Microcontroller Based Supervisory Control of a Solar Tracker

Kevin Searle

A Thesis in the Department of Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of

Master of Applied Science at

Concordia University, Montréal, Québec, Canada

December 20th, 2016

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Kevin Searle

Entitled: Microcontroller Based Supervisory Control of a Solar Tracker

and submitted in partial fulfillment of the requirements for the degree of

Master's of Applied Science (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Luiz Lopes	Chair
Dr. J. Dargahi:	External Examiner
Dr. C. Skonieczny:	Internal Examiner
Dr. Shahin Hashtrudi Zad:	Supervisor

Approved by _____

Chair of Department or Graduate Program Director

_____

Dean of Faculty

Date December 8th, 2016

Abstract

Microcontroller Based Supervisory Control of a Solar Tracker

Kevin Searle

The Supervisory Control theory (SCT) of Discrete-Event Systems is concerned with the design of supervisors that can generate appropriate control command sequences that meet the plant design specifications. Examples of these sequences include the startup and shut-down command sequences of a spacecraft engine.

This thesis addresses the implementation of Supervisory Controller (SC) executing on microcontroller hardware. The plant studied is a 2 degree-of-freedom solar tracker.

Existing implementations of Supervisory Control Theory focus on Programmable Logic Controller (PLC) based systems. PLCs are mainly used in process control and manufacturing applications. These implementations have proven advantages but suffer from numerous drawbacks. The hardware is large, expensive and over engineered for many embedded system applications. Additionally, a number of disconnects between Supervisory Control Theory and their practical application exist. These include but are not limited to: the Avalanche Effect, Inexact Synchronization and Simultaneous Events.

This thesis extends the application of SCT to the field of microcontroller-based embedded systems. Methods to minimize or remove the effects of Avalanche Effect, Inexact Synchronization, and Simultaneous Events are analyzed in a microcontroller-based environment.

Additionally, design procedures for the modelling and implementation of a supervisory controller executing within a microcontroller are explored. To this end, a physical implementation of a real system was created and documented. An offline computation of the system supervisor is derived in MATLAB and stored in the system's onboard memory as a State Transition Table (STT). An optimized storage method is developed that allows for fast execution of state transitions and a low memory footprint.

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Shahin Hashtrudi Zad for his support during the course of this Master's Thesis. This body of work would not have been possible without his constant support and patience.

Second, I would like to thank the management team of Rockwell Collins and the entirety of the Global 7000/8000 Avionics team for their flexibility and support. It's not an easy task working a full time position while completing a Master's Thesis. There were many occasions when I felt the balancing act was impossible, only to have my coworkers encourage and spur me forward.

Third, I would like to thank Dr. Scott Gleason and Dr. Samar Abdi for their support during the initial development of this project.

Finally, I would like to thank the wonderful Ms. Kimberly Bota for her emotional support during the development of this thesis. Words can't accurately describe my appreciation for her, so we shall have to leave it at that.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

This thesis explores the use of Microcontrollers as possible implementations platforms of Supervisory Controller Theory (SCT). Typical implementations of (SCT) utilize Programmable Logic Controllers (PLCs) as their development platforms [1] [2] [3]. PLC based applications have proven advantages, but suffer from drawbacks such as Simultaneous Events, Inexact Synchronization and the Avalanche Effect [1]. Additionally, PLC hardware is large, expensive and over engineered for the needs of most small scale, low power embedded systems. For this reason, this thesis considers the application of supervisory control theory operating within the microcontroller of an embedded system.

However, the use a microcontrollers in supervisory controller theory is not without its own set of disadvantages. The amount of memory available to microcontrollers is limited in size. This is a notable limitation, as supervisory controllers tend to compile into very large sets of data, which in turn must be stored inside the system. Additionally, issues such as Simultaneous Events and Inexact Synchronization are also present in microcontroller based applications of SCT. In order for the advantages of microcontroller based implementations to be of any value, these notable disadvantages must be overcome or minimized.

To this end, this thesis outlines the development of a memory efficient, search optimized storage method that allows for supervisors to be stored and executed within the microcontroller's execution environment. Additionally, the issues of Simultaneous Events, Inexact Synchronization and the Avalanche Effect are explored in detail and methods to reduce or eliminate their effects are developed. The thesis is organized as follows:

The remainder of this chapter introduces a brief overview of discrete events systems and supervisory control theory. An overview of PLCs and Microcontrollers is then provided. A detailed literature review of existing PLC and Microcontroller based supervisory control theory applications is provided. Finally, a list of thesis contributions is clearly outlined.

1.1 SUPERVISORY CONTROL OF DISCRETE EVENT SYSTEMS

A complex system can be broken down into sub components and modelled as finite state automatons. The finite state machines create a discrete state-space and use event-driven state transitions to fully define the behaviour of the system.

For large systems with multiple components, a separate automaton that defines the behaviour of each system component can be modelled. By modelling each system component, they can be reassembled (by applying a synchronous product between the models) to fully define the system's behaviour. The synchronous product of all component models is known as the "plant". The plant model is therefore a discrete event system that can be defined as a deterministic automaton that fully defines all possible behaviour for which the plant is capable of manifesting.

Typically, at least some part of the plant's behaviour is unsafe. Supervisory Control Theory (SCT) [4] is a general approach that allows the synthesis of a Controller (supervisor) for a Discrete Event Systems (DES). By modelling a system as a Discrete Event System (plant), a designer can automatically synthesize a supervisor capable of controlling the plant to stay within a particular specification. A safety specification is in the form of a set of sequences (language) which is later modelled as automata.

The supervisor dynamically disables plant events that lead to states or transitions which are not permitted by the plant specifications. Events generated by the plant are made visible to the supervisor. A visualization of the relationship between the Plant and its supervisor is illustrated below, where G denotes the Plant and S the supervisor:

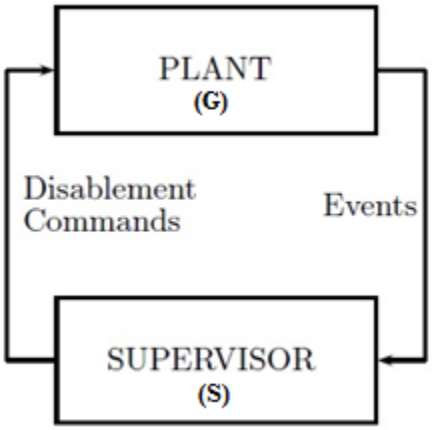


FIGURE 1.1-1 PLANT AND SUPERVISOR INTERACTION

An event may be controllable or uncontrollable, as such, not all events present in the plant's alphabet may be disabled. The supervisor may only enable or disable controllable events. In general, controllable events represent actions the plant can perform, such as motor actuations, initiation of a broadcast over the wireless device, opening a valve, etc. Uncontrollable events represent events that may occur within the plant and cannot be prevented (disabled) by the supervisor, such as events associated to periodically polled sensor data, information received from a master controller, failure indications, etc. This divides the event set of the plant into two disjoint sets, controllable and uncontrollable events.

$$\sum_p = \sum_c \cup \sum_{uc}$$

This leads to a number of properties unique to a supervisor. The first is the notion of admissibility. A supervisor automaton is said to be admissible with respect to the plant if the supervisor does not disable any uncontrollable events within the plant. The supervisor must ensure the plant satisfies all design specifications. Some of the design specifications deal with safety issues. Other deal with block issues.

The system under supervision must be nonblocking (i.e. must be free of deadlocks and livelocks). By creating a supervisor the system under control is guaranteed to function within the behaviour defined by the specification. As the specification is an automaton, the design process is visual and relatively straight forward.

1.2 PROGRAMMABLE LOGIC CONTROLLERS (PLCS)

This sub-chapter offers a brief overview of Programmable Logic Controllers (PLCs). A PLC is essentially an industrial digital control device that was first developed in the late 1960s. As these are industrial devices, they tend to be larger and more rugged than other comparable commercial digital devices.

They typically consist of an I/O (Input/Output) module, a CPU, and a Programmer/Monitor. A visualization of this can be seen in the figure below:

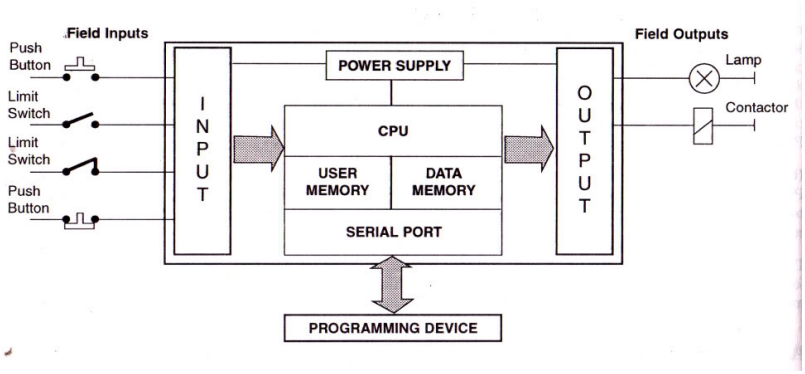


FIGURE 1.2-1 PLC BLOCK DIAGRAM [5]

The I/O card allows easy integration of peripherals such as Push Buttons, Analog Sensors, Lamps, Relays, etc. The CPU portion houses an onboard digital computer (typically a microprocessor), a memory array and a communication interface. The Programmer/Monitor allows new programming requirements to be uploaded to the controller and monitoring of the device itself.

The commonly used programming language for PLCs is Ladder Diagram (LD), a.k.a Ladder Logic, and is controlled via the IEC 61131-3 international standard. Ladder logic is a visual coding method using rungs, contacts, coils and latches. A sample ladder logic diagram is illustrated in the figure below:

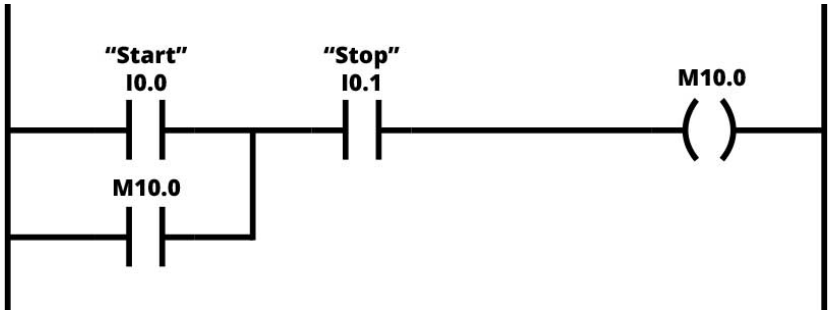


FIGURE 1.2-2 SAMPLE LADDER LOGIC DIAGRAM [6]

The ladder logic execution is sequential, meaning the evaluation begins at Rung number 1 and sequentially evaluates all remaining Rungs in ascending order.

PLCs execute a scan cycle in order to collect input data, execute the logic stored in the LD, and energize outputs. In some instances a communication and diagnostics cycle are also performed.

Scan cycles are continuously executed by the PLC as quickly as possible. The amount of time required for a scan cycle to complete depends on the amount of inputs, the size of the LD. The technology used in the PLC also impacts how quickly the LD is executed. Typical scan cycle times can be as low as 10ms but may be as long as 100ms. A visualization of a scan cycle is seen below:

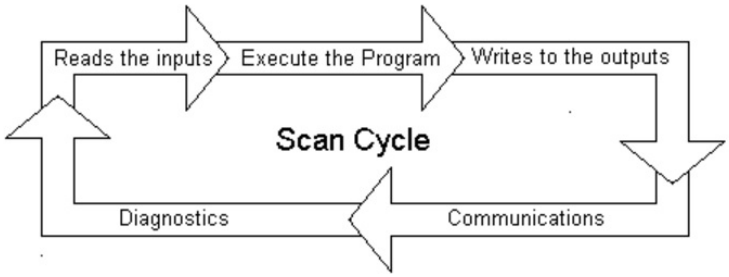


FIGURE 1.2-3 PLC SCAN CYCLE [7]

PLCs are often selected as a SCT implementation platform because they are durable, relatively easy to program, and are already widely used in the automation industry. They utilize microprocessors in order to evaluate the logic encoded using Ladder Diagrams.

1.3 MICROPROCESSORS AND MICROCONTROLLER UNITS (MCUS)

Microcontrollers are specially designed boards that allow easy integration of peripherals with the onboard microprocessor. Typically, microprocessors require a sizeable amount of support electronics to allow interfacing of peripheral devices with the various functionalities supported by the microprocessor. For example, the analog to digital converters on a microprocessor often benefit from low pass filters in series with their input pins. For this reason, a Microcontroller device may include onboard electronics that can be easily enabled this feature.

In general, microcontrollers are rarely used in final implementations for commercial products, as many of the onboard electronics native to the microcontroller may not be required. Instead, a custom made electronics board containing the microprocessor, memory, and specific hardware functionalities required to support the required peripherals is manufactured. The creation of a custom electronics board is outside the scope of work of this thesis. Instead, a microcontroller that allows easy integration of peripheral components is used.

The use of a microcontroller unit for the implementation outlined in this thesis is acceptable. The purpose of this implementation is to establish a proof of concept, not to prepare for the mass production of a new commercial product.

In general, microcontrollers are smaller digital computation units (when compared to PLCs) that contain a processor, memory, a wide (and varying) array of input/output peripherals, and a wide (and varying) array of communication protocols. A visualization of a basic microcontroller block diagram is illustrated below:

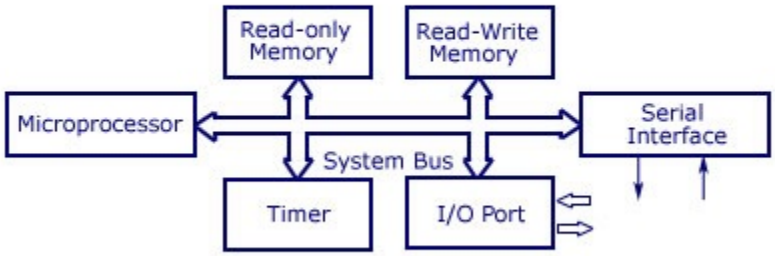


FIGURE 1.3-1 BASIC MCU BLOCK DIAGRAM [8]

The behaviour and functionality of the microcontroller are typically coded in the "C" programming language. This can be seen as either an advantage or a disadvantage, depending on the designer's engineering background. In general, the "C" programming language allows for a large degree of control over the functionality of the unit, yielding numerous benefits in terms of execution times and memory storage. For this reason, a microcontroller was selected as the implementation platform for the supervisory controller.

1.4 LITERATURE REVIEW

The implementation of supervisory control theory (SCT) has been extensively explored in academia [1] [2] [3] [9] [10] [11] [12]. The theory states that by modelling a real system as a discrete event system, a supervisor can be automatically generated that guarantees that "the resulting controlled behaviors do not contradict the behavioral specifications and are nonblocking. Additionally, the resulting controlled behaviours are maximally permissive within the behavioral specifications." [1]. For critical systems, guarantees over the controlled behaviour of a system are obviously desirable qualities.

However, numerous disconnects between the theory of supervisory control and their practical applications are exist. First, issues with the accurate modelling of a real system as a discrete event system must be overcome. Second, there are a number of limitations on the behaviour of SCT that must be identified and minimized in the sequential environment of digital systems.

To overcome the first issue, the real system must be completely and accurately defined by a discrete model. For large systems this poses a significant challenge, as some systems can "quickly reach state spaces of 10^{16} possible combinations" [3]. Attempting to model such a large system as a single monolithic model is extremely difficult and error prone. Instead, the system itself is broken down into the various components that are used by the system, each being modelled in turn. "It is more accurate to design several small models as opposed to keeping track of all the intricacies of one large model." [3] Additionally, "It is quicker to design several small models than one large model." [3] Therefore, to minimize the risk of disconnects and decrease the modelling time required, "the system as a whole must be decomposed into several independent subsystems, each one uniquely associated with a specific piece of equipment" [9]. With the subcomponents modelled, they are recombined using an operation called the "synchronous product", creating a single model called the "plant" that describes the full behaviour of the system. From here, designers may create specifications the supervisor must enforce and a supervisor that guarantees the non-blocking, maximally permissive behaviour can automatically be calculated. The designed supervisor itself will be in the form of an automaton.

With a plant and supervisor modelled and generated, the results must be implemented in a physical environment. However, "There are very few guidelines for how to implement the controller, once the abstract supervisor model has been calculated". [10] [2]. Additionally, there are numerous disconnects between the theory of supervisory control and their practical implementation. First, SCT states that "the plant serves as the sole generator of events" [12] and that these "events occur spontaneously" [12]. In real systems, not all events occur spontaneously. In [10], the author argues that "a plant receives commands and reacts to these commands with responses." [10]. In other words,

events that occur as responses to commands will not be generated spontaneously and will only occur as a response to the generator event. This is a phenomenon known as *causality* and "cannot be avoided in the implementation; designers have to answer the question "who generates what"" [2]. In order to overcome the issue of causality "the implementation needs something which generates all events, but the plant only generates the uncontrollable ones (events)" [13]. Second, the theory of supervisory control assumes that events occur asynchronously. This is not possible in the synchronous realm of digital logic as changes are only observed in discrete time slices. For example, "two events may occur between successive clock cycles" [12] which may in turn lead to "several events occurring simultaneously" [2] - "For practical purposes, the supervisor must interpret these as simultaneous" [12]. While this may work in practice, it is a deviation from the assumptions of supervisory control theory and introduces a new phenomenon known as *Simultaneous Events*. This occurs when "the supervisor has to choose between several event alternatives in a single state" [2]. If the implementation is that of a PLC, the selected event depends on "the order of the implementation internally made in the PLC" [2], in other words, the first expression in the program to be evaluated as true will be selected. This creates possible dormant issues, "if designers do not choose the sequential execution of the program within the PLC, the PLC will make the choice by itself" [2]. If errors unknowingly introduced in the rung ordering of the PLC Ladder Diagram, the behaviour of the system may be entirely different. Even worse, the only way to test these conditions is to stimulate simultaneous events, which can be difficult to reproduce. Microcontrollers can offer more flexibility when tackling the issue of *Simultaneous Events*, as scheduling policies can be applied that determine which event should be triggered first. The final issue with the implementation of supervisory control theory is the issue of *inexact synchronization.* In the real world, exact synchronizations do not exist due to the "time delays and the periodic updating of input signals" [2]. For PLC based implementations it is impossible to ensure the "implemented supervisor executes in exact synchrony with the plant" [2] due to the cyclic nature of the PLC scan cycle. Microcontroller based implementations are not limited to PLC style scan cycles. "In microcontrollers, the freedom of design gives options to reduce this (inexact synchronization) problem" [13], additional options such as faster algorithms can lead to better response times and a reduction in inexact synchronization. In [13], the authors outline two methods in which the calculated supervisor of a real system can be stored in

the onboard memory of a microcontroller. The first method is known as a "chained list method that prioritizes execution speed" [13]. The second method is known as the "memory safe method that prioritizes memory size" [13]. The authors of [13] also note that the memory space available for microcontrollers is a limited resource. It is therefore important to understand the amount of memory the calculated supervisor requires so appropriate hardware can be sourced ahead of time.

1.5 THESIS CONTRIBUTIONS AND OUTLINE

The motivation behind this thesis is to further expand the implementation of supervisory control theory in the field of microprocessor based embedded systems. Current implementations focus heavily on the use of Programmable Logic Controllers, which can be impractical or over-engineered for some applications.

In [10], the author noted that "very few guidelines for how to implement the controller, once the abstract supervisor model has been calculated". The first objective of this thesis is to outline a development process that enables a supervisory controller to execute on a microprocessor based system. Chapter 3 outlines the design and modelling methods used to create a dual axis solar tracking system, forming a plant with 1,728 states and 20,192 transitions. The software used to generate, store, and execute a non-blocking, maximally permissive supervisory controller is outlined in Chapter 4. The microprocessor executes the supervisory controller using the "C" programming language. The process outlined in this thesis can be highly automated, straight forward, and benefits from numerous advantages (discussed in Chapter 5).

The second objective is to develop a method that enables the efficient storage and fast execution of the calculated supervisory controller. By achieving this, the effects of inexact synchronization between the plant and supervisor may be reduced. Additionally, memory size is a limited resource for microcontrollers. The storage solution must therefore optimize the amount of space required as well as execution speed.

The third objective is to illustrate that some of the issues faced in PLC based implementation of supervisory control theory are not applicable or mitigated in the microprocessor based implementation outlined in this thesis. The testing results and discussions are presented and analyzed in Chapter 5.

Chapter 2 presents a brief review of languages, automata, and operations on automata used in this thesis.

Chapter 3 defines the application selected for development and the required hardware components. A design outlining the process used to transform the hardware into a set of component models is then detailed. Finally, the hardware interactions and system specifications are defined in a set of discrete models.

Chapter 4: outlines the software process used to transform the discrete models generated in Chapter 3 into a supervisor stored as a State Transition Table (STT). The microcontroller's driver code and details on how the supervisor is translated into C code is then explained. The Chapter concludes by outlining the Evaluation Cycle that enables the triggering of events.

Chapter 5: details the results achieved by this thesis. The effects of the implementation detailed in this thesis on the notable disadvantages with Microcontroller based application of SCT are explored. It concludes by analyzing the code size required for the implementations three implementations considered in this thesis.

Chapter 6: Summarizes the thesis conclusions and outlines a number of future research topics that can potentially be explored as a result of the findings presented in this thesis.

CHAPTER 2. BACKGROUND

This chapter provides a brief review of formal languages, automata, and common operators used on automata. The contents of this chapter are mainly based off on [14] and [15].

2.1 LANGUAGES

The transitions that occur within a discrete event system's state space are attributed to the triggering of discrete events. The complete list of all possible events that can occur within the DES is called the **Alphabet** also known as "∑". As events occur within the DES, they form a **string**, or **trace**, of events. The set of all possible strings that can occur within a DES is known as the system's **language**. The empty string, denoted by ϵ, is defined as a string with no event. The language that includes all the strings except ϵ is defined as:

$$\sum^{+} = \{\alpha_1 \dots \alpha_n | n > 0, \alpha_i \in \sum\}$$

Then define:

$$\sum^{*} = \sum^{+} \cup \{\epsilon\}$$

2.2 AUTOMATA

An automaton is the formal manner in which discrete event systems are expressed. Each "model" is composed of 5 components and can be visualized below:

$$G = \langle X, \sum, \eta, x_o, X_m \rangle$$

where: X is the finite set of discrete states the automaton can occupy.

$\sum$ is the finite set of events that the automaton can generate (the alphabet).

η is the partial transition function of the plant (X x $\sum \rightarrow$ X)

x_o is the initial state of the plant.

X_m is the set of marked states (a.k.a: Final, Marker, Accepting states)

For example, a system may contain an assembly line that can be modelled as three separate states. State I identifies that the assembly line is Idle, State P identifies that the

assembly line is Producing, and state D identifies that the assembly line has broken down. The α, β, λ, μ events are used to transition between these states, where α occurs when a new part arrives, β occurs when a part is assembled, λ occurs when the line has broken down, and μ occurs when the line has been repaired. This relationship can be visualized in the figure below:

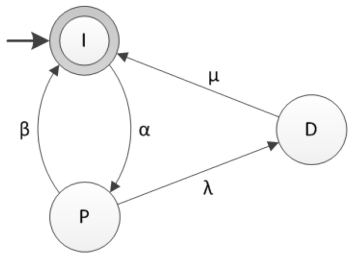


FIGURE 2.2-1 EXAMPLE ASSEMBLY LINE AUTOMATON

Using the Automaton 5 tuple format outlined above, it yields the following information

$X = \{I, P, D\}$

$\sum = \{\alpha, \beta, \lambda, \mu\}$

η = The partial transition function, which is a set of functions that defines the state that must be transitioned to for each state and event permutation.

$\eta(I, \alpha) = P,$

$\eta(I, \beta) = \eta(I, \lambda) = \eta(I, \mu) = \text{Not Defined}$

$\eta(P, \beta) = I, \eta(B, \lambda) = D$

$\eta(P, \alpha) = (P, \mu) = \text{Not Defined}$

$\eta(D, \mu) = I$

$\eta(D, \beta) = \eta(D, \lambda) = \eta(D, \alpha) = \text{Not Defined}$

$x_o = I$

$X_m = I$

The partial state transition function returns the target state of each transition. The definition can be extended over the entire $\sum^*$(all possible sequences of alphabet symbols

present in the plant, including the empty sequence ϵ) by applying a recursive approach to the partial transition function.

From the example above: $\eta(I, \alpha\,\lambda) = \eta(\eta(I, \alpha), \lambda) = \eta(P, \beta) = D$

The closed behaviour of the plant "G", denote by L(G), is the set of strings that the plant may generate. A few examples from the figure above are: α, αβ, αβα, αβαβ, αλ…

The strings β, λ, αββ are a few examples of strings that are not be part of the closed behaviour of the plant, as they cannot be generated by the assembly line automaton.

The second property is the marked behaviour of the plant "G", denoted by $L_m(G)$. These are the set of strings that take the automaton from the initial state to marked state(s). From the example above, αβ, αβαβ, αλμ are a few sequences belonging to the marked behaviour of the plant.

The closed and marked behaviour of the plant automaton are used during the synthesis of the supervisory controller and are therefore important to the modelling process.

2.3 SUPERVISORY CONTROL

Consider a DES plant "G" and a safety design specification modelled as an automaton "SPEC". Suppose the plant event set is:

$$\Sigma = \Sigma_c \cup \Sigma_{uc}$$

Where: $\sum_c$ and $\sum_{uc}$ are the controllable and uncontrollable event sets.

Let S denote the supervisor and S/G, the system under supervision, illustrated below:

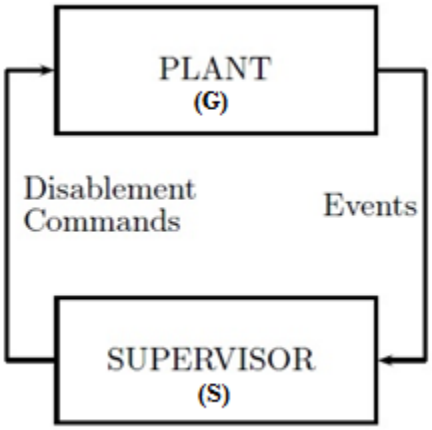


FIGURE 2.3-1 PLANT (G) / SUPERVISOR (S) INTERACTION

The objective in a supervisory control problem is to find a supervisor S such that:

$$(1)\ L_m(S/G) \subseteq\ L_m(SPEC)$$

$$(2)\ \ L(S/G) \subseteq\ L(SPEC)$$

$$(3)\ (S/G) is\ nonblocking$$

Here, L( ) and L_m( ) denote the closed and marked behaviours.

[16] shows that the above problem has an optimal (minimally restrictive) solution given by the supremal controllable sublanguage of:

$$L_m(G) \cap\ L_m(SPEC)$$

[4] Provides an algorithm for the computation of this language and the resulting supervisor.

2.4 DISCRETE EVENT TOOL KIT AND FUNCTIONS ON AUTOMATA

This thesis implementation makes use of the Discrete Event Control Kit (DECK) developed by Dr. Shahin Hashtrudi Zad and two of his graduate students, Shauheen Zahirazami and Farzam Boroomand [15]. DECK is a toolbox (a set of functions) written in the programming language of MATLAB [2] for the analysis and design of supervisory control systems based on discrete–event models.

This implementation makes use of three notable DECK functions: Sync, Product and SupCon. Details about these three are available in the sections that follow. For a complete list of DECK functions refer to Appendix B.

2.4.1 SYNCHRONOUS PRODUCT (PARALLEL COMPOSITION):

The synchronous product operation is denoted by || and expresses the result of the automata functioning jointly, where common events are only allowed to occur when both automata execute them simultaneously, and events not contained in both alphabets are always allowed to occur whenever they are defined in one of the automata [12].

Formally the synchronous product is defined as follows:

Let $G_1\ =\ (X_1, \sum_1,\ \delta_1, x_{0,1}, X_{m,1})$ and $G_2\ =\ (X_2, \sum_2, \delta_2, x_{0,2}, X_{m,2})$. Then Define:

$$G_1\ ||\ G_2 = Reachable\ Part\ of\ (X_1 \text{ x } X_2, \sum_1 \cup\ \sum_2, \delta, (\ x_{0,1}, x_{0,2}), X_{m,1} \text{ x } X_{m,2})$$

where

$$\delta\big((x_1, x_2), \sigma\big) \coloneqq \begin{cases} (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma)) & if\ \sigma \in \sum_1 \cap \sum_2 \wedge \delta_1(x_1, \sigma)! \wedge \delta_2(x_2, \sigma)! \\ (\delta_1(x_1, \sigma), x_2) & if\ \sigma \in \sum_1 - \sum_2 \wedge \delta_1(x_1, \sigma)! \\ (x_1, \delta_2(x_2, \sigma)) & if\ \sigma \in \sum_2 - \sum_1 \wedge \delta_2(x_2, \sigma)! \\ undefined & otherwise \end{cases}$$

2.4.2 PRODUCT (MEET)

The product, or Meet, of two or more automata is used to join two of more specifications into a single specification. Consider the following two automata:

$$G_1 = (X_1, \sum_1, \delta_1, x_{0,1}, X_{m,1}), G_2 = (X_2, \sum_2, \delta_2, x_{0,2}, X_{m,2})$$

Then,

$$G_1 \text{ x } G_2 = Reachable\ Part\ of\ (X_1 \text{ x } X_2, \sum_1 \text{x} \sum_2, \delta, (x_{0,1}, x_{0,2}), X_{m,1} \text{ x } X_{m,2})$$

where

$$\delta\big((x_1, x_2), \sigma\big) \coloneqq \begin{cases} (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma)) & if\ \delta_1(x_1, \sigma)\ and\ \delta_2(x_2, \sigma)\ are\ defined \\ undefined & otherwise \end{cases}$$

By the definition at the state (x_1, x_2) the event σ appears in G1 × G2 if and only if G1 and G2 can execute σ from x_1 and x_2) respectively. Consequently, it can be shown that product operation represents the intersection of languages of automata G1 and G2:

$$L(G_1 \times G_2) = L(G_1) \cap L(G_2)$$

$$L_m(G_1 \times G_2) = L_m(G_1) \cap L_m(G_2)$$

2.4.3 SUPCON (SUPERVISORY CONTROLLER)

The SupCon function is a DECK function that calculates the supremal controllable sublanguage of the first input automaton (G) to the second automaton (SPEC) and the input vector of uncontrollable events. The function produces an automaton that marks:

$$SupC(L_m(G) \cap L_m(SPEC))$$

This automaton is calculated based on the algorithm in [4] and can be regarded as a realization of a maximally permissive supervisor.

CHAPTER 3. SYSTEM OBJECTIVES, HARDWARE AND MODELLING

This chapter outlines the modelling and development process of a Dual Axis Solar Tracking system and the subsequent modeling as a Discrete Event System. The chapter begins by outlining the objectives of the system under application. Next, a detailed analysis of the hardware used to satisfy the system objectives are reviewed. Finally, the modular modelling process is explored that enables the "real" system to be translated into a Discrete Event System.

3.1 SYSTEM OBJECTIVES

The application chosen for development is a dual axis solar tracking system. Two position controlled servo motors are used to articulate a photovoltaic (PV) cell mounted on a pan-tilt assembly. Power collected by the PV cell is regulated by a maximum power point tracking device and distributed to the system components. Excess power is stored in the system's onboard battery to power the system components when insufficient power is generated by the PV cells.

The system employs a wireless transmitter/receiver that allows two way communications with a higher level system, here after referred as the Master Controller (MC). The schematic diagram of the solar tracker is shown in Figure 3.1-1:

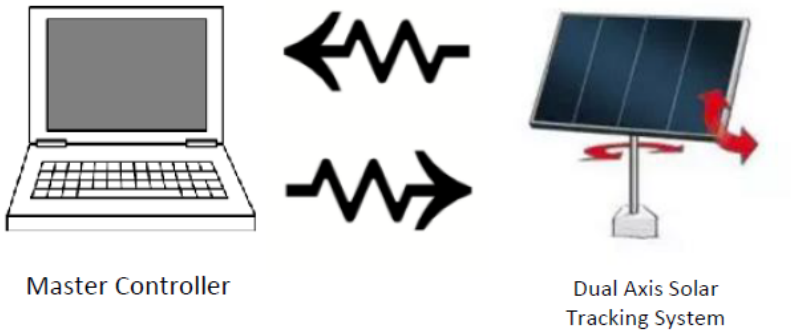


FIGURE 3.1-1 MASTER CONTROLLER AND DUAL AXIS SOLAR TRACKING SYSTEM INTERACTION

Four sensors are used to monitor the state of the system components. A voltage sensor is used to monitor the voltage generated by the solar panel and corresponds to the

illumination level present on the cell. Current sensors are placed in series with the Servo Motor power lines to monitor current consumption after a move command has been issued. If the current sensor detects a continuous current draw, the motor is declared to be obstructed. This implementation assumes failure conditions in one motor, the Elevator. Finally, a battery state of charge (SOC) fuel gauge monitors the onboard battery and reports the battery SOC as a percentage to the microcontroller.

The hardware interactions are visualized in the figure below:

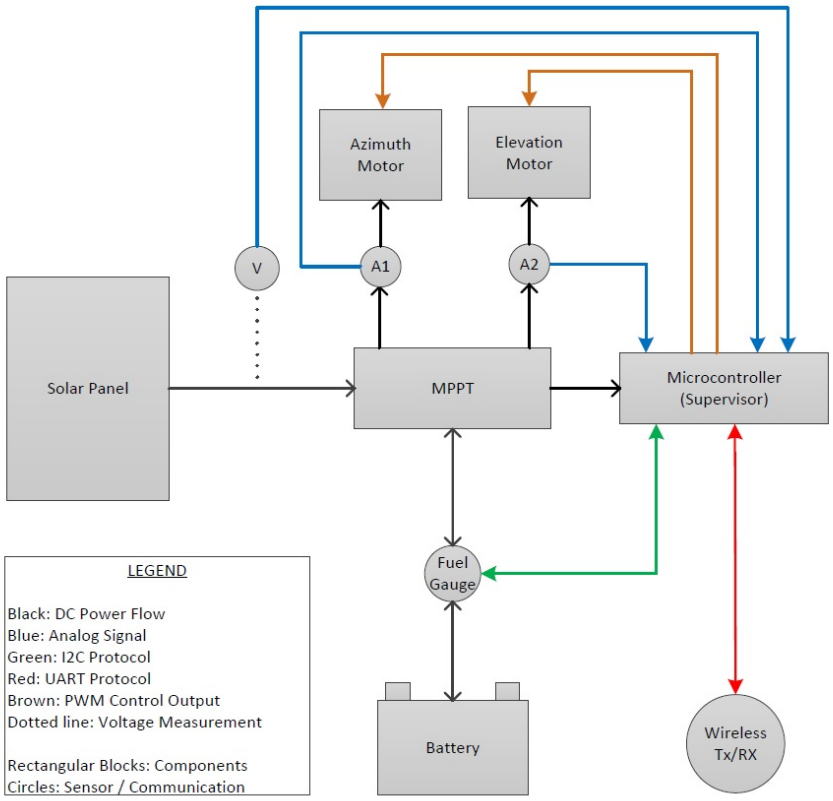


FIGURE 3.1-2 DUAL AXIS SOLAR TRACKING SYSTEM SCHEMATIC DIAGRAM

The system's main objective is to store Solar Power generated by the PV cells into the System's onboard battery. The concept is to simulate a power generation system integrated within a larger device (eg: a cube sat) that shares the usage of the onboard battery.

The system reacts to commands to reposition the azimuth and elevation of the solar panel. The commands are broadcast wirelessly from a laptop that acts as the MC in this implementation. The sweep commands are:

- Azimuth Clockwise (CW) Sweep
- Azimuth Counter-Clockwise (CCW) Sweep
- Elevation Clockwise (CW) Sweep
- Elevation Counter-Clockwise (CCW) Sweep
- Full Sweep

The system reacts to two separate types of Sweep commands. Individual motor control is performed with the Azimuth CW, CCW and Elevation CW, CCW sweep commands. The Full Sweep command will initiate a pre-determined motion path that allows full hemispherical measurement of the illumination levels surrounding the system. When any of the above commands are received, the system begins sweeping the solar panel in the desired direction. As a sweep command is performed, the system continuously monitors the illumination on the solar panel. If a bright level is detected the system ceases motor movement and indicates to the master controller that the sweep was successful. Each motor has a limited range of 180 degrees. If the maximum motor range is reached without detecting sufficient brightness, the system indicates to the master controller that the sweep has failed. If an obstruction prevents the elevation motor from reaching the commanded position, the system informs MC of the motor failure and the system prevents future elevation movement commands to be issued. A list of system responses to control commands is listed below:

- Bright Detected
- Sweep Failure
- Motor Failure

The implementation of the MC is a python based graphic user interface executing on a windows based laptop computer. The commands are issued to the system by pressing the corresponding Sweep buttons. The values measured by the four sensors and the system responses are made visible to the user. In this manner, the status of the system is made aware to the users.

3.2 SYSTEM HARDWARE

This Chapter explores the selected hardware in more detail. The datasheet of all hardware used are readily found via online sources, this Chapter outlines a few specific parameters that are essential to the operation of the system.

3.2.1 SOLAR CELL

Solar panels generate DC power proportional to the amount of illumination and its angle of incidence to the panel. The ideal condition is to have the panel perpendicular (ie: 90 degrees) to the illumination source. The current and voltage produced by a solar panel increase in a proportional fashion to the sun's intensity. The Solar Cell used in this implementation is a PT15-300 by the company "Flex Solar Cell and has a maximum output wattage value of 3.08 Watts (in full and direct sunlight). The current vs voltage curve is shown in the figure below:

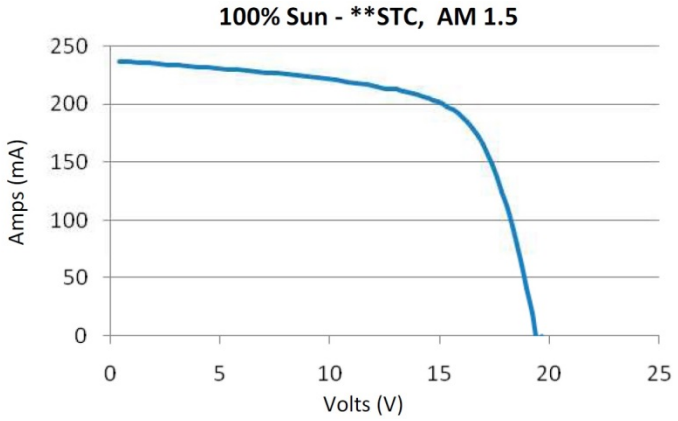


FIGURE 3.2-1 PT15-300 PV CELL, CURRENT VS VOLTAGE [17]

As the light intensity diminishes, the output current will decrease but the voltage will remain fairly consistent. The power generated is a product of these two parameters, and the percentages of power produced relative to the full sunlight graph seen above are illustrated in the table below:

TABLE 3-1 PV CELL EFFICIENCY RATINGS [17]

Energy Available at Various Light Conditions Relative to Full Sun	
Condition	**Intensity** (% of full sun)
Full sun - Panel square to sun	100%
Full sun - Panel at 45° angle to sun	71%
Light overcast	60-80%
Heavy overcast	20-30%
Inside window, single pane, double strength glass, window & module square to sun	91%
Inside window, double pane, double strength glass, window & module square to sun	84%
Inside window, single pane, double strength glass, window & module 45° angle to sun	64%
Indoor office light - at desk top	0.4%
Indoor light - store lighting	1.3%
Indoor light - home	0.2%

The maximum power that the solar panel can produce is known as the “knee” or -3db point of the graph shown in Figure 3.2-1. This is equivalent to the 15.4 V, 200mA mark that results in a power output of 3.08 Watts.

For testing purposes it can be challenging finding a bright, sunny day in Montreal (especially in the winter months). For this reason, the test setup uses a simple desktop lamp that simulates the light source. From Table 3-1 we observe that the power generated by the solar panel is extremely low. This is due to the fact that the current produced when subjected to an indoor light is extremely low. However, the characteristics of the panel are such that voltage produced by the panel is not affected by lower light intensity. Since the intensity of the light source used is small, the sensor selected to monitor the brightness present on the solar panel is a voltage monitor, as accurate measurement of extremely low currents can be challenging. In an ideal condition, the sensor used to measure the brightness on the panel should be a current sensor.

3.2.2 MAXIMUM POWER POINT TRACKER

The amount of power generated by a solar cell is a product of the voltage and the current generated as a by-product of the photovoltaic effect. The graph shown in Figure 3.3-1 is recalculated to illustrate it as a power curve:

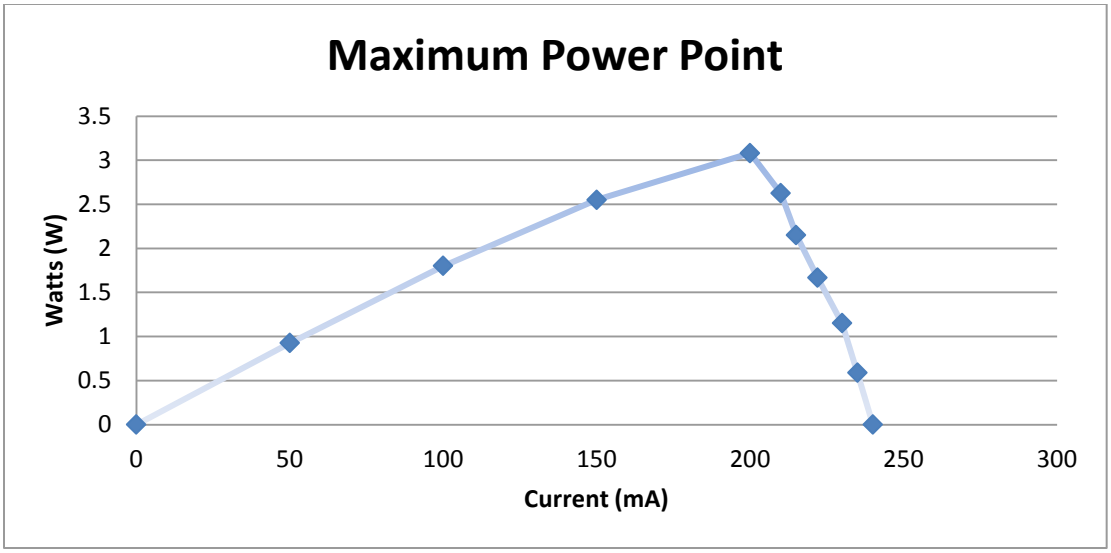


FIGURE 3.2-2 MAXIMUM POWER POINT

The graph clearly illustrates that the maximum power generated by the solar panel occurs around the 200 mA point, which corresponds to 15.4 volts. This is known as the -3db point (knee). By limiting the output voltage of the solar panel to this level, it is possible to obtain the maximum power output from the generator.

The maximum power point tracking device is a step-down DC to DC power converter designed to limit the PV cell output voltage to 15.4 volts. By doing this, the PV cell will continuously produce 3 watts of power when the solar panel produces greater than 15.4 volts.

The MPPT device used in this implementation is a "SparkFun Sunny Buddy - MPPT Solar Charger". This board is capable of accepting solar panel voltages up to 20 volts and charges a single cell Lithium Polymer (LiPo) battery greater than 450mAh. Additionally, the board allows for parallel connections between the onboard battery and the system load, allowing seamless power transmission between the PV cell, onboard battery, and system components.

3.2.3 ONBOARD BATTERY

The onboard battery selected for this implementation is a basic 1 cell, 3.7V LiPo battery capable of producing 2200mAh of current.

3.2.4 MICROCONTROLLER

The microcontroller selected for this implementation is an EFM32LG990F256. It is an ARM Cortex-M3 based MCU operating at 48 MHz and is built on a platform oriented towards low energy consumption. Key features include 256 kB of Flash memory, 32 MB of RAM, 87 Digital I/O pins, 4 x 16 bit timers, UART, I2C, Direct Memory Access (DMA) and Analog to Digital Converters (ADC).

The EFM32LG-STK3600 is a starter kit created by Silicon Labs and enables quick access to the various I/O pins via breakout pads. The onboard memory is programmable via USB interface. As this is an ARM based MCU, multiple Integrated Development Environments (IDEs) are possible. The Silicon Labs "Simplicity Studio" was selected for this implementation.

3.2.5 VOLTAGE SENSOR

The illumination present on the solar panel is monitored by a Phidgets 1135 precision voltage sensor connected in parallel to PV cell output. The sensor is capable of translating a differential voltage of ±30 Volts into an analog signal ranging from 0.5 to 4.5 Volts. The ADC of the EFM32LG utilizes a 12 bit sequential approximation register to translate the analog value into a digital value between 0 and 4095. A 4.5V signal on the ADC corresponds to a value of 4095 (or 30 Volts) and a value of 0.5 volts corresponds to 0 (or -30 Volts). A zero volt measurement across the PV cell is therefore equivalent to a 2.5 analog signal which converts to a 2048 digital value.

The formula to convert a 12 bit digital ADC reading back to its corresponding analog voltage is defined as:

$$Analog\ Voltage\ Measured = ADC\ Reading * \frac{4096}{5V} \qquad \textit{(Formula 1)}$$

Once the original measured analog voltage is calculated, the following formula is used to translate into a ±30 Volt signal:

$$Vdiff = \frac{Analog\ Voltage\ Measure - 2.5}{0.0681} \qquad \textit{(Formula 2)}$$

Note: the value of 0.0681 is given by the Phidgets user manual.

The Vdiff value represents the voltage generated by the solar panel and therefore the illumination present.

3.2.6 CURRENT SENSOR

The amount of current flowing into each of the two motors is monitored by two separate ACS712 low current sensing breakout boards. The low current sensors are capable of translating a differential current of ±5 Amps into an analog signal ranging from 0 to 5 Volts. Chapter 3.2.5 outlines the ADC architecture and formula 1 outlines the steps required to translate the sensed current back into an Analog Voltage Measurement between 0 and 5 volts.

Once the original measured analog voltage is calculated, the following formula is used to translate into a ±5A signal:

$$Adiff = \frac{Analog\ Voltage\ Measure - 2.5}{0.185} \qquad \textit{(Formula 3)}$$

Note: the value of 0.185 is the sensor Sensitivity outlined in the ACS712 datasheet

The Adiff value represents current consumed by a Servo Motor.

3.2.7 FUEL GAUGE

The percent state of charge (SoC) of the onboard Lithium Polymer battery is monitored by the Sparkfun LiPo Fuel Gauge. This breakout board utilizes the MAX17043 integrated circuit and communicates with the EFM32LG microprocessor over Inter-integrated circuit (I2C) protocol. A comprehensive review of this protocol is beyond the scope of this thesis. The Percent State of Charge (SoC) is reported from the fuel gauge to the microprocessor by reading the 16 bit SOC register calculated by the sensor.

The LiPo fuel gauge is also capable of reading the 16 bit Voltage register.

3.2.8 WIRELESS RECEIVER/TRANSMITTER

The system is designed to remain idle until sweep commands are received by the master controller. These wirelessly transmitted commands are sent from an XBee S1 module connected to a laptop computer running a python based Graphic User Interface. The commands are received by a paired XBee S1 module on the system.

The Xbees use UART protocol to serially stream data into the transmit buffer and out of the receive buffers. The data is organized into 8 bit data blocks, where each 8 bits represents a 256 ASCII character.

A basic protocol is used to decode data packets from one another. Data sent to and from the system is formed in the following manner:

!XX@YYYYY&

The "!" character denotes the beginning of a packet, and the "&" character the end. The "XX" value is a 2 digit packet identifier, the "YYYYY" value is a 5 digit payload variable and the "@" character is used as a identifier/value separator. A complete list of communication packets is visualized in the table below.

TABLE 3-2 LIST OF COMMUNICATION PACKETS

Identifier	Packet Name	Generated By:	Description
01	Elevation Current Draw	System	Broadcasts the current draw of the Elevation Motor
02	Azimuth Current Draw	System	Broadcasts the current draw of the Azimuth Motor
03	PV Voltage	System	Broadcasts the voltage on the PV cell
04	System Time	System	Broadcasts the current system time in seconds
05	Battery Voltage	System	Broadcasts the Battery Voltage
06	Battery SOC	System	Broadcasts the Battery SOC
07	State	System	Broadcasts the current system state
11	AZ_Sweep_CW	Master Control	Broadcasts a command to sweep the Azimuth motor in the CW direction
12	AZ_Sweep_CCW	Master Control	Broadcasts a command to sweep the Azimuth motor in the CCW direction
13	EL_Sweep_CW	Master Control	Broadcasts a command to sweep the Elevation motor in the CW direction
14	EL_Sweep_CCW	Master Control	Broadcasts a command to sweep the Elevation motor in the CCW direction
15	Full_Sweep	Master Control	Broadcasts a command to sweep the Elevation and Azimuth motors in a predefined "Full Sweep" pattern.
20	Bright_Detected	Master Control	Broadcasts a system response to MC indicating a Sweep command successfully found a Bright area
21	Sweep_Failure	Master Control	Broadcasts a system response to MC indicating a Sweep command failed to find a Bright area in the requested direction

22	Elevation Motor Failure	System	Indicates to Master Control that the Elevation Motor has failed

Each data packet consists of 10 x 8 data bits + 1 start bit and 1 stop bit, resulting in each command being 100 bits in length. At an 115200 bit per second frequency the end systems are able to decode packets in approximately 0.87 milliseconds.

3.2.9 SERVO MOTORS

The system uses two position controlled servo motors to control the azimuth and elevation of the solar panel, granting 2 degrees of freedom when tracking the sun. A visualization of the azimuth and elevation is seen below:

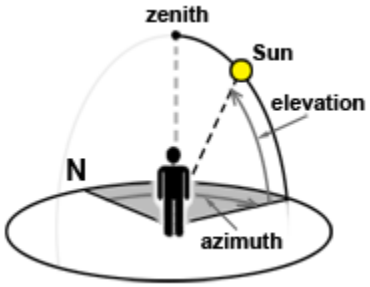


FIGURE 3.2-3 AZIMUTH & ELEVATION DIAGRAM

The motors selected for this implementation are an HS-645MG Servo Motor for the Azimuth direction and a HS-805BB for the Elevation direction. The control the solar panel mounted on a Lynxmotion Large Pan/Tilt kit.

The servo motors operate over a 180 degree range. In this implementation the Elevation servo motor is limited to ± 45 degrees, allowing for a 90 degree range. The initial positions of the solar panels are 90 degrees azimuth (center) and 45 degrees Elevation (fully CW).

The servo motor moves its armature to a position specified by the demand signal. The position of the armature is measured by a potentiometer connected to the motor shaft and is used by the control system to determine whether the controlled position has been reached. The motor will remain energized until the potentiometer value associated to the commanded position has been reached. If an obstruction prevents the motor from

moving to the commanded position, the motor will remain energized, causing an increase in current draw and potential damage to the system components.

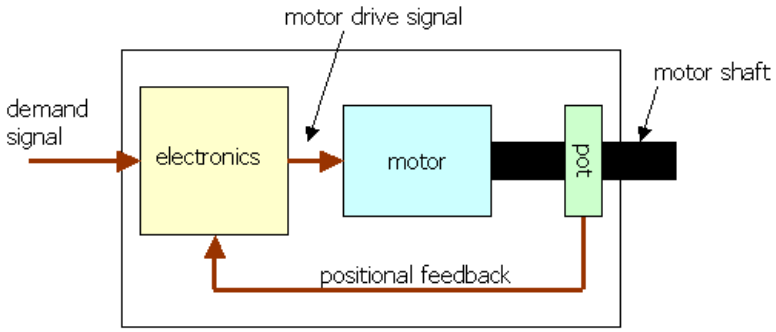


FIGURE 3.2-4 SERVO MOTOR BLOCK DIAGRAM

The demand signal associated to a particular position is controlled by varying a 2ms gap that is periodic on 20ms. By varying the amount of time the 2ms signal is left high, users are able to command the motor to a particular position. For example, if the demand signal is high for 1ms, every 20ms, the position will be commanded to the 0 degree position. If the demand signal is high for 2ms, every 20ms, the position will be commanded to the 180 degree position. A visualization of the demand signal is visualized in the figure below:

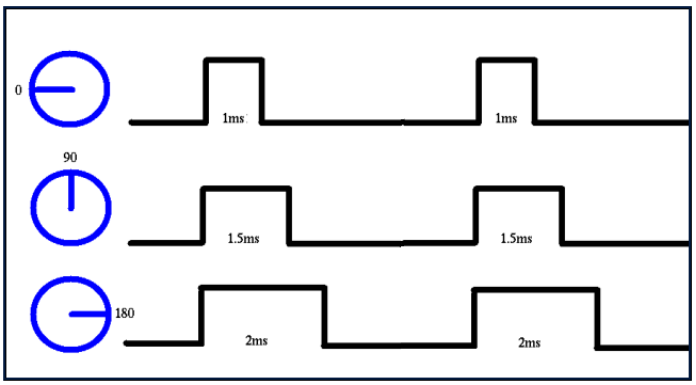


FIGURE 3.2-5 SERVO MOTOR DEMAND SIGNAL

The Azimuth and Elevation demand signals are controlled via pulse width modulation. For this implementation, Timer1 of the EFM32LG is used.

The motors are capable of moving extremely fast; the azimuth motor is capable of moving 60 degrees in 200 milliseconds and the elevation motor is capable of moving 60 degrees in 140 milliseconds. Limiting the speed at which these motors are capable of

moving is essential to ensure the solar panel is not damaged when sweep commands are issued. For this reason, the motor commands are limited to 2 degree increments every 2 seconds. This time delay allows sufficient time for the motors to move position and for current measurements to be performed on the motor power inputs to ensure obstructions are not encountered.

When the servo motors are commanded to reposition, the motor electronics generate drive signals periodic on 20 milliseconds. The current sensor will therefore detect an increase in current draw every 20ms lasting approximately 1 – 1.5ms. When the motor successfully reaches the commanded position the current draw returns to the idle value. This can be visualized in the figure below:

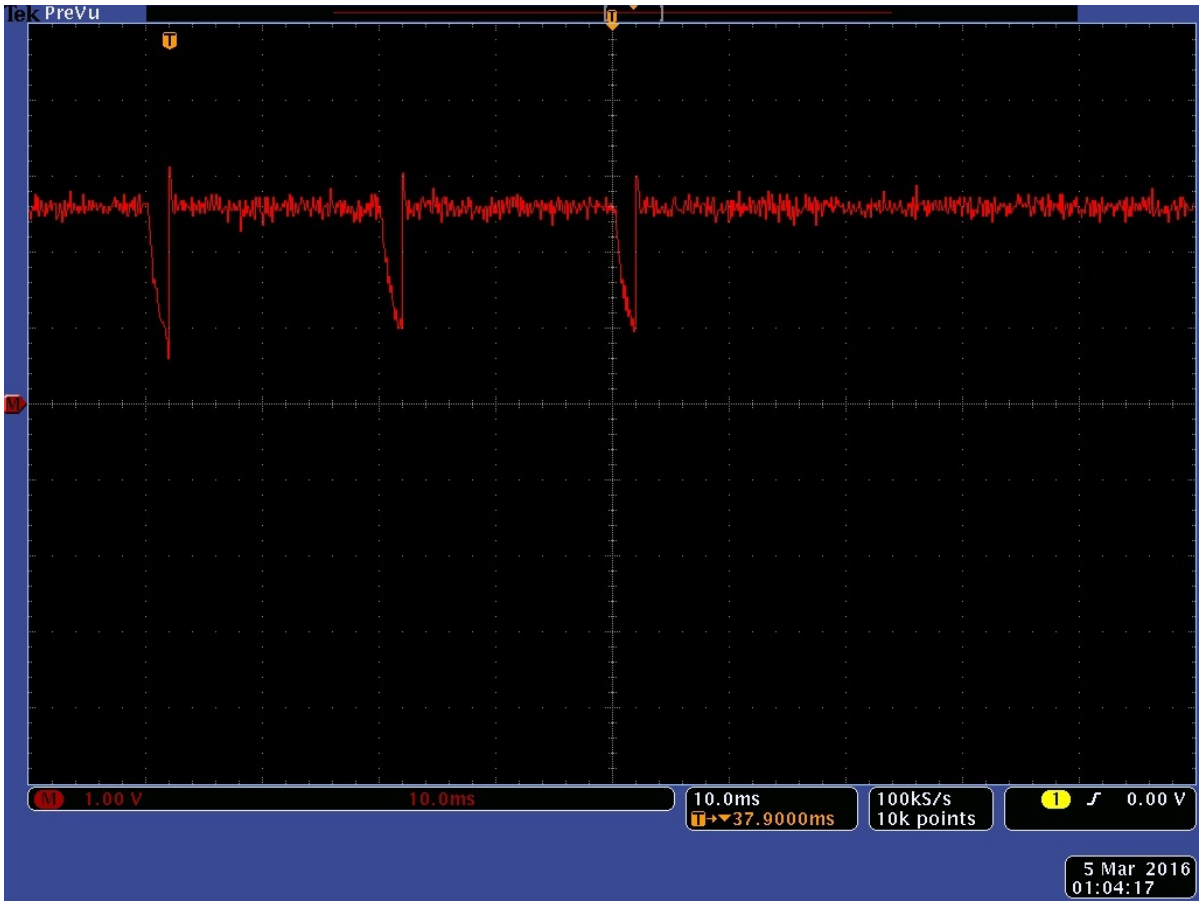


FIGURE 3.2-6 VOLTAGE MEASURED FROM THE CURRENT SENSOR OUTPUT - SUCCESSFUL MOVE COMMAND. X AXIS = 10 MS PER DIVISION, Y AXIS = 1 VOLT PER DIVISION

In the figure above, an oscilloscope is used to measure the voltage output of the current sensor A2 during a move command. At steady state the output voltage is 2.5V and drops

as current is detected by the sensor. The servo motor electronics create new motor drive commands every 20ms and a corresponding current spike is detected by the sensor. When the motor reaches the commanded position the current returns and remains at 2.5 volts.

It's important to note that graph above depicts the current transduced by the sensor before it is sampled by the microcontroller's analog to digital converter. In order to adequately digitize the 20ms (50hz) current pulses, the analog to digital converter must sample the current sensor every 10ms (100hz). The sample period must be equal to half of the pulse width to ensure it is adequately captured. The ADC on the selected microprocessor is capable of sampling up to 1 million samples per second, and is therefore up to the task when correctly configured.

If the motor fails to reach the command position, the servo motor continuously drives the motor output and high current will be detected by the sensor. The current will remain high for almost the entire 20ms period and will be re-commanded every 20ms. A visualization of this is seen below:

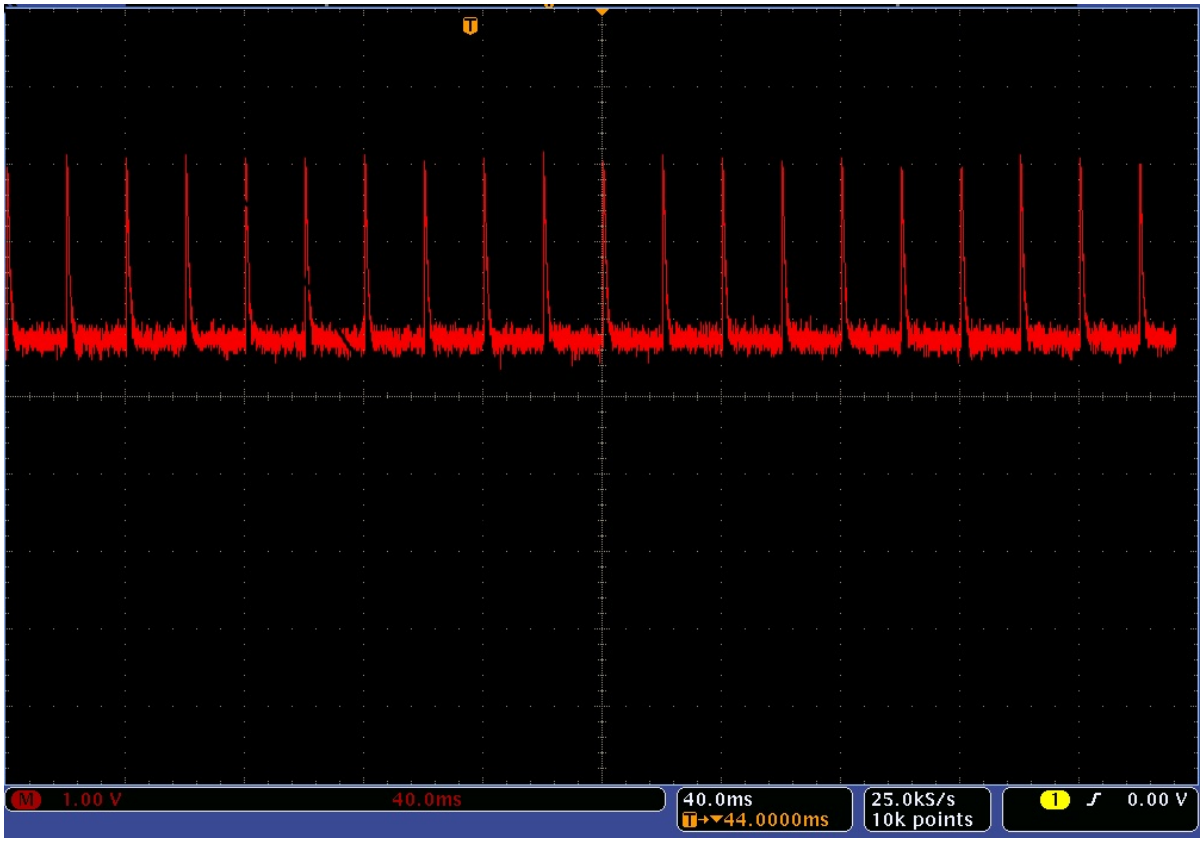


FIGURE 3.2-7 VOLTAGE MEASURED FROM THE CURRENT SENSOR OUTPUT - OBSTRUCTED MOVE X AXIS = 40 MS PER DIVISION, Y AXIS = 1 VOLT PER DIVISION

The current measured by the sensor is nearly continuously high when an obstruction prevents a motor from reaching its commanded position.

3.3 SYSTEM MODELLING

All quantities that require monitoring or controlling must be abstracted in a model. In many cases, the system objectives are used to determine which system components need modelling. In this implementation, the dual axis solar tracking system is broken down into component, interaction and specification models. Chapter 4 discusses the creation of the supervisor and its corresponding software implementation.

3.3.1 COMPONENT MODELLING

Discrete components such as switches are simple to model, as the states required to fully describe their functionality is typically straight forward. For analog quantities, translating their quantities into a discrete model is more challenging. In [3] the example

of tank being filled with water until a specific level is reached is illustrated. The author correctly points out that by modelling the events by simply adding 1 unit a time would lead to extreme inefficiency. Designers must remain aware that the growth of a model's state space is exponential and that adding unnecessary states to a model will ultimately lead to state explosion. The model must therefore be granular enough to adequately define the behaviour of the component while trim enough to avoid unnecessary states.

The Chapters that follow outline the rationale used to model the system components into a discrete event system that adequately satisfies the system objectives and includes some margin for future growth.

3.3.1.1 PHOTOVOLTAIC (PV) CELL MODELLING

The system objectives state that the system must seek out a "Bright" level on the solar panel. The brightness present on the solar panel is monitored by the analog voltage sensor V1, and is limited to a range from -30 to +30 Volts. The voltage range is decomposed into three distinct brightness levels: "Dark", "Dim" and "Bright"; each represented one of three distinct states in the PV cell model. The value measured by the voltage sensor is sampled at 4 Hz and is used to determine if the "Dark_to_Dim", "Dim_to_Bright", "Dim_to_Dark" or "Bright_to_Dim" events should be triggered. The PV illumination model is visualized in Figure 3.3-1.

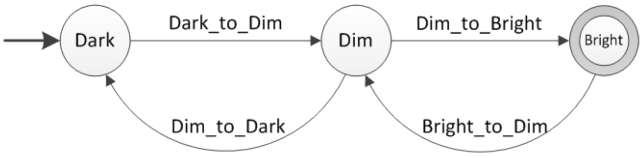


FIGURE 3.3-1 PV ILLUMINATION MODEL

From the above model, it's clear that a finer decomposition is possible, for example, five or even ten states could be used to capture the brightness present on the solar panel. However, the system has two separate behaviours based on the illumination present on the solar panel. If the system is bright, the system will stop sweeping the motors. If the system is not bright, the system will begin sweeping the motors (if the proper commands are received).

It could easily be argued then, that the usefulness of three separate states is unnecessary. This argument is not false, for this implementation however includes a small degree of growth potential already modelled within the system. This way, a separate behaviour that targets Dim illuminations can easily be designed.

The PV illumination model identifies four separate events. These events are associated to the analog values returned by the voltage sensor V1. Table 3-3 illustrates the triggering values unique to each event.

TABLE 3-3 PV CELL LIST OF EVENTS AND DESCRIPTIONS

Current State	Current Average (Volts)	Event Triggered	Controllable?
Dark	< 6	-	-
Dark	> 6	Dark_to_Dim	No
Dim	< 5	Dim_to_Dark	No
Dim	> 16	Dim_to_Bright	No
Bright	< 15	Bright_to_Dim	No
Bright	>15	-	-

The information illustrated in the table above is shown graphically in Figure 3.3-2.

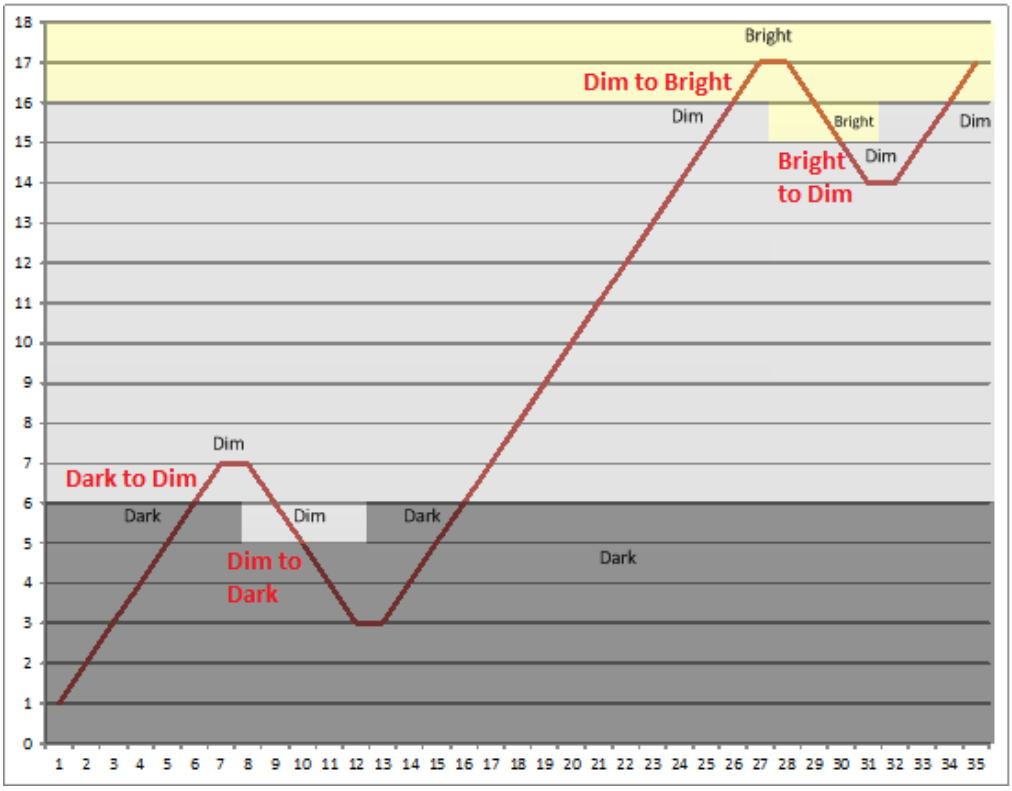


FIGURE 3.3-2 VOLTAGE MAPPING TO ILLUMINATION STATES

The microcontroller polls the analog value 4 times per second (250ms). The analog value must then be above the voltage threshold for at least 250ms to be triggered. When modelling events associated to analog measurements, it's important to enforce hysteresis around the triggering logic values. This way, events will not be continuously strobed when the measured values vary around the boundary values. For example: If no hysteresis is used and the Dark_to_Dim event triggers when the voltage transitions from below 5V to above 5V, and the Dim_to_Dark event triggers when the voltage transitions from above 5V to below 5V. When the measured voltage is 5, the slightest deviation due to signal noise will continuously trigger both events one after the other as the measured value oscillates around the 5V threshold. By setting a value of 1V hysteresis around the trigger levels, the system must detect at least a 1Volt change before the events will occur.

3.3.1.2 BATTERY STATE OF CHARGE

There is no directly observable system objectives associated to the battery state of charge. However, the system has an interaction such that the motors are unable to move when the battery is in the Critical state. For this reason, a minimum abstraction of

"Critical" and "Non-Critical" states is required. For this implementation, a third state is added for future growth potential.

The battery state of charge spans a range of 0 to 100 percent full. The system has observability over the battery state of charge via the Fuel Gauge sensor. The battery's total range is decomposed into states: Critical, Safe and Full.

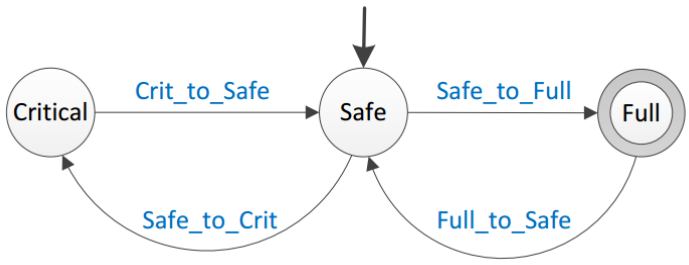


FIGURE 3.3-3 BATTERY STATE OF CHARGE MODEL

The Microprocessor reads and stores the value returned by the Fuel Gauge once every 10 seconds. The frequency can easily be increased, but the battery state of charge does not change quickly enough to warrant a higher frequency.

The Battery State of Charge model identifies four separate events. These events are associated to the battery state of charge percentage returned by the I2C interface of the Fuel Gauge. Table 3-4 illustrates the triggering values unique to each event.

TABLE 3-4 BATTERY STATE OF CHARGE LIST OF EVENTS AND DESCRIPTIONS

Current State	State of Charge (%)	Event Triggered	Controllable?
Critical	> 55	Critical_to_Safe	No
Safe	< 50	Safe_to_Critical	No
Safe	> 95	Safe_to_Full	No
Full	< 90	Full_to_Safe	No

If the percent state of charge state is "Safe" and the average Percent State of Charge is greater than 95 the Safe to Full event is triggered and the Battery State of Charge model is transitioned to the "Full" state. A hysteresis value of 5 percent is used around the boundary values to prevent event events from strobing around the set point. .

3.3.1.3 MOTOR MODELLING

The motors have two main properties that must be modelled: Range and Motion. Once again, the modelling criteria are determined by analyzing the system objectives. For the motors, they state that the system will stop sweeping when the maximum range of the commanded direction has been reached. Additionally, the Elevation motor must be able to detect whether an obstruction has been encountered while performing a move command. At a minimum these two properties must be modelled within the system.

3.3.1.3.1 MOTOR MOTION

The system limits move commands to increments of 2 degrees in order to limit the speed at which move commands are performed. This reduction in speed prevents damage of the solar panel mounted on the pan/tilt assembly.

The system generates move commands that reposition the azimuth and elevation motors. When the move command has been triggered, the system will monitor that motor's current consumption in order to determine whether an obstruction has occurred. For brevity, this implementation, only considers the Elevation motor to be fault capable, the Azimuth motors are considered fault free (cannot be obstructed). The Motor Motion Component model for Azimuth motor is visualized below:

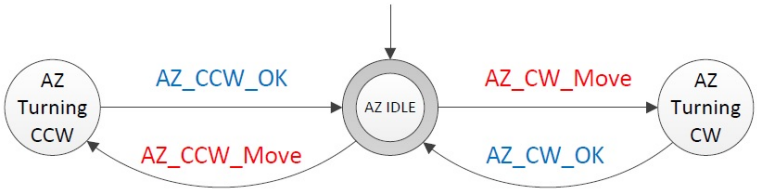


FIGURE 3.3-4 AZIMUTH MOTOR MOTION COMPONENT MODEL

The model above has three states that describe the state of the Azimuth motor and initializes in the idle state. When the motor is idle, the system can generate a controllable CCW or CW move command that begins moving the motor in the instructed direction. The model will then transition to the Turning CCW or CW state, depending on the generated move command. The system remains in the turning state until the uncontrollable CCW or CW_OK event is generated. This event uses the value measured by the A2 current sensor that monitors the current consumption of the

Azimuth motor. The behaviour of the current consumption during motor movements was discussed in Chapter 3.2.9.

In this implementation, the current consumed by each servo motor is averaged over a 2 second time frame after a move command has been issued. If the average current is less than 0.5 amps, the move command is declared to be successful and the _OK event matching the commanded direction will be triggered.

TABLE 3-5 AZIMUTH MOTOR MOTION EVENT LIST

Current State	Current (mA)	Event Name	Controllable?
AZ IDLE	N/A	AZ_CW_MOVE	Yes
AZ IDLE	N/A	AZ_CCW_MOVE	Yes
AZ Turning CW	500 > AZ_CW_OK > 100	AZ_CW_OK	No
AZ Turning CCW	500 > AZ_CCW_OK > 100	AZ_CCW_OK	No

The azimuth motor motion is assumed to be fault free in this implementation. However, the elevation servo motor may become obstructed during a motor move command. The system objectives specify that commands to reposition the elevation motor must be ignored if the motor becomes obstructed (failed). For this reason, a fourth state named "EL Failed" must be modelled into the Elevation Motor Motion Component model:

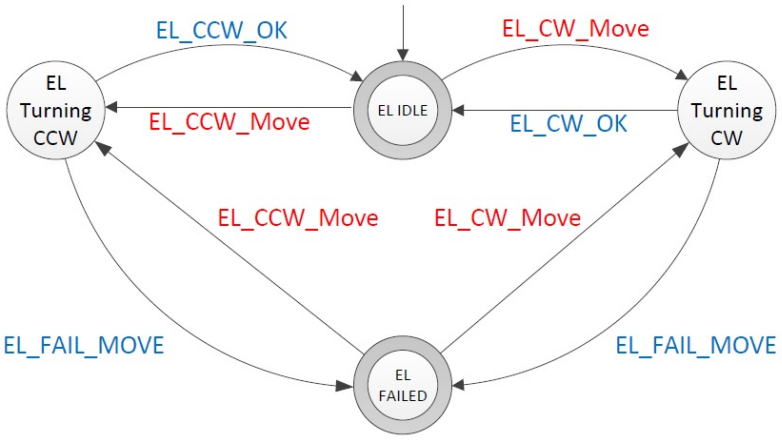


FIGURE 3.3-5 ELEVATION MOTOR MOTION COMPONENT MODEL

If the current sensor measures an average current greater than or equal to 0.5 amps over a 2 second time frame, the elevation sensor is determined to be stuck and the

EL_FAIL_MOVE command will be triggered. The system objectives assume that obstructions are permanent in nature and cannot be removed. However, this abstraction is a specification, not a representation of the component models. For this reason, the component model above must allow the possibility of move commands to be generated while in the failed state. Chapter 3.3.3.3.3 limits the behaviour of the system to assume failures are permanent.

TABLE 3-6 ELEVATION MOTOR MOTION EVENT LIST

Current State	Current (mA)	Event Name	Controllable?
EL IDLE	N/A	EL_CW_MOVE	Yes
EL IDLE	N/A	EL_CCW_MOVE	Yes
EL Turning CW	500 > EL_CW_OK > 100	EL_CW_OK	No
EL Turning CW	>= 500	EL_FAIL_MOVE	No
EL Turning CCW	500 > EL_CCW_OK > 100	EL_CCW_OK	No
EL Turning CCW	>= 500	EL_FAIL_MOVE	No

The table above illustrates the possible elevation motor motion events.

3.3.1.3.2 MOTOR RANGE

The servo motors are restricted to a 180 degree range of motion with initial positions of 90 degrees azimuth (center) and 45 degrees Elevation (max CW). The current position of each servo motor is monitored by dedicated variables named “AZ_Current_Range” and “EL_Current_Range”. These variables are stored within the microcontroller’s onboard memory and are incremented by 2 degrees each time a “XX_CW_OK” event is triggered and decremented when a “XX_CCW_OK” event is triggered. This process is done automatically in the microcontroller’s code and therefore does not need to be modelled at a DES level.

The range variables are polled when the the controllable “AZ_POLL_RANGE” or “EL_POLL_RANGE” events are triggered. The system will then respond by triggering the “XX_IN_RANGE”, “XX_MAX_CW”, or “XX_MAX_CCW” events. A visualization of this is seen in the figure below:

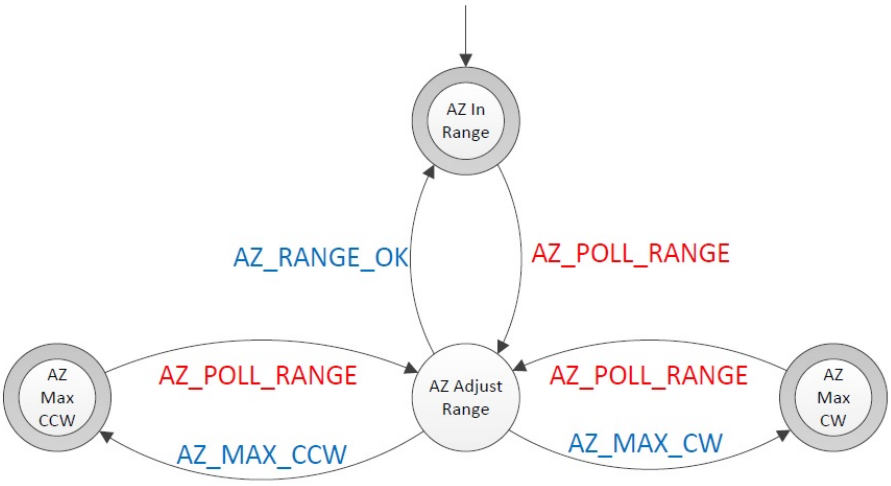


FIGURE 3.3-6 MOTOR RANGE COMPONENT MODEL

The Elevation motor range component model is nearly identical to the figure above and is not shown for brevity. The only difference is that the Elevation motor range begins in the AZ Max CW state. The table below summarizes the 8 events unique to the Motor Range Component Models.

TABLE 3-7 MOTOR RANGE EVENT LIST

Current State	Range (Degrees)	Event Name	Controllable?
AZ_In_Range, AZ_Max_CW, AZ_Max_CCW	N/A	AZ_POLL_RANGE	Yes
EL_In_Range, EL_Max_CW, EL_Max_CCW	N/A	EL_POLL_RANGE	Yes
AZ_Adjust_Range	= 0	AZ_MAX_CW	No
AZ_Adjust_Range	180 > RANGE > 0	AZ_In_Range	No
AZ_Adjust_Range	= 180	AZ_MAX_CCW	No
EL_Adjust_Range	= 0	EL_MAX_CW	No
EL_Adjust_Range	45 > RANGE > -45	EL_In_Range	No
EL_Adjust_Range	= 180	EL_MAX_CCW	No

3.3.1.4 MASTER CONTROL

The master controller sends and receives control commands via the Wireless TX/RX device outlined in Chapter 3.2.8. The component diagram of this component consists of a single state with the control and response events modelled as selfloops. This is visualized in the figure below:

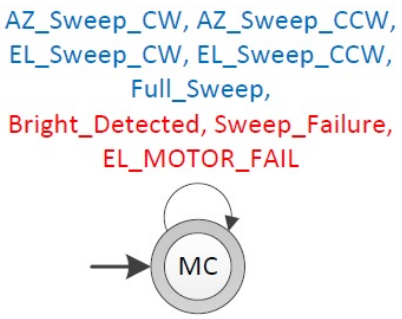


FIGURE 3.3-7 MASTER CONTROL COMPONENT MODEL

The Master Controller is modelled from the viewpoint of the supervisor. When command requests are generated by the Master Controller, they are modelled as uncontrollable events. When the supervisor completes a control request, the responses are broadcasted back to the Master Controller. These are controllable events as they are controlled by the system. A complete list of MC events is tabled below:

TABLE 3-8 MASTER CONTROL EVENT LIST

Current State	Event Name	Controllable?
MC	AZ_Sweep_CW	No
MC	AZ_Sweep_CCW	No
MC	EL_Sweep_CW	No
MC	EL_Sweep_CCW	No
MC	Full_Sweep	No
MC	Bright_Detected	Yes
MC	Sweep_Failure	Yes
MC	EL_MOTOR_FAIL	Yes

The “Full_Sweep” uncontrollable event will trigger the system to begin commanding its motors to sweep in a pattern defined by the specifications. The system may also inform the Master Controllable of certain key events. The “Bright_Detected” event is a controllable event used to inform the MC that a bright area was found while performing a sweep command. The “Sweep_Failure” event is a controllable event used to inform the MC that no bright areas were detected during the entirety of the sweep command. Finally, the “EL_MOTOR_FAIL” event is a controllable event used to inform MC that the elevation motor has encountered an obstacle during its operation.

3.3.2 SYSTEM INTERACTIONS

Interactions reduce the system plant size by removing state transitions that are impossible to occur due to physical limitations. For example, the battery state of charge cannot increase if no sunlight is present on the solar panels. This is a limitation of the system based on the interaction between the components.

The modeling of these interactions is based on the physical interfaces and interactions between components modelled in the previous Chapter. In order to obtain accurate interaction models, a detailed understanding of the system’s functionality is required. It is critical that interactions be completely accurate in all cases. Removal of a potential state transition will lead to an incorrect plant model, resulting in a plant model that does not match the physical implementation.

Interactions are generally expressed as the events of one model being a function of the states of another. Interactions are modelled by adding the events of a model as self-loops on the states of another. The interactions of this implementation are detailed in the Chapters that follow.

3.3.2.1 BATTERY SOC AND PV CELL ILLUMINATION

The events generated by the Battery State of Charge model are directly related to the amount of brightness present on the solar panels. The events associated to the battery going up in charge will never occur if the PV cell is in the “Dark” state. The maximum voltage the solar panels can produce while in the Dark state is 6 Volts, resulting in a maximum potential wattage of approximately 1 Watt. This power input is insufficient to power the system and charge the batteries. For this reason, the events associated to charging the battery are impossible to trigger when operating in the Dark state. If the PV

cell is Dim or Bright, there is no restriction to the Battery SOC events. This is seen visually in the figure below:

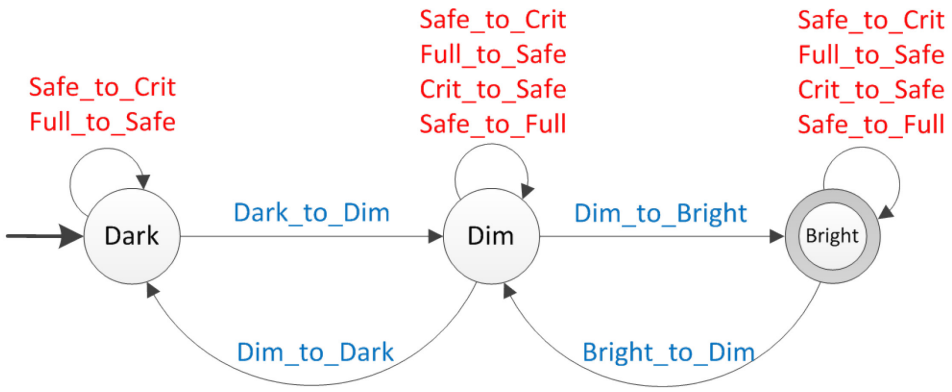


FIGURE 3.3-8 BATTERY SOC AS A FUNCTION OF PV CELL STATES

By omitting the Crit_to_Safe and Safe_to_Full events from the self-loop on the Dark state, the combination will be removed when the synchronous product is performed.

3.3.2.2 MOTOR MOTION AND BATTERY SOC

The onboard battery selected for this implementation is a single cell Lithium Polymer battery with a nominal voltage of 3.7 Volts. This type of battery has a specific battery discharge pattern that is used by the Fuel Gauge to determine the remaining state of charge within the battery. A sample Lithium Polymer battery discharge curve is illustrated in the figure below:

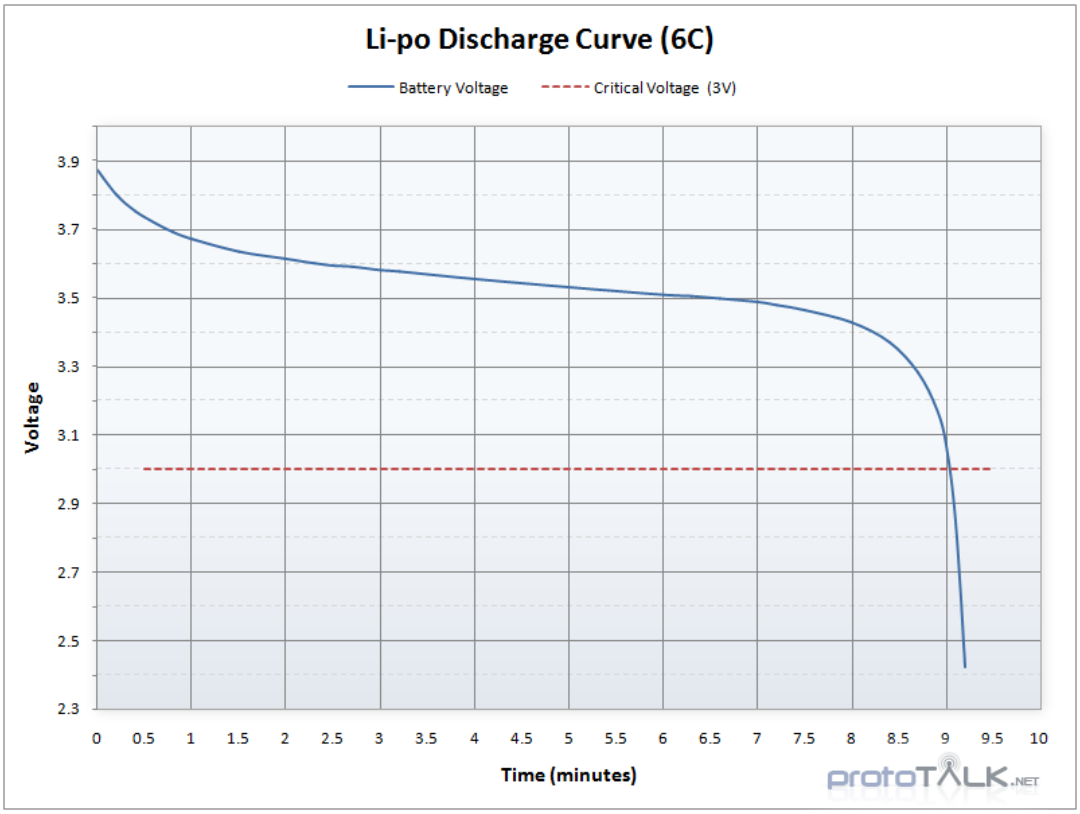


FIGURE 3.3-9 SAMPLE LIPO BATTERY DISCHARGE CURVE [18]

The red dashed line in the figure above is the critical point of the battery. Most LiPo batteries have a protection circuit that cuts off the voltage output if the battery drops below a particular level (typically 3Volts). If the LiPo battery shuts down, the entire system becomes lifeless. Modelling this interaction is therefore useless as the system would never have visibility over it (since the system would be offline!).

However, there is a point on the graph where the power output delivered by the batteries becomes insufficient to reposition the motors. This is due to the fact that the power consumption of each motor is greater than the power delivered by the battery. This value is roughly 50% of the battery state of charge. Therefore, there exists an interaction where the motors are unable to move when the battery state of charge is below 50 %. This is visualized in the model below:

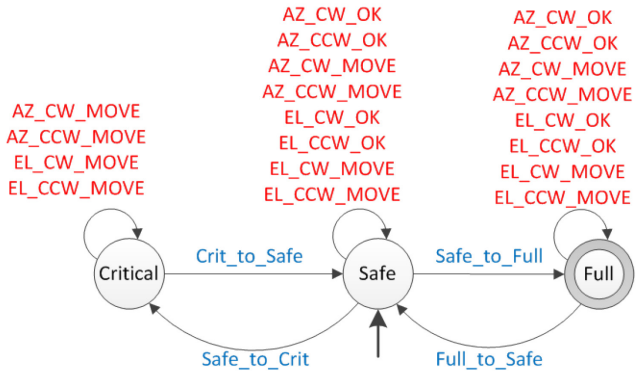


FIGURE 3.3-10 MOTOR MOTION AS A FUNCTION OF BATTERY SOC

When the system is in the Critical Battery SOC state, the system is capable of initiating Move commands, but will never receive the correct _OK commands as the current values will never be reached. Table 3-4 and Table 3-6 outline the triggering conditions of the _OK commands. Both commands require a minimum value of 100 mA over a 2 second time period in order to successfully consider the move motion as successful. If the motors are in the critical state, this value will never be reached and the _OK events will never trigger. For this reason, they can be safely removed from the plant.

3.3.2.3 BATTERY SOC AND MOTOR MOTION

When the motors are moving they consume a substantial amount of power. The Elevation motor requires 700mA and the Azimuth motor requires 350mA. These values are much larger than the 200mA maximum current input of the Solar panel. For this reason, when either of the motors are moving, the battery can only be discharging.

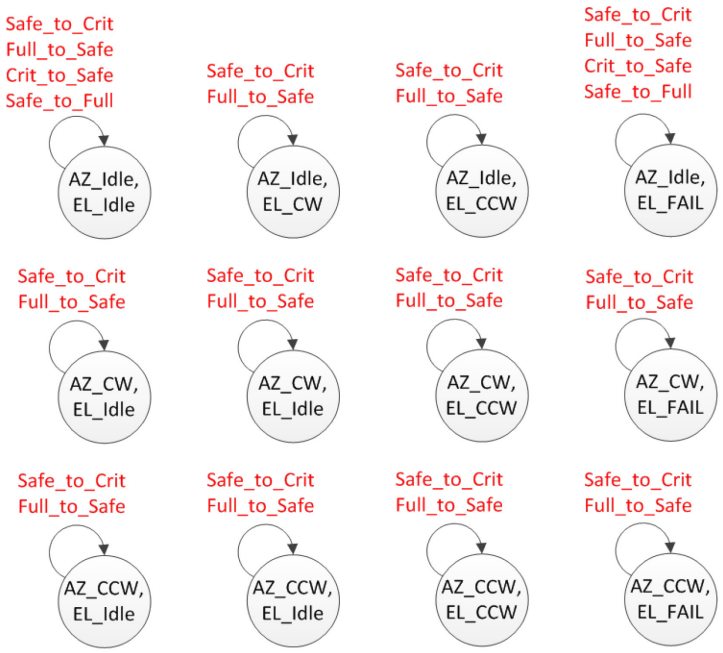


FIGURE 3.3-11 BATTERY SOC AS A FUNCTION OF MOTOR MOTIONS

The figure above creates a synchronous product of the two Motor Motion component models outlined in 3.3.1.3.1. If either of the motors are in a moving state, the only events of the battery SOC that can occur as the ones associated with discharging the battery. If the motors are both idle, or if the Azimuth motor is idle and the Elevation motor is failed, the battery may either be charging or discharging.

3.3.3 SYSTEM SPECIFICATIONS

Specifications allow designers to tailor the behaviour of the system to meet the system objectives. This Chapter details the method used to ensure that the behaviour of the models are limited to a specific manner. For example, the motors must never move in the clockwise direction when the motor range is in the maximum clockwise state.

Additionally, we specify a specific set of movement commands that must be generated when the associated command signals are received from the master controller.

In total, a set of four modular motor control specifications and one monolithic movement specification are created for this implementation. For the purposes of this thesis, three separate implementations were created. The first creates guidelines to accommodate individual sweep commands and does not include motor failure events. The second creates guidelines to perform a pre-determined movement pattern without

motor failure events; this is referred to as a full sweep. The final implementation creates guidelines for a full sweep command with motor failure events.

Each of these implementations is separate from each other (a unique specification automaton was generated for each).

3.3.3.1 CONTROLLING MOTOR MOTION BASED ON RANGE

The servo motors operate on fixed range. When they reach their maximum range values, additional move commands may still be received to further position the motor in that direction. This will cause damage to the motor and will ultimately lead to component failure. A specification must therefore be put in place *to prevent the motors from moving further in the clockwise (respectively counter-clockwise) direction when the motor range is in the maximum clockwise (respectively counter-clockwise) state*. To accomplish this, a similar process to the one outlined in 3.3.2 is used.

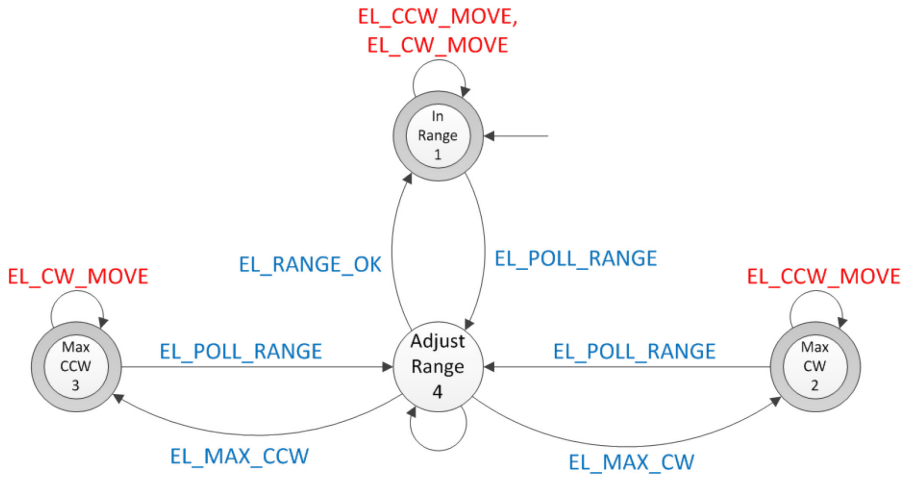


FIGURE 3.3-12 MOTOR MOTION LIMITATIONS BASED ON MOTOR RANGE

The modelling process is identical to the process used in 3.3.2. This model takes the Motor Range component model seen in 3.3.1.3.2 and self-loops the events defined in the Motor Motion component model. By omitting "EL_CCW_MOVE" from the "MAX CCW" state, the supervisor will disable that event from being generated in that state.

Note: Both Azimuth and Elevation control specifications are identical. Only the Elevation is shown for brevity.

3.3.3.2 MOTOR RANGE POLLING AND MOTOR MOTION

The motor range events allow the designers to trigger a "XX_POLL_RANGE" event, which will in turn request the plant to generate corresponding event that informs the system of the current range.

In Chapter 3.3.1.3.2 states that the range value of the servo motors are tracked by an internal software variable. This variable is only updated when a corresponding "_OK" event has been triggered. For this reason, *polling of the servo range must not be performed unless the motors are idle.*

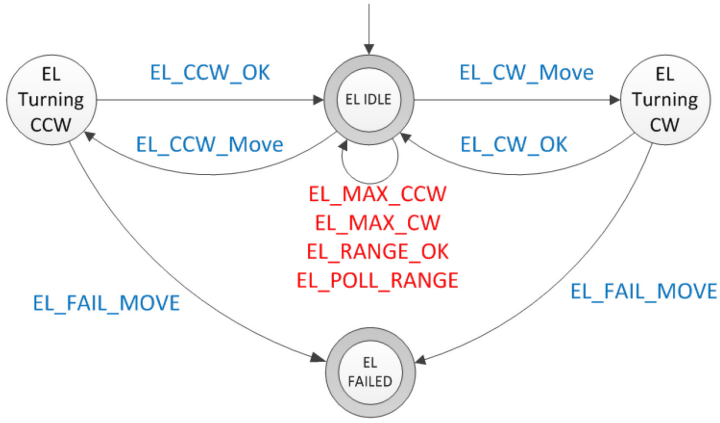


FIGURE 3.3-13 MOTOR RANGE POLLING LIMITATIONS BASED ON MOTOR MOTION

Range response events are added as self-loops to ensure that the motors never trigger a move command until both a poll and response event have been triggered.

Note: Both Azimuth and Elevation control specifications are identical. Only the Elevation is shown for brevity.

3.3.3.3 MOVEMENT PATTERN SPECIFICATIONS (SWEEP COMMANDS)

The system objectives clearly outline the behaviour the system must follow. The system will not command the motors to move unless a Sweep command from the Master Controller has been received. There are five types of sweep commands:

- Azimuth Sweep, Clockwise
- Azimuth Sweep, Counter Clockwise
- Elevation Sweep, Clockwise

- Elevation Sweep, Counter Clockwise
- Full Sweep

The Sweep commands will signal the motors to begin turning in the desired directions until the amount of sunlight present on the panels is sufficient to transition PV Cell model to the “Bright” state. If the Sweep command successfully finds a Bright point the “Bright_Detected” event is broadcast to the Master Controller. If the motor reaches the maximum position without detecting a bright spot the supervisor will broadcast the “Sweep_Failure” event to the master controller.

If the solar panel is pointed at a bright location when a sweep command arrives, the supervisor sends a “Bright_Detected” event without moving position. If a motor is at its maximum range value and a sweep command for that direction arrives, the motors will not move and the supervisor will send a “Bright_Detected” event if the system is in a bright state or a “Sweep_Failure" event if the system is in a dim or dark state.

When a Full Sweep event is received the supervisor will command the motors to the initial position of fully counter clockwise in both Azimuth and Elevation directions. It will then sweep the 180 degree azimuth range in a clockwise motion. When the azimuth motor is fully clockwise the elevation motor sweeps from fully counter clockwise to fully clockwise (a 90 degree range). Finally, the azimuth motor will sweep the 180 degree range in a counter clockwise motion. The resulting motion combinations provide a full hemispherical sweep of the ambient brightness. If a bright event is detected during the sweep process, the supervisor broadcasts the “Bright_Detected” event to the master controller. If the Full Sweep terminates without finding a bright spot, the supervisor broadcasts the “Sweep_Failure” event to the master controller.

The supervisor will ignore all Sweep commands when executing a sweep command.

In order to prevent voltage sag and unnecessary strain on the battery, only one motor may be moving at a time. Each of these criteria is addressed in the Chapters that follow.

3.3.3.3.1 INDIVIDUAL SWEEP COMMANDS

The system receives Sweep clockwise and counter-clockwise commands for Azimuth and Elevation motors. The figure below outlines the implementation of the Azimuth Sweep Clockwise sweep command response.

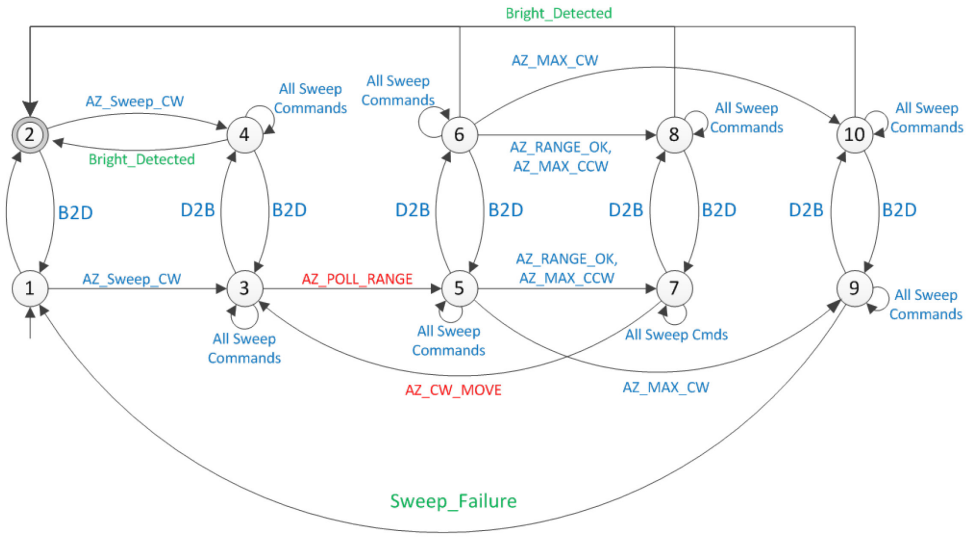


FIGURE 3.3-14 AZIMUTH SWEEP CLOCKWISE SPECIFICATION

In the figure above, the acronyms "B2D" and "D2B" refers "Bright_to_Dim" and "Dim_to_Bright" respectively. The specification begins in state 1, if a bright to dim event is triggered it will transition to state 2. All odd numbered states refer to "Dim or Dark" states on the PV cell and all even numbered states refer to a Bright value on the PV cell.

When the AZ_Sweep_CW event is triggered, the system will begin ignoring all other sweep commands until the current sweep has been completed. The specification states that the system must first poll the range and depending on the system response, will either continue moving the motor clockwise, or, transition to state 9 if the motor has reached the maximum clockwise range. If the maximum range was reached, the system generates the "Sweep_Failure" command and transitions back to state 1 for additional sweep commands.

If the PV cell goes bright at any time, the "D2B" event will trigger and the system will trigger the "Bright_Detected" controllable event. From this point, the system will transit back to state 2 where it awaits for additional commands.

Using a symmetrical pattern, the Azimuth CCW Sweep command is easily integrated into the existing movement pattern specification:

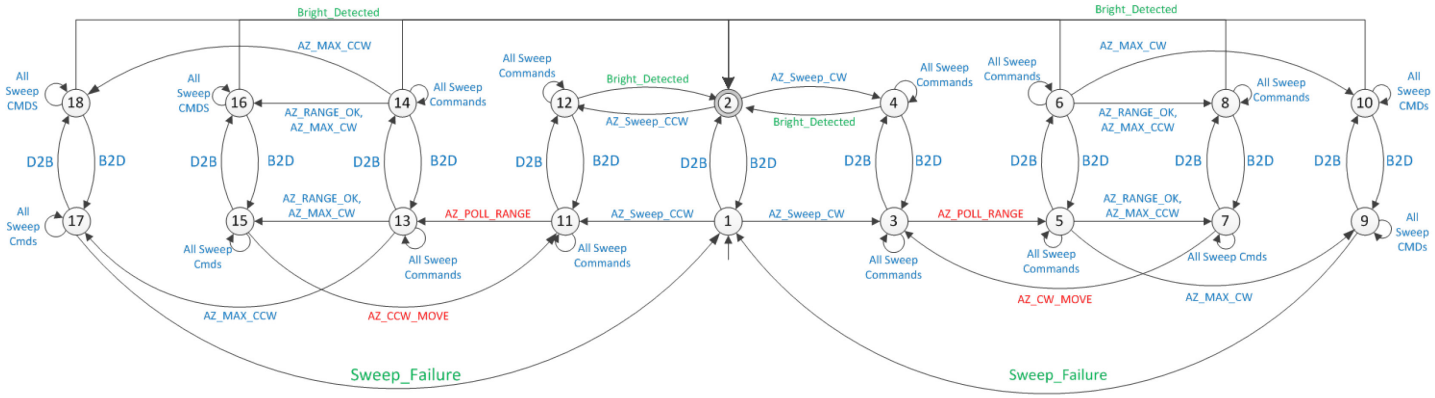


FIGURE 3.3-15 AZIMUTH SWEEP CW AND CCW SPECIFICATION

Since the behaviour of each sweep command is independent from one another, it's possible for a module and systematic approach to be used in the development of the movement specifications.

The addition of the Elevation Sweep CW and CCW specifications are performed in a similar manner and are not shown for brevity.

3.3.3.3.2 FULL SWEEP COMMAND

Along with the individual sweep commands, the system has the ability to initiate a pre-determined set of movements that will grant the solar panel a complete hemispherical sweep of the surrounding area. When a full sweep command arrives, the system will perform the following:

1. Position the Azimuth and Elevation motors to their maximum CCW positions.
2. Sweep from Azimuth Maximum CCW to Azimuth Maximum CW.
3. Sweep from Elevation Maximum CCW to Elevation Maximum CW.
4. Sweep from Azimuth Maximum CW to Azimuth Maximum CCW.

The system will stop and inform the MC if a Bright area is detected. If at the end of the Full Sweep there was no bright area found, the system informs MC with a Sweep Failure command.

For the purposes of this Chapter, the Elevation Motor Failure is not considered. This is discussed in the Chapter 3.3.3.3.3.

The model is quite large and is broken down into smaller more manageable chunks for readability. The figure below outlines the procedure used to position the Azimuth and Elevation motors to their Maximum CCW positions:

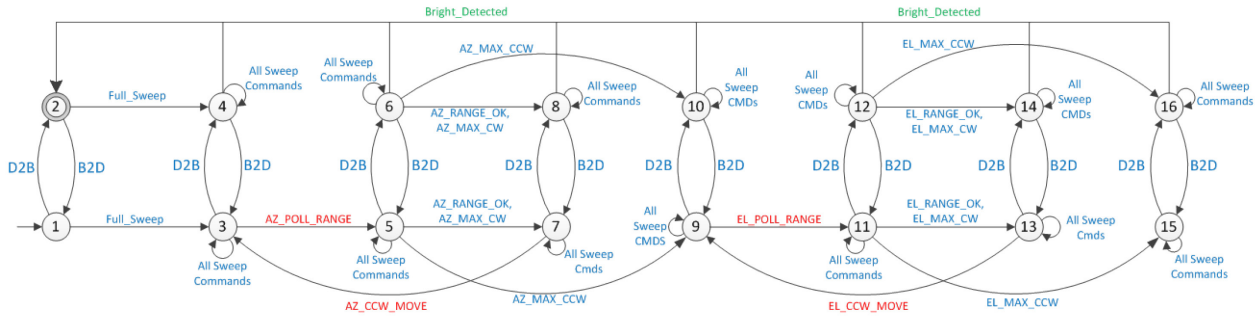


FIGURE 3.3-16 FULL SWEEP SPECIFICATION – SETTING UP INITIAL CONDITIONS

Similar to the Individual Sweep specifications, the even states represent "Bright" conditions and the odd states represent "Dim or Dark" conditions. When the system reaches state 15 of the specification the azimuth and elevation motors are both in the maximum CCW state.

From state 15 onward, the system begins steps 2 – 4 of the list outlined above.

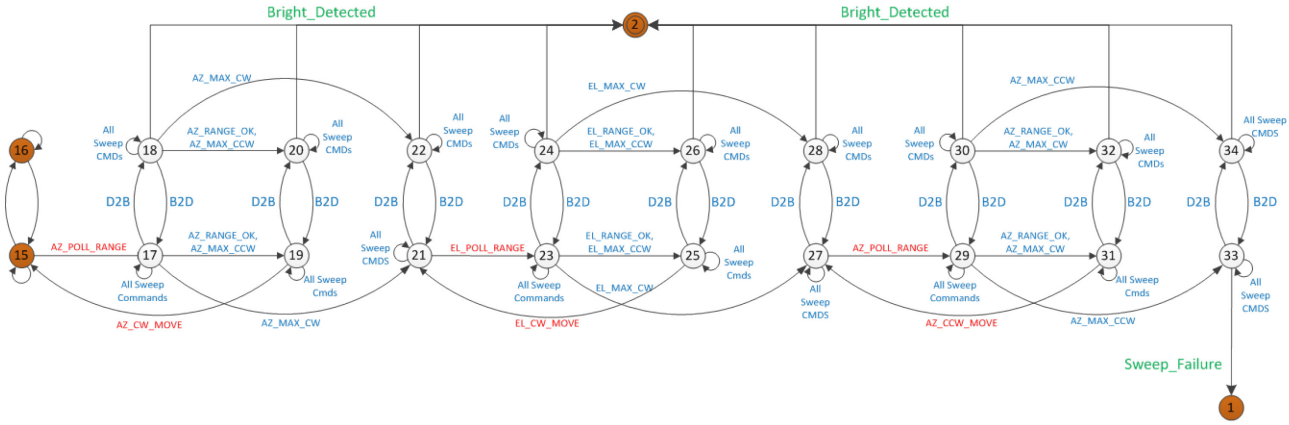


FIGURE 3.3-17 FULL SWEEP SPECIFICATION – HEMISPHERICAL MOVEMENT PATTERN

Note: The orange states represent states already defined in Figure 3.3-16.

3.3.3.3.3 FULL SWEEP COMMAND - CONSIDERING MOTOR FAILURE

The full Sweep outlined the previous Chapter can be expanded to handle Elevation motor failure. In the Elevation Motor Motion component model defined in 3.3.1.3.1, the determination of whether a motor failure has occurred follows after a motor movement has been issued. The Specification is therefore expanded to specify the behaviour that must be followed when either a _OK or a _FAIL_MOVE event has occurred.

As the entire specification is quite large, only a small Chapter has been outlined in the figure below.

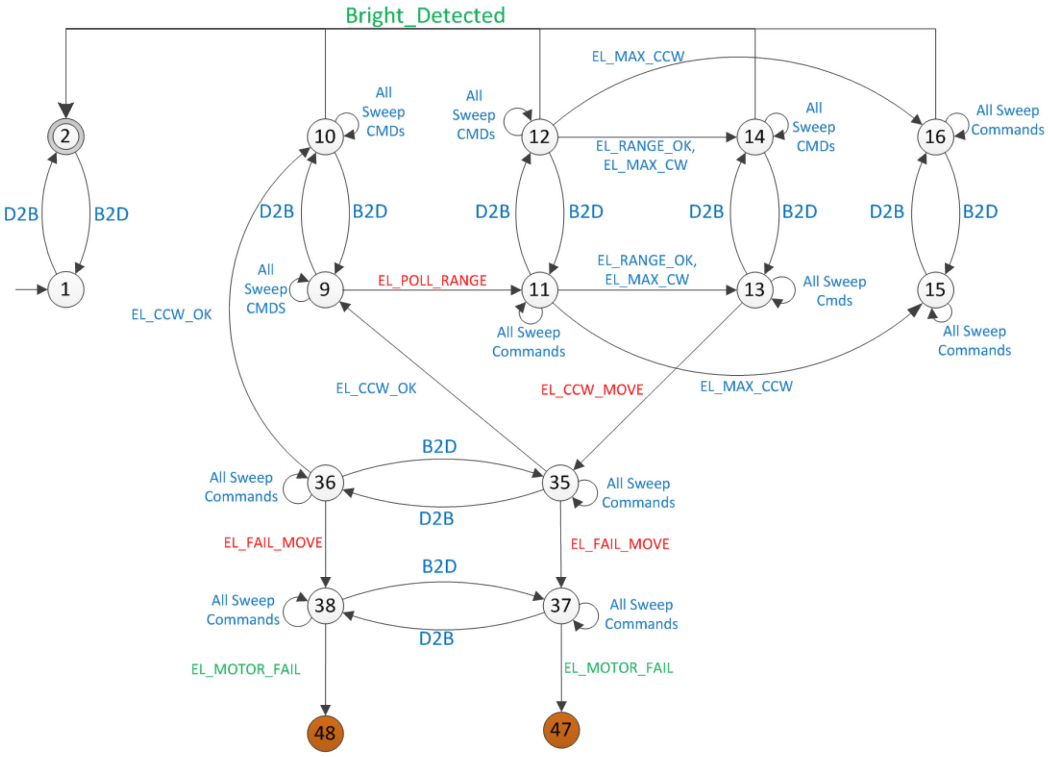


FIGURE 3.3-18 EXPANSION OF FULL SWEEP SPECIFICATION TO INCLUDE MOTOR FAILURE

It should be noted that the _OK and _FAIL_MOVE events are both uncontrollable. When modelling the behaviour of multiple uncontrollable events it's necessary to consider all possible combinations from the point the events may occur. For example, after a move command is issued, the _OK or _FAIL_MOVE events may trigger. However, it's equally possible that the uncontrollable events of Bright_to_Dim or Dim_to_Bright may trigger before (or after) the motor events. For this reason, all four

possible combinations must be included in the specification, or in its own dedicated specification.

Note: An elevation motor failure may also occur when after triggering the "EL_CW_MOVE" event from state 25 to 21 of Figure 3.3-17. However, the implementation is nearly identical to that of Figure 3.3-18 and is therefore not discussed.

Once the Motor failure has been detected, the system moves into a subset of the normal behaviour, denoted by the orange states 48 and 47. This is visualized in the figure below:

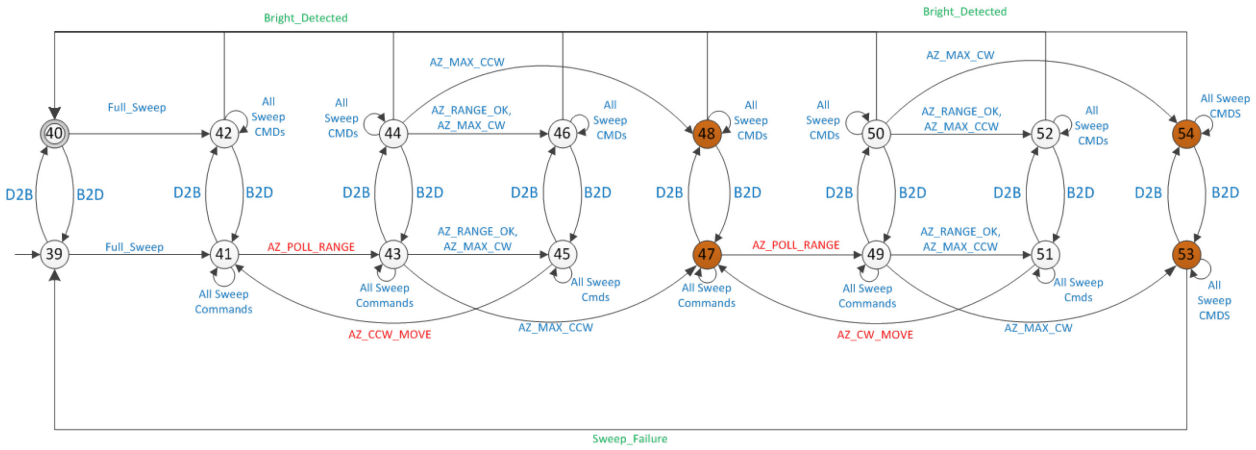


FIGURE 3.3-19 FULL SWEEP SPECIFICATION – AFTER ELEVATION MOTOR FAILURE.

The system will remain in this subset until the system resets. Any future "Full_Sweep" commands will completely ignore the elevation motor and full sweeps will result in a 180 degree sweep in the azimuth direction only.

3.4 CONCLUSIONS

In this chapter, a discrete event model was developed for the (uncontrolled) Dual Axis Solar Tracking System. Also, the design specifications for the supervisor were discussed and modelled as automata. In the next chapter, a supervisor is designed based on the supervisory control theory (SCT) and the software architecture for its implementation will be discussed.

CHAPTER 4. SYSTEM SOFTWARE

This chapter begins by outlining the graphic user interface that functions as the Master Controller for this implementation. Next, a high level view of the "C" based driver code required to initialize the microcontroller peripherals is presented. The software environment used to calculate the supervisory controller of the plant that is based on the specification models is then explored. Once the supervisor has been calculated, it must be efficiently stored within the microcontroller. To achieve this, a novel storage method is designed and explained that allows for both memory efficiency and fast execution speeds. Finally, the method in which the microcontroller executes the stored supervisory controller is documented in detail.

4.1 GRAPHIC USER INTERFACE (GUI)

The Graphic User Interface (GUI) functions as the Ground Station. It allows observability and control over the Dual Axis Solar Tracking System. Table 3-2 lists a set of wireless communication packets that are used to communicate information between the Master Controller and the System. Information generated by the system is broadcasted to the MC and displayed on the GUI. The MC sends sweep commands to the System by selecting one of the 5 possible command buttons, visualized in the lower right corner of Figure 4.1-1. The sweep commands are classified as Uncontrollable commands by the System, as they can occur at any time and without warning. A visualization of the GUI is seen below:

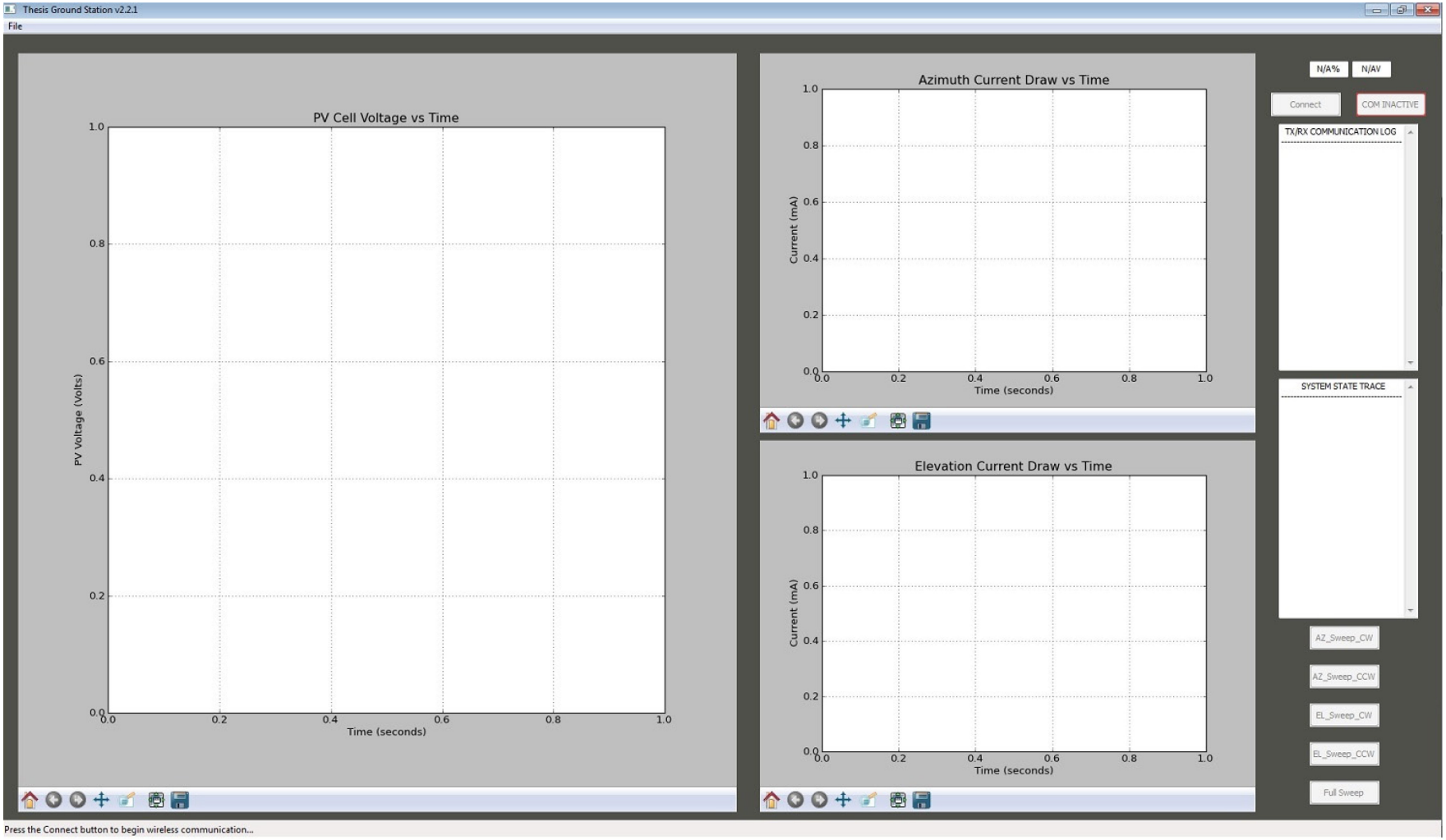


FIGURE 4.1-1 GRAPHIC USER INTERFACE – INITIALIZATION

The Figure 4.1-1 illustrates the graphic user interface at initialization (before communication is established with the system). Three large graphs allow users to monitor the voltage measured at the solar cell, and the current draw of the Azimuth and Elevation motors. The graphs update every 2 seconds and dynamically scales its axes based on the data received.

The right hand side contains a Percentage battery charge and current voltage at the battery. Before communication is established with the system these values are labelled "NA%" and "NAV". Below the battery indicators is the "Connect" button that will establish communication with the system. The button will remain disabled until the user selects a USB port and BAUD in the "Configure RX/TX" selection of the File menu. Connecting to the system is not a command (does not need to be modelled); it simply opens a serial port through the computer's operating system and waits for data. When communication is established with the system, the red outlined "COM INACTIVE" button switches to a green outlined "COM ACTIVE" button. If two way communications with the system is broken for longer than 1 second, the red outlined "COM INACTIVE" is displayed. This allows users to determine whether a healthy two way communication path exists.

Commands that are issues by the GUI and responses that are received from the system are logged in the "TX/RX Communication Log". When the system changes state the information is broadcast to the GUI and is logged on the "System State Trace" window. This way, traceability over the system's behavior can easily be cross referenced against the calculated supervisor.

The GUI has five buttons that become enabled when the "Connect" button has been pressed. Each of these five buttons corresponds to a sweep command. When pressed, the command is broadcast wirelessly to the system. This corresponds to the MC component model seen in 3.3.1.4.

4.2 DRIVER CODE (EMBEDDED SYSTEM)

Microcontrollers require driver software to properly configure the internal registers that control peripherals such as the ADC, I2C, UART, PWM, Timers and Direct Memory

Access (DMA) modules. Additionally, two custom packages were produced to store the supervisor and define the triggering logic for each of the system events. In total, 8 custom packages were produced for this implementation. These are outlined in the table below:

TABLE 4-1 C DRIVER CODE

Package Name	Device Interface	Purpose
thesis_adc.h, thesis_adc.c	PV Voltage Sensor, Motor Current Sensors	Allows measurement of the Analog to Digital devices
thesis_dma.h, thesis_dma.c	N/A	Allows storage of ADC values directly into memory without CPU intervention.
thesis_events.h, thesis_events.c	N/A	Defines the triggering logic of each event.
thesis_i2c.h, thesis_i2c.c	Fuel Gauge	Allows measurement of the percent state of charge and voltage of the onboard battery.
thesis_pwm.h, thesis_pwm.c	Azimuth and Elevation Servo Motors	Allows control of the Azimuth and Elevation Servo Motors.
thesis_supervisor.h, thesis_supervisor.c	N/A	Stores the supervisor calculated in DECK.
thesis_timer.h, thesis_timer.c	N/A	Creates periodic interrupts at 250ms to poll the sensors at 4hz.
thesis_uart.h, thesis_uart.c	Wireless TX/RX Device	Allows two way communications with the master controller (GUI).

The driver code is implemented in the “C” programming language and is stored directly into the Microprocessor’s FLASH memory. High level functions are created to enable wireless data to be transmitted, motor motions to be executed, I2C values to be measured, etc.

The microprocessor collects data from the peripherals in two ways. The first is by using a timer that triggers an overflow event 4 times per second. The Analog sensors (PV Cell & Motor Current values) are polled when an overflow occurs. The Battery State of

Charge does not change often and is polled every 10 seconds. The values are polled and their values are stored into variables using Direct Memory Access (DMA).

The second occurs when data is received by the UART. When data is received, an interrupt occurs and the system begins decoding the packet. Once decoded, the corresponding data variable is set. For example, if a "Full_Sweep" command packet is received, the UART decodes the value and sets the "Full_Sweep" Boolean variable to True.

4.3 DISCRETE EVENT CONTROL TOOLKIT (DECK)

Discrete Event Control Kit (DECK) is a toolbox (set of M–file functions) written in the MATLAB programming language. It enables the analysis and design of supervisory control systems based on the Ramadge–Wonham (RW) theory of supervisory control of discrete–event systems (DES) [15]. The current version of DECK supports the case of supervision under full event observation. [15]

The models illustrated in Chapter 3.3 are coded into MATLAB using the functions defined in DECK. A model is created for each Component, Interaction and Specification. An example of how to code the PV Cell in DECK is illustrated below:

```
Dark_to_Dim = 301;

Dim_to_Bright = 302;

Dim_to_Dark = 303;

Bright_to_Dim = 304;


PV_Marked_States = [3];

PV_STT = [1 Dark_to_Dim 2; 2 Dim_to_Bright 3; 3 Bright_to_Dim 2; 2 Dim_to_Dark 1];

PV_Cell = automaton(3, PV_STT, PV_Marked_States);
```

In the code Chapter above, we assign unique 3 digit values for each event. In Figure 3.3-1 the component diagram for the PV cell is defined. The PV_STT variable is a coded representation of this diagram in the form of a "From" "Event" "To" architecture. For example, State 1 transitions to State 2 when the event "Dark_to"Dim" occurs. Using this method, all component, interaction and specification models are coded into DECK.

The Component and Interaction models are combined using a synchronous product to form the Plant model. This implementation results in a plant of 1,728 states and 20,192 transitions.

As discussed in 3.3.3, three separate specifications for movement are created for this implementation. These include: a specification for Individual Sweep Commands, a specification for a Full Sweep Command, and a specification for a Full Sweep Command with Motor Failure. One of the aforementioned movement specification is combined with the 4 motor control specifications defined in 3.3.3.1 and 3.3.3.2 by means of the product function in DECK. The resulting automaton forms a single specification that combines all five modular specifications into a single specification.

From this point forward the product of the movement commands and motor control specifications are referred as “Individual Sweep”, “Full Sweep” and “Full Sweep with Motor Failure”.

The specification is combined with the plant using the DECK function “supcon”. The output of this function is a maximally permissive, non-blocking supervisor represented as a state transition table (STT). The STT outlines all possible legal event transitions (based on the specification) that the plant may traverse. The following results are tabulated for the three specifications:

TABLE 4-2 SUPERVISOR RESULTS

Supervisor based on Spec	Number of States	State Transitions
Individual Sweep	3213	27,064
Full Sweep	1143	10326
Full Sweep with Motor Failure	1620	7,142

Each of the three supervisors listed above are separate implementations, based on the three unique Movement Specifications outlined in 3.3.3.3. For example, the Full Sweep with Motor Failure implementation resulted in a calculated supervisor that is expressed as a 7,142 by 3 array. A sample of the STT is illustrated below:

1 301 2

1 504 3

1 604 4

2 302 5

2 303 1

2 504 6

2 601 7

2 604 8

The column in the left hand side denotes to “from” state. The 3 digit value in the middle is a unique identifier for the event. The final digit on the right hand side denotes the destination state. For example, the system will transition from state 1 to state 2 when the “Dark_to_Dim” event occurs (*note that the Dark_to_Dim event corresponds to the unique identifier 301*). When we enter state 2, we transition from 2 to 5 when Dim_to_Bright (*302*) occurs.

It should be noted that event 302 (Dim_to_Bright) is not present in state 1. Even though this is an uncontrollable event, the supervisor will not include it in the STT as it is not a possible within the plant. Therefore, not only does the calculated supervisor limit the controllable events to enforce legal behaviour, it also removes the events that are not possible within the plant. The STT is therefore a complete list of events that could potentially occur on a state by state basis.

For a complete list of the DECK code created for this application, refer to Appendix C.

4.4 STORAGE OF THE SUPERVISOR

The system must quickly read the STT in order to determine which events are possible, which should be disabled and which state to transition to when a legal event occurs. Looping through a list of 7,142 entries is extremely time inefficient. For this reason the STT is pre-processed and stored as an array of structs, where each struct stores the data associated to the state at that index value. By doing this, the state number does not need to be stored, as it can be implicitly defined as the array index value. The data structure includes: the total number of enabled events for that state and an N x 2 array of events and their destination states. The data structure is visualized below:

```
struct state_elements{
    uint16_t len;
    const uint16_t (*stt)[2];
```

};

For example, consider the sample portion of the Full Sweep with Motor Failure STT defined in 4.3. The STT would consist of two state element structures stored in an array of size 3 (for simplicity, we leave array entry 0 blank to allow direct equivalency between state and index values). For this reason, the array length is always 1 larger than the number of states present in the STT. The data stored in Array value 1 is equivalent to the data associated to state "1". This includes a length variable equal to 3 and a 3 by 2 table that lists the possible events and their destination states. For this example, they would be: [[301,2],[504,3],[604,4]]. The array value 2 contains, a length variable of size 5 and a 5 by 2 table that lists [[302,5],[303,1],[504,6],[601,7], [604,8]].

As events are triggered in the system, the current state is updated accordingly. As the system always keeps track of the current state, determining the data that corresponds to that state is directly indexable and does not require searching.

Note: post processing is required to transform the N by 3 state transition table calculated in DECK to the Array of Structures format. A python based script was developed to allow the output generated from DECK to be quickly be converted to the proper "C" code.

4.5 EVALUATION CYCLE

The evaluation cycle locates and unlocks the legal events permitted by the supervisor, determines whether to trigger an event, and determines which state to transition to when an event occurs. This is visualized in the figure below:

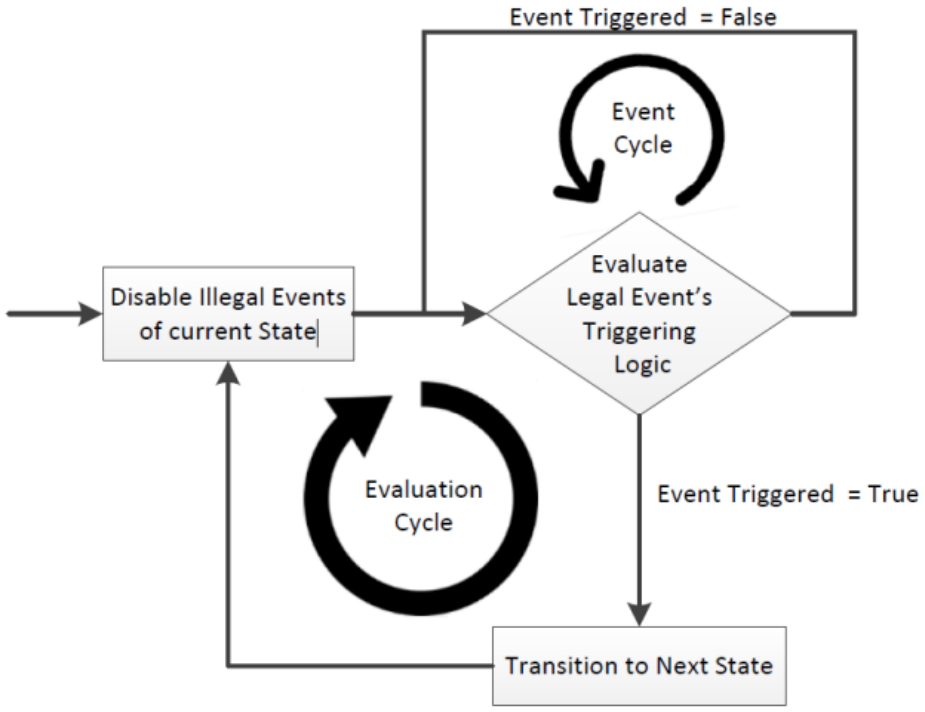


FIGURE 4.5-1 EVALUATION CYCLE

The evaluation cycle is continuously run in the microcontroller's main loop. The system initializes in state 1 and the events allowed by the supervisor at this state are enabled. When an event occurs, the system transitions to the state associated with that event and the process repeats. This process is encapsulated in three separate phases and is discussed in detail in the sub-chapters that follow.

4.5.1 EVENT DISABLEMENT PHASE

The first phase in the Event Cycle is the Event Disablement Phase. The current state is used to determine which events are permissible. To do this, the following pieces of code are used:

```
void unlock_events(uint16_t state){
        for(int i = 0;  i< SUPERVISOR[state].len; i++)
                enable_EVENT(SUPERVISOR[state].stt[i][0]);
          }
```

The "unlock_events" function uses the current system state as an input. It then uses the STT, named SUPERVISOR[] and looks up the data stored at the index value of the current state. Using the length variable associated to the current state, the code loops through each of the entries listed in the 2 dimensional state array and enables them using the "enable_EVENT" function.

Every event in the plant has a Boolean "_ENABLED" value associated to it. If the value is true, the event is enabled. If false, the event is disabled (or not part of the plant's event set for the current state). Additionally, each event uses a unique three digit variable to identify it. For example: the "Dark_to_Dim_var" represents the unique identifier for the Dark_to_Dim event (401). This identifier is assigned when the system is translated into DECK. The information is stored in the state transition table whenever that variable should be enabled.

The "enable_EVENT" function accepts the unique event identifier stored in the 2 dimensional state array as an input and sets the corresponding _ENABLED event to True. The code for the "enable_EVENT" function is outlined below:

```
void enable_EVENT(uint16_t event_Number){
        if(event_Number == Dark_to_Dim_var)     Dark_to_Dim_Enabled = true;

        …

        else if(event_Number == EL_MOTOR_FAIL_var) EL_MOTOR_FAIL_ENABLED = true;}
```

When a state transition occurs, the Boolean _ENABLED variables associated to each event are set to false. Immediately after this occurs, a new set of event enablements is issued by the microcontroller. This architecture ensures that no illegal events may be triggered.

In conclusion, the objective of the Event Enable Phase is to enable the set of permissible events associated to the current state.

4.5.2 EVENT CYCLE PHASE

The system sensors are polled at 250 ms intervals and the results are stored as variables in local memory. At this point, the variables represent signals, not events. Events are *only* triggered during the Event Cycle Phase. An event will be triggered if the event is enabled and the triggering logic has been satisfied. In some cases the triggering logic is associated with a discrete value; in other cases the events are associated to analog values. For analogs, the system checks whether or not the current value exceeds some boundary – ultimately, the evaluation must return a True or False value.

4.5.2.1 EVENT PRIORITY

The system sequentially evaluates every possible event defined in the models (and subsequently coded into DECK). The first event that is evaluated as "True" breaks the

loop and the system proceeds to the next phase. This creates a priority list between events. General guidelines on how to determine these priorities are provided in the paragraphs that follow. A complete list of events and their priorities is tabled below:

TABLE 4-3 EVENT PRIORITY LIST

Priority	Event Name	Component Model	Controllable?
1	AZ_CCW_OK	Azimuth Motor Motion	No
2	AZ_CW_OK	Azimuth Motor Motion	No
3	EL_CCW_OK	Elevation Motor Motion	No
4	EL_CW_OK	Elevation Motor Motion	No
5	EL_FAIL_MOVE	Elevation Motor Motion	No
6	Safe_to_Full	Battery SOC	No
7	Full_to_Safe	Battery SOC	No
8	Crit_to_Safe	Battery SOC	No
9	Safe_to_Crit	Battery SOC	No
10	Dark_to_Dim	PV Cell	No
11	Dim_to_Bright	PV Cell	No
12	Dim_to_Dark	PV Cell	No
13	Bright_to_Dim	PV Cell	No
14	AZ_MAX_CW	Azimuth Motor Range	No
15	AZ_MAX_CCW	Azimuth Motor Range	No
16	AZ_RANGE_OK	Azimuth Motor Range	No
17	EL_MAX_CW	Elevation Motor Range	No
18	EL_MAX_CCW	Elevation Motor Range	No
19	EL_RANGE_OK	Elevation Motor Range	No
20	AZ_Sweep_CW	Master Controller	No
21	AZ_Sweep_CCW	Master Controller	No
22	EL_Sweep_CW	Master Controller	No
23	EL_Sweep_CCW	Master Controller	No
24	Full_Sweep	Master Controller	No
25	EL_MOTOR_FAIL	Master Controller	Yes
26	Bright_Detected	Master Controller	Yes
27	Sweep_Failure	Master Controller	Yes

28	AZ_POLL_RANGE	Azimuth Motor Range	Yes
29	EL_POLL_RANGE	Elevation Motor Range	Yes
30	AZ_CCW_MOVE	Azimuth Motor Motion	Yes
31	AZ_CW_MOVE	Azimuth Motor Motion	Yes
32	EL_CCW_MOVE	Elevation Motor Motion	Yes
33	EL_CW_MOVE	Elevation Motor Motion	Yes

There are three rules of thumb associated to creation of the event priority list. These are discussed below.

1. Set all controllable events to the lowest priorities, while setting uncontrollable events at higher priorities.

This way, the system has visibility over the status of the various components before making a decision (ie: triggering a controllable event). If Simultaneous Events are capable of being triggered in the same Evaluation Cycle, only the highest priority one will be triggered. Possible issues arise if the triggered event transitions to a state where the second un-triggered event is no longer possible in the plant model. This can occur if the first event triggers an interaction that prevents the second event from being present in the plant model.

> For example: At time = 250ms, the sensors poll values such that the events of Safe_to_Full and Dim_to_Dark are both eligible for triggering. If Safe_to_Full triggers first, the Dim_to_Dark event will still be present in the next state as well. However, if Dim_to_Dark triggers first, the event of Safe_to_Full will not be present the next state, as there is an interaction model (Figure 3.3-8) that removes it from the plant.

If the Dim_to_Dark event triggers before the Safe_to_Full event, the Safe_to_Full event will never trigger and the Supervisor will be out of sync with the system. This is obviously undesirable. For this reason, the priority in which the uncontrollable events are evaluated is important. Therefore, the second rule of thumb is to

2. Evaluate the system interaction models to ensure that events suppressed in certain states are evaluated before the event that triggers the entry into that state.

By observing this rule, the following ordering rationale was determined for this implementation:

From Figure 3.3-8, it's observed that the events of Crit_to_Safe and Safe_to_Full must be evaluated before the Dim_to_Dark event.

From Figure 3.3-10, it's observed that the events of AZ_CW_OK, AZ_CCW_OK, EL_CW_OK and EL_CCW_OK must be evaluated before the Safe_to_Crit event.

From Figure 3.3-11, it's observed that the Crit_to_Safe and Safe_to_Full events must be evaluated before the motor MOVE commands.

Chapter 3.3.3.3.3 states that when modelling the specification behaviour of multiple uncontrollable events it's necessary to consider all possible combinations from the point the events may occur. Assuming this is correctly implemented, the DECK software creates a supervisor such that all possible permutations of uncontrollable event triggering are possible. This is the maximally permissive property of supervisory control.

3. The first controllable event to be enabled will be triggered. Ordering of the controllable event priority is therefore also a priority. It's recommended that failure indications be controlled first (EL_MOTOR_FAIL), followed by command responses (Bright_Detected and Sweep_Failure), followed by component actuations (AZ_POLL_RANGE, AZ_CCW_MOVE, etc…).

There are 9 controllable events present in the implementation. The event cycle triggers the first enabled controllable event in the priority list. If two controllable events are unlocked in the same state, the event cycle will always trigger the first controllable event and the second will never be triggered. This is a by-product of the sequential behaviour of the Event Cycle and cannot be avoided. The selection over which controllable events are to be triggered over others is achieved by ordering them in the priority list. For this implementation, no situations exist where more than one controllable event may occur at any given time. The priority of the controllable events is therefore unnecessary for this specific implementation. However, if multiple controllable events do occur within the selected implementation, a static priority list is not advised as only the first controllable event will ever be triggered. In these situations, additional logic must be applied to interleave the firing of these controllable events. This topic is beyond the scope of this thesis and methods to alleviate this issue are discussed in [13].

4.5.2.2 EVENT TRIGGERING

The event triggering logic is essentially a list of "IF" statements coded in "C" that contain three main components. First, a Boolean parameter identifies if the event is enabled. Second, a boolean value identifies if the events triggering logic is satisfied. This can either be a Boolean variable, or in the case of Analog values, the result of a Boolean function that verifies whether the associated parameter has crossed a predefined boundary. Third, an optional delay parameter becomes True after a particular time interval has elapsed. A visualization of this is seen below:

```
if(Event_Enabled && Event_Trigger_Logic && Event_Time_Delay){
        eventTriggered = true;
        //Reset Variables, Find Next State
        return;
        }
```

As this is an "If" statement, each of the three components are logically "anded" together and must evaluate to either True or False. Below is an example of the event triggering logic of the Azimuth Sweep Clockwise command. Recall that this is an uncontrollable event that is broadcast from the GUI (MC) and decoded by the System's UART.

```
if(AZ_Sweep_CW_ENABLED && AZ_Sweep_CW){
        eventTriggered = true;
        //Reset Variables, Find Next State
        return;
        }
```

In the code sample above, the "AZ_Sweep_CW_ENABLED" parameter is a Boolean variable that identifies whether the event is enabled or disabled. These values are set during the Event Disablement Phase discussed in chapter 4.5.1.

The AZ_Sweep_CW parameter is a Boolean variable that is set to True when the UART decodes an "AZ_SWEEP_CW" command sent by the Master Controller. After the event triggers, this command must be reset to False (hence the "Reset Variable" comment in the code body).

There is no time delay associated to the Azimuth Sweep Command. For this reason, the optional time delay Boolean is simply omitted from the expression.

Next, we consider an event that triggers when a pre-set logical boundary has been crossed. In Table 3-3 the events associated to the PV Cell and their transition boundaries are defined. For example: the Dark_to_Dim event will occur when the model is in the

"Dark" state and the voltage generated by the PV cell is greater than 6. This leads to the event definition below:

```
if(Dark_to_Dim_Enabled && Dark_to_Dim()){
  eventTriggered = true;
  //Reset Variables, Find Next State
  return;
  }
```

The supervisor tracks the solar panel state and will only include the "Dark_to_Dim" event into the STT when the solar panel is in the "Dark" state. Therefore, if the Dark_to_Dim_Enabled variable is True after the Event Disablement Phase, the Solar Panels are guaranteed to be in the Dark state.

The Dark_to_Dim function determines if the voltage of the PV cell is greater than a pre-set boundary. The logic for this function is illustrated below:

```
inline bool Dark_to_Dim(){
   if( PV_Cell_Voltage > 6 ) return true;
   else return false;
   }
```

If the voltage of the PV Cell is greater than 6 the function returns True, otherwise the function returns False. The function is defined as an "inline" function to increase the evaluation speed at run time. By logically "anding" the Dark_to_Dim_Enabled variable with the Dark_to_Dim() function, the Dark_to_Dim event will trigger when the PV cells are in the "Dark" state and the voltage generated by the PV cell is greater than 6. There is no time delay associated to the Dark_to_Dim event. For this reason, the optional time delay Boolean is simply omitted from the expression.

As a final example, an event that requires a time delay parameter is considered. The "AZ_CW_MOVE" command is a controllable event that triggers the Azimuth servo motor to move clockwise by 2 degrees. The motor takes approximately one second to reposition itself from its initial position to 2 degrees further in the clockwise direction. The "AZ_CW_OK" event evaluates whether the "AZ_CW_MOVE" command was executed successfully. It accomplishes this by evaluating the logic defined below:

```
if(AZ_CW_OK_ENABLED && AZ_CW_OK() && move_delta_T()){
        eventTriggered = true;
        //Reset Variables, Find Next State
        return;
        }
```

The AZ_CW_OK function behaves in a similar fashion to the "Dark_to_Dim()" function defined above. The boundaries for this event are defined in Table 3-5 and are not repeated for brevity.

When an AZ_CW_MOVE event is triggered, the current system's second timer is stored as a variable named "t_move_requested". The move_delta_T() function evaluates whether the current system's second timer is at least 2 seconds greater than the "t_move_requested" variable. This is visualized in the logic below:

```
inline bool move_delta_T(void){
        if(second_counter >= (t_move_requested + 2)) return true;
        else return false;
}
```

If two seconds have not yet passed the expression evaluates to false and the microcontroller continues sequentially down the priority list. If at least two seconds have elapsed, the move_delta_T() function evaluates to True and the event is permitted to be triggered.

It's important to keep the time granularity in mind when determining the minimum time boundary. In this implemention, it's possible that the AZ_CW_MOVE command occurs at time 1.997 seconds, resulting in a t_move_requested value of 1. When the second counter reaches 3, the move_delta_T() function will return a True value, but the event will have only been delayed by 1.003 seconds. However, even with this worst case analysis the 1 second time period required to reposition the motor by 2 degrees is still satisfied. This architecture is therefore valid for this implementation. For implementations requiring sensitive time delays it's suggested to use a timer that sets a Boolean value to True during the overflow routine.

Finally, the triggering logic for controllable events is simply the event's associated Boolean _Enabled variable. An example of the AZ_CW_MOVE command is shown below:

```
if(AZ_CW_MOVE_ENABLED){
        AZ_CW_MOVE();
        //Reset Variables, Find Next State
return;
}
```

As previously discussed, only the highest priority controllable event present in the current state's potential transitions will be triggered as the microcontroller immediately exits the evaluation cycle when an event is triggered.

4.5.3 STATE TRANSITION PHASE

The State Transition Phase begins immediately after an event has been triggered. The objective of this phase is to determine the next state indicated by the newly triggered event. As previously mentioned in 4.2, the STT is indexed as an array of structs and the N x 2 array stt* pointer identifies the enabled events and their destination states when triggered. A function named "find_next_state" is used to search the N x 2 array to determine the correct destination state when an event becomes triggered. The logic for this function is outlined below:

```
uint16_t find_next_state(uint16_t current_state, uint16_t event){

        uint16_t length = SUPERVISOR[current_state].len;
        …
        for(i = 0;  I < length; i++){
                if(SUPERVISOR[current_state].stt[i][0] == event){
                        return SUPERVISOR[current_state].stt[i][1];
                        }
        }
```

The function uses the current state and the newly triggered event as inputs. The N x 2 array is searched and the destination state is returned by the function.

When the State Transition Phase is complete, the evaluation cycle repeats in the new system state.

CHAPTER 5. RESULTS

This Chapter outlines the results achieved from the physical implementation of the Dual Axis Solar Tracking System and the supervisory controller that ensures it is maximally permissive and non-blocking. The chapter begins by illustrating the results collected by the GUI during a typical operation of the system. Next, the effects of Inexact Synchronization, Avalanche Effect and Simultaneous Events present in the implemented system are explored. A formula used to calculate the exact code size required for the storage of the calculated supervisor is then presented. Finally, the disadvantages of the Microcontroller based implementation are discussed.

A picture of the final assembled system and graphic user interface can be found in Appendix D.

5.1 SAMPLE RESULTS

This chapter explores sample results obtained during the testing of the Dual Axis Solar Tracking System. An explanation of the default configuration is provided and sample sequences of events are provided. Chapter 3.3.3.3 outlined three separate supervisors that were generated for three separate movement specifications. This section only considers the "Full Sweep with Motor Failure" supervisor as the sole implementation. In total this supervisor contains 1,620 states and 7,142 transitions.

The status of the Dual Axis Solar Tracking system is monitored by the Graphic User Interface. The system configuration on start-up is defined by the entry states defined in the component models outlined in 3.3.1. For this implementation: the motors are idle, the Range is "In Range", the PV Cell is "Dark", the battery is "Safe" and the MC is Idle. The number associated to this state is "1". The system periodically polls the sensors at 4 Hz frequencies and corresponding events are triggered if their transition logic becomes satisfied. The system remains in state 1 unless a Full Sweep, a Dark_to_Dim, or Safe_to_Crit event occurs.

When a "Full_Sweep" command is issued by the MC, the system responds by transmitting the new state of the system, in this case, "3". The system remains in state 3 until a Dark_to_Dim, Safe_to_Crit, Full_Sweep (which will lead back to state 3), or a

AZ_POLL_RANGE event is triggered. Since the "AZ_POLL_RANGE" event is a permissible controllable event, the Event Cycle will trigger this event unless the aforementioned uncontrollable commands are also triggered. The system continues to execute the supervisor outlined in Appendix A by sequentially evaluating the enabled events for each state. Below is a sample of the State Trace portion of the GUI after a "Full_Sweep" command has been issued by the MC:

1 (Full_Sweep occurred)

3 (AZ_POLL_RANGE occurred)

9 (AZ_RANGE_OK occurred)

20 (AZ_CCW_MOVE occurred)

35 (AZ_CCW_OK occurred)

3 (AZ_POLL_RANGE occurred)

…

The supervisor will continue to enforce this behaviour until the movement specification has terminated or a Dim_to_Bright event has occurred. The amount of illumination on the PV cells is visualized on the GUI's "PV Cell Voltage vs Time" graph. A sample output is visualized below:

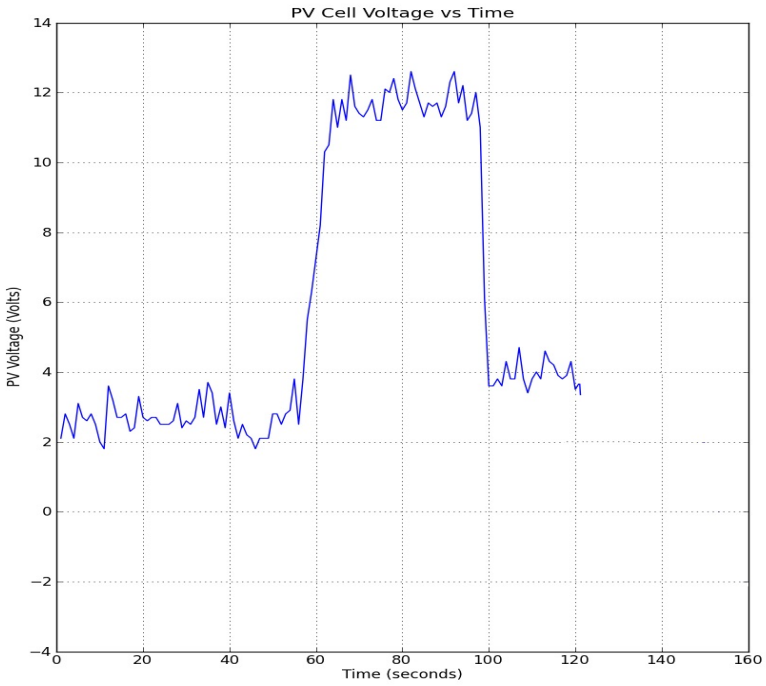


FIGURE 5.1-1 SAMPLE PV CELL VOLTAGE VS. TIME GRAPH. X AXIS = VOLTAGE MEASURED BY THE VOLTAGE SENSOR (V) Y AXIS = TIME IN SECONDS

A corresponding graph is also provided for the Azimuth and Elevation motor currents. The system monitors whether the current draw on the elevation motor is significantly high after a move command has been issued. This is visualized in the Elevation Current Draw vs Time graph illustrated below:

