations 144 were proven to be true or true with respect to another lemma. With respect to another lemma refers to the lemmas which build on one another. In Figure 15 below lemma 22 has been proved with respect to lemma 22d. However, lemma 22d has a nitpick counter example – when the FSM is an empty map the lemma does not hold. This is due to the specification and invariants missing that it is possible for the TFSM to accept a map which is empty or an error in the code which doesn't close of this loophole from appearing. If lemma 22d was true, then lemma 22 would also be true. In fact, this would then mean lemma 23 was true and lemma 24 was true etc. When using the term with respect to – it is equivalent to saying under the assumption that the other lemma is proven true. By using the term "sorry" in lemma 22d, the program reads lemma 22d and treats it as if it was true even though it has not been proved.

```
lemma PO_22d:
  "∀ fsm::TFSM. SPI_TX_FINISH ∈ dom fsm"
  nitpick
  sorry

lemma PO_22:
  "∀ fsm::TFSM . inv_TXMap (send_states ◁ the (fsm CONT))
     ⟹ ∀ fsm::TFSM. SPI_TX_FINISH ∈ dom fsm"
  using PO_22d
  apply auto
  done
```

Figure 14: Example Proof with Respect to Another Lemma

This style of proof was powerful when it came to the chains of proof discussed before. Although a single more general proof would be more practical by generalising

the whole proofs to show that they are all true. However, if the first proof is false it is then also relevant that all the chain proofs are also false.

6.5 Specification Properties Proven in Isabelle

The POs state which section of the code they come from and therefore a whole section can be proven to be true. For example, all the properties of the State Map have been proven to be true as all eight proof obligations regarding the State Map definition have been translated and proven correctly. The same is true for the definition of Transition Map, Identity Map, Error Map, Packet Map, Receive Map, and Total State Map. For the section Total Finite State Machine there were two proof obligations the first was proved simply while the second has been translated but not proven. This means it could be proved to be true or a counter example can be found, but either result is currently unknown.

By proving the definitions of the types State Map and the other Maps a strong foundation can be built for the design of the remaining program. Knowing the building blocks do not have design faults, and have been proven to not have any, shows great promise for the functions and operations that use them. Any errors that occur must have come from the definitions of functions or operations themselves and not errors in the setup of the program structure or the FSM data.

Other areas shown to be true were large parts of the sm2tsm function and the fsm2tfsm function. For both functions two thirds of the lemmas were proven to be true while the remaining lemma for each was translated but not proven. The general FSM was in a similar boat with most properties being proven true and the last property needing proof but being translated and ready for proof.

For all the state invariants on the operations the same lemma was produced and shown to be true. The problem with this lemma is that it was trivial and requested a proof that x in true was true. Anything in true will be true so this proof was simplistic and there is probably an underlying proof missing that this proof attempted to touch on but got wrong. If this PO was accurate and it had been proved that would be a good state for the model to be in with many of the properties proved mathematically so that assurances can be given to the reliability of the program. Currently most POs are shown to be true but the reliability of this is not as high as would be ideal.

6.6 Counter Examples and Unfinished Proofs

It was discussed that 3 counter examples were found within the proofs of the model. Some of these were at the start of a chain of proofs and so were the first domino to fall causing others to fall also. If these cannot be rectified, then the issue that the proof displays will propagate through the model due to the chain like properties linking areas of code. If they can be rectified the counter examples have been beneficial since they have corrected the model to be more accurate than it was before and to catch faults which the model previously would have missed. Even if unrectified the knowledge of the potential error is valuable.

The proof obligations which give counter examples were POs 21, 42 and 49. The first appeared in the CandoFSM type section where a proof to check that CONT appears in the domain of all TFSMs is attempted. Using the nitpick tool to find counter examples one is given when the TFSM is an empty map since an empty TFSM cannot contain any events so CONT cannot be in the empty map. In practical terms the TFSM will likely never run while being empty however since there is no prevention to stop

this occurring the proof remains untrue. To correct the mistake the invariant of the model can be changed to enforce that the TFSM cannot be the empty map. The issue of the TFSM or FSM being an empty map is the counter example which is given for all 3 cases where the PO was shown not to be true. This should be an easy fix because if the program will break when the map is empty then the fix is to prevent to program allowing for the map to be empty in the first place using the invariants of the model. At the end when the invariants of the model are coded back into the source code then the program will be much safer to run.

There are some potential issues that occurred with these counter examples, mainly that the program did specify some form of prevention to the counter example in the invariant's definition. The definition of Total FSM or Total State Map states they must contain all the events and the states. In theory this means the same as not being empty but to show that to the Isabelle proving language is trickier than anticipated. This is likely due to a lack of experience with the language to that depth and something to work on in the future.

Referring now to the proofs which are still unfinished the lack of experience just described is a stumbling block with completing them. Isabelle has thousands of proofs that can be used to help with proofs made by a user however determining which ones to use, finding beneficial ones and not being trapped in rabbit holes is a scary task. Most proven lemmas came from the combination of a few Isabelle commands; applying simp or auto which solved trivial proofs immediately, nitpick for counter examples and sledgehammer to do an automated search for a proof over a period and then return a full proof if successful. Harder proofs like the ones unfinished require manual work and larger experiences levels than currently are available.

6. Implementation: SPIN Model

6.1 Timeouts not as they Seem?

In [1] Pollitt analysed the same source code being used in this project as well as addition code which he was given access to. From his findings the termination of the loop came into question. Other areas of the code require the FSM loop so it is essential that it can terminate so the resource can be shared. Until the cmd_finish state is entered, and the command finish flag is set to true the FSM loop will iterate through the states in the FSM based on the current state and current event. The only way to reach the cmd_finish state is through receive_packet state 27. This is a bottleneck only allowing for one safe termination of the loop through that route.

Should an error be thrown the error state allows three repeat attempts at the command before completing a chip reset which restarts the FSM loop and the command is attempted once more. It is possible for the next attempt to once again have errors and the above process occur again, so a chip reset happens once more. In fact, this loop can potentially occur infinitely unless an external stimulus such as a timeout triggers the FSM loop to exit, or the program exits safely.

Pollitt's analysis showed the implementation of timeouts currently occurs on each state and causes the FSM to transition to the error state within the FSM loop. This means the timeout imposed on the system does not do it's intended job of exiting the loop. The timeouts do not end the execution of the loop and so potentially it is possible for a loop to continue indefinitely. If the program can be proved to terminate eventually then the timeout is an inconvenience but not fatal, should the proof for termination of

the loop fail and provide a counter example it would be possible for the program to be trapped in an infinite loop.

6.2 What is SPIN?

SPIN is an open-source verification tool with high popularity, it is used for four main purposes. The first as a simulator, the second as an exhaustive verifier checking the validity of correctness requirements in a system, thirdly as a proof approximation system for large models and finally a driver for swarm verification. As a simulator and exhaustive verifier, it is useful for this project since a model can be made and tested for correctness – in the case of this project loop termination. A feature of SPIN are labels which act as flags. Two of note are the progress label and the end label [21].

SPIN can show non-progress cycles within a model it is given. A non-progress cycle is a loop which may never terminate, effectively it can show quite simply that termination cannot be proved. Progress labels are used to check that a part of code is visited within a run of a program. In a program that runs infinitely the labelled state should be executed infinitely. End labels are used to show correct termination of a program. If a program terminates while not in an end label state, then the program throws an invalid end state error [21].

6.3 Proving Termination

The model being used for the SPIN analysis is the generic pattern that summarises the FSM transitions.

$$start \rightarrow packet\ creator\ state \rightarrow send\ packet \rightarrow receive\ packet \rightarrow cmd\ finish$$

The above transitions summarise this model since the stage one and stage two packet creator states version is equivalent to a loop back through from packet creator state to send packet onto receive packet before reaching the cmd_finish state. This is the shortest path to the successful termination of the program. However, all these states can also transition to the error state and after 3 attempts the error state will cause the chip reset which uses the chip_rst state. A progress label will be added at the transition from the receive_packet state to the cmd_finish state. This will check that there is not a non-progress cycle in the code before this transition and therefore that the program can terminate. Should termination be proven for this model then a larger model of the system will be created and tested for termination until the complete system is shown to terminate. If this model is shown to have a non-progress cycle however that will mean that the FSM loop cannot prove termination. The cmd_finish state will have an end label so that correct termination can also be checked with no invalid end states should it terminate. A diagram of the states and possible transitions is given in Figure 11.

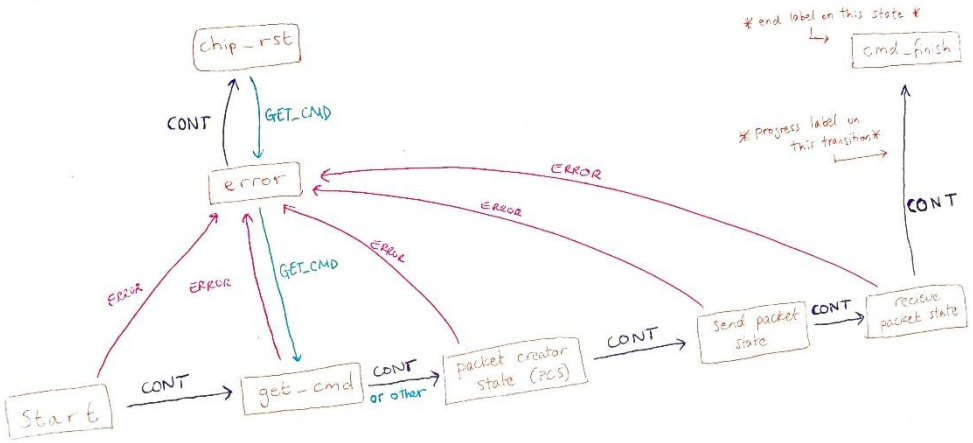

Figure 15: Diagram of States and Transitions for Termination Proof (also appendix F)

SPIN uses messages to move between states. This technique will be adapted for the SPIN model – one message will be passed around all the states. This will represent the transitions around the FSM loop. The likelihood of a state transitioning to the error state or choosing the safer route to the cmd_finish state will be weighted equally. The model works under the assumption that all messages sent will arrive without timeout, that all data unrelated to transitions will be abstracted away from including all SPI Identity Map transitions.

The results of running the test can be seen in Figure 12. In particular it is clear that a non-progress cycle was found at depth 26. This means that it is possible for an infinite loop to occur inside this simplified part of the model and hence complete CandoFSM. Since there was a non-progress cycle termination cannot be proved for this loop and a lack of timeout method to exit the loop means the code could be trapped forever in a cycle.

```
spin -a  epilepsy_optrode_draft1.pml
gcc -DMEMLIM=1024 -O2 -DNP -DNOCLAIM -w -o pan pan.c
./pan -m10000  -l
Pid: 6824
pan:1: non-progress cycle (at depth 26)
pan: wrote epilepsy_optrode_draft1.pml.trail

(Spin Version 6.4.4 -- 1 November 2015)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim         + (:np_:)
        assertion violations + (if within scope of claim)
        non-progress cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)

State-vector 236 byte, depth reached 5657, errors: 1
    3597 states, stored (4365 visited)
     256 states, matched
    4621 transitions (= visited+matched)
       0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
   0.906   equivalent memory usage for states (stored*(State-vector + overhead))
   0.769   actual memory usage for states (compression: 84.92%)
           state-vector as stored = 196 byte + 28 byte overhead
 128.000 memory used for hash table (-w24)
   0.534   memory used for DFS stack (-m10000)
 129.218 total actual memory usage



pan: elapsed time 2.93 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Figure 16: SPIN Verification Summary - Orange Line Shows Non-Progress Cycle

7. Evaluation and Conclusion

For this project there were 5 aims and objectives – namely to complete a full formal verification with proof of the epilepsy optrode which was being analysed and the breakdown of the steps needed to complete that final goal. The use of this would be the support of the CANDO team with their aim of the first in-human trial of the cortical implant using optogenetics.

The outcome of this project has been a complete formal specification of the optrode based on analysis of the source code given and the detailed invariants imposed on the following model. Specifying the rules of the various types of maps which summarised the FSM found in the source code. The maps summarised the columns and their impact on the state to state relation of transitions found in the table.

A formal model was created in VDM which summarised the source code and which had many invariants which imposed the formal specification created. By feeding in the values of the source code FSM into the model both the original FSM (appendix A) and the corrected FSM (appendix C) could be verified formally, checking for any logical errors and illegal transitions. This verification could show that the original FSM had unused states which could not be transitioned to and code which did not follow the rule of self-looping. The corrected FSM did not have these issues and so verified correctly against the specification.

One issue that was not necessarily picked up in the verification or defined in the specification was no check that each node is transitioned to except for the start node which starts the program and so does not get called again. State 1 (get_cmd) does not check that it transitions to all stage one packet creator states and so could miss state 13 as being uncalled. Defining this in the VDM with maps being used would be tricky syntactically and logically. On top of this, errors would be thrown within the model anyway since state 15 which would be a stage one packet creator state would be called by a receive packet which is not allowed in the specification of receive packets. And if state 15 was defined as a stage two packet creator state then an error would still be thrown since the get_cmd state can only transition to stage one packet creator states and that transition would be illegal. Therefore, although an error would still be thrown in the circumstances the reasoning might not highlight the desired issue within the program and fully declare all that was erroneous.

Next, formal proof of correctness was attempted on the model. The VDM code was converted to an Isabelle model and formal proof was attempted using the 190 Proof Obligation states generated in Overture which was where the VDM model was based. Most formal proofs were showed true with few exceptions found, and others still need to be proven. The exceptions found showed that the State Maps or FSMs could be wrong if the map was empty. This required an invariant to define this in the model and these proofs should then resolve to true – when tested it appeared that a syntax issue could be at play as this logic was imposed but written in a different way. This logic was not being incorporated into that proof and so the exception was being unveiled when it may have otherwise been shown irrelevant.

Another issue exposed from verification techniques in this project was the need for a timeout method within the FSM loop since it was shown in [1] that there did not exist one in the source code and from this projects analysis of the loop it cannot prove termination and so could loop infinitely. Another issue needing improvement was the Overture Proof Obligation tool which provided too many proofs which were trivial and others which were spurious or even wrong. This was discussed in section 5.3.

From formal specification to formal verification using a model to the attempt at full formal proof complexity increased dramatically. The formal specification was simple to implement since it required analysis and discovery of patterns within code to work out a rule. Verification was more laborious since it required the creation of an accurate model which showed properties of the underlying source code but also held to the specification. Lastly formal proof was the hardest to implement because once again a model was required but then properties had to be drawn from this model to logically satisfy a property for the whole model. Proof was the trickiest section also because experience was a large part of success within this section.

Overall this task was a success even though full formal proof wasn't completed. It is said having a verified model is an advantage even if errors are found in the model or there is a gap between the model and the actual implementation [12]. The increased knowledge of the code from this project will benefit the CANDO team as they will benefit from the knowledge of the properties held in the FSM and can base their code around the fulfilment of those properties and rules. The proofs which were shown to be true will provide a reliability to the code regarding security and safety. Something paramount when considering medical devices and the aspect of human life which is at stake with any mistakes that occur!

Some limitations to the work undertaken are the lemmas remaining to prove and the proof obligations relating to the execute operation which were trivially false as the underlying functionality was missing. These proofs were hinting at a need for a proof based on the STATE holding a condition when an operation is called. In order to do this kind of proof a witness would be required which involves proofs of the witness as well as the proof of the obligation. These proofs can be completed with respect to this witness but would be more convoluted than simply copying the other POs across from VDM as had been happening. The limitation here are that not much more information can be gathered on these sections of the code.

8. Further Work

Further work can be completed despite this project being ended. The remaining proofs which are unproven or untranslated can be finished. Ideally all POs would be proved true, this will require the proofs which use the witness to also be implemented correctly and other missing POs should they be discovered. In the proofs it was shown that counter examples existed when the maps could be empty so correcting the models is something else that can be done.

When all this is completed the original source code can be edited to impose all the rules from the created specification; this code will be trustworthy as it will be proved to hold the properties shown from the model especially corrections which rewrite previous design errors. This feedback can be passed back to the CANDO team so that their future code can be up to the standard of the specification and their design process incorporate formal verification to increase the safety and security of the system.

To expand the project another model could be created using a real-time system. Currently errors that occur stop the execution of the model, by using a real-time system it could be possible to observe the system in an error state as well and test any timeout principles brought in. Another expansion would be to not just use the optrode in the head but also the chest piece. Verifying not just the chest piece but also the connections between the two of them.

In [2] a study of human behaviour was looked at to see how it impacted the software as it is known to be a major factor in system failure. Applying the techniques, they used to develop human errors the same idea could be applied to this project. If an accurate model of the hardware inside the human body was created, then one could attempt to test the impact of human based issues. These could be a person trying to play with the settings of their chest piece and the impact of that on the optrode or perhaps a knock to the head which jolts the device and the potential impact of an unlikely event like that.

To conclude, this project has caused a deeper understanding of safety and security in safety critical devices, particularly medical systems. It has also developed a greater knowledge of verification techniques, and the additional skills that have appeared due to taking part in such an endeavour.

9. Acknowledgements

Acknowledgements need to go to Alastair Pollitt and Leo Freitas, my supervisor, both of whom were involved with this project when I started it in the summer and were great guides. A huge amount was learned from Leo, after he kindly took me aboard last summer to be a part of this project. On top of this, an acknowledgement to the CANDO team for providing the code which was used and verified in this project. Finally, all the friends and family who spurred me on!

References

1. Pollitt, A.: Verifying the CANDO project optrode command interface in eCv. Newcastle University (2018).
2. Matthew L. Bolton, Kylie A. Molinaro, Adam Houser, A Formal Method for Assessing the Impact of Task-based Erroneous Human Behavior on System Safety, Reliability Engineering and System Safety (2019), doi: https://doi.org/10.1016/j.ress.2019.03.010
3. Hu, A.: Automatic Formal Verification of Software: Fundamental Concepts. IEEE, 1155-1159 (2009).
4. Zita, A. ; Mohajerani, S. ; Fabian, M. (2017) "Application of Formal Verification to the Lane Change Module of an Autonomous Vehicle". 13th IEEE Conference on Automation Science and Engineering August 20-23,Xi'an, China, 2017
5. Alemzadeh, H., Iyer, R., and Kalbarczyk, Z.: Analysis of Safety-Critical Computer Failures in Medical Devices. IEEE, 14-26 (2013).
6. Jiang, Z., Abbas, H., Jang, K.J., and Mangharam, R.: The Challenges of High-Confidence Medical Device Software. IEEE Computer Society, 34-42 (2016).
7. Félix Ingrand. Recent Trends in Formal Validation and Verification of Autonomous Robots Software. IEEE International Conference on Robotic Computing, Feb 2019, Naples, Italy. 2019. <hal-01968265>
8. Sametinger, J., Rozenblit, J., Lysecky, R., and Ott,P.: Security Challenges for Medical Devices. Communications of the ACM 58 (4), 74-82 (2015).
9. Imran, M., Zafar, N., Alnuem, M., Aksoy, M., and Vasilakos, V.: Formal verification and validation of a movement control actor relocation algorithm for safety critical applications. Wireless Netw, 247-265 (2016).
10. Artho, C. and Olveczky, P.: Formal techniques for safety critical systems. Science of Computer Programming (175), 35–36 (2019).
11. Harrison, M., Freitas, L., Drinnan, M., Campos, J., Masci, P., di Maria, C., and Whitaker, M.: Formal techniques in the safety analysis of software components of a new dialysis machine. Science of Computer Programming (175), 17-34 (2019).
12. Ortiz-Martin, L., Picazo-Sanchez, P., Peris-Lopez, P., Tapiador, J., and Schneider, G.: Feasibility analysis of inter-pulse intervals based solutions for cryptographic token generation by two electrocardiogram sensors. Future Generation Computer Systems (96), 283-296 (2019).
13. Daw, Z., Cleaveland, R., and Vetter, M.: Formal verification of software-based medical devices considering medical guidelines. Springer, 145-153 (2013).
14. Babamir, S., and Borhani, M.: Formal verification of medical monitoring software using Z language: a representative sample. Springer, 2633-2648 (2011).
15. Buonamici, F., Belmonte, G., Ciancia, V., Latella, D., and Massink, M.: Spatial logics and model checking for medical imaging. International Journal on Software Tools for Technology Transfer (2019) doi: https://doi.org/10.1007/s10009-019-00511-9
16. T. Belkhouja, X. Du, A. Mohamed et al., Biometric-based authentication scheme for Implantable Medical Devices during emergency situations, Future Generation Computer Systems (2019), https://doi.org/10.1016/j.future.2019.02.002
17. Bu, L., Karpovsky, M., and Kinsy, M.: Bulwark: securing implantable medical devices communication channels. Computers and Security, 1-14 (2018).
18. Epilepsy. NHS, https://www.nhs.uk/conditions/epilepsy/, accessed 06/05/19
19. CANDO: About cando, http://www.cando.ac.uk/aboutcando/, accessed 06/05/19
20. Wah, Benjamin W, et al. “Vienna Development Method.” p. 364., doi:10.1002/9780470050118.ecse447.
21. Verifying Multi-threaded Software with Spin. Spin checked, http://spinroot.com/spin/whatispin.html, accessed 14/05/19
22. Isabelle. https://isabelle.in.tum.de/, accessed 14/05/19
23. Zhao H, Dehkhoda F, Ramezani R, Sokolov D, Constandinou TG, Liu Y, Degenaar P, 2016, A CMOS-Based Neural Implantable Optrode for Optogenetic Stimulation and Electrical Recording, IEEE Biomedical Circuits and Systems (BioCAS) Conference, Publisher: IEEE, Pages: 286-289

Appendix A: Original Optrode_cmd_state_machine FSM

\Event State\	CONT	ERROR	SPI_TX_FINISH	SPI_RX_FINISH	LED_ON_E	SET_VLED_E	SET_BRE_E	LED_ALL_OFF_E	PROG_DELAY_DIAG	PROG_OP_MEM_1	RUN_MEM_E	PROG_CLK_CNT_E	RESET_ANA_E	SET_ANA_E	CONFIG_REC_E	PROG_ID_E	DUMMY_E	READ_LED	READ_DIAG	READ_LFP	GET_CMD
State 0	1	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 1	2	31	31	31	4	5	8	11	12	15	16	17	18	19	20	21	22	23	25	26	31
State 2	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 3	27	31	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 4	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 5	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 6	28	31	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 7	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 8	9	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 9	29	31	10	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 10	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 11	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 12	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 13	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 14	30	31	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 15	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 16	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 17	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 18	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 19	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 20	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 21	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 22	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 23	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 24	27	31	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 25	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 26	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 27	33	31	31	27	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 28	7	31	31	28	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 29	10	31	31	29	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 30	15	31	31	30	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 31	32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1
State 32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 33	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31

Appendix B: Optrode_cmd_state_machine FSM with Errors Marked

\Event State\	CONT	ERROR	SPI_TX_FINISH	SPI_RX_FINISH	LED_ON_E	SET_VLED_E	SET_BRE_E	LED_ALL_OFF_E	PROG_DELAY_DIAG	PROG_OP_MEM_1	RUN_MEM_E	PROG_CLK_CNT_E	RESET_ANA_E	SET_ANA_E	CONFIG_REC_E	PROG_ID_E	DUMMY_E	READ_LED	READ_DIAG	READ_LFP	GET_CMD
State 0	1	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 1	2	31	31	31	4	5	8	11	12	15	16	17	18	19	20	21	22	23	25	26	31
State 2	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 3	27	31	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 4	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 5	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 6	28	31	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 7	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 8	9	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 9	29	31	10	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 10	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 11	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 12	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 13	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 14	30	31	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 15	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 16	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 17	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 18	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 19	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 20	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 21	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 22	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 23	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 24	27	31	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 25	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 26	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 27	33	31	31	27	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 28	7	31	31	28	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 29	10	31	31	29	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 30	15	31	31	30	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 31	32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1
State 32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 33	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31

Appendix C: Final Optrode_cmd_state_machine FSM After Error Correction

\Event State\	CONT	ERROR	SPI_TX_FINISH	SPI_RX_FINISH	LED_ON_E	SET_VLED_E	SET_BRE_E	LED_ALL_OFF_E	PROG_DELAY_DIAG_	PROG_OP_MEM_1	RUN_MEM_E	PROG_CLK_CNT_E	RESET_ANA_E	SET_ANA_E	CONFIG_REC_E	PROG_ID_E	DUMMY_E	READ_LED	READ_DIAG	READ_LFP	GET_CMD
State 0	1	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 1	2	31	31	31	4	5	8	11	12	13	16	17	18	19	20	21	22	23	25	26	31
State 2	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 3	27	31	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 4	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 5	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 6	28	31	6	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 7	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 8	9	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 9	29	31	9	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 10	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 11	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 12	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 13	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 14	30	31	14	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 15	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 16	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 17	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 18	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 19	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 20	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 21	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 22	3	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 23	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 24	27	31	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 25	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 26	24	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 27	33	31	31	27	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 28	7	31	31	28	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 29	10	31	31	29	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 30	15	31	31	30	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 31	32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1
State 32	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
State 33	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31

Appendix D: States, Names and Operations

State Number	Name	Operation
0	start	start()
1	get_cmd	get_cmd()
2	LED_off	LED_off()
3	send_packet	send_packet()
4	LED_on	LED_on()
5	set_vLED	set_vLED()
6	send_packet	send_packet()
7	set_sDAC	set_sDAC()
8	set_bre	set_bre()
9	send_packet	send_packet()
10	set_dDAC	set_dDAC()
11	LED_all_off	LED_all_off()
12	prog_delay_diag	prog_delay_diag()
13	prog_op_mem_1	prog_op_mem_1()
14	send_packet	send_packet()
15	prog_op_mem_2	prog_op_mem_2()
16	run_mem	run_mem()
17	prog_clk_cnt	prog_clk_cnt()
18	reset_ana	reset_ana()
19	set_ana	set_ana()
20	config_rec	config_rec()
21	prog_ID	prog_ID()
22	dummy	dummy()
23	read_LED	read_LED()
24	send_packet	send_packet()
25	read_DIAG	read_DIAG()
26	read_LFP	read_LFP()
27	receive_packet	receive_packet()
28	receive_packet	receive_packet()
29	receive_packet	receive_packet()
30	receive_packet	receive_packet()
31	error	error()
32	chip_rst	chip_rst()
33	cmd_finish	cmd_finish()

Appendix E: State Transition Diagram of the FSM

Appendix F: SPIN Diagram for Termination Proof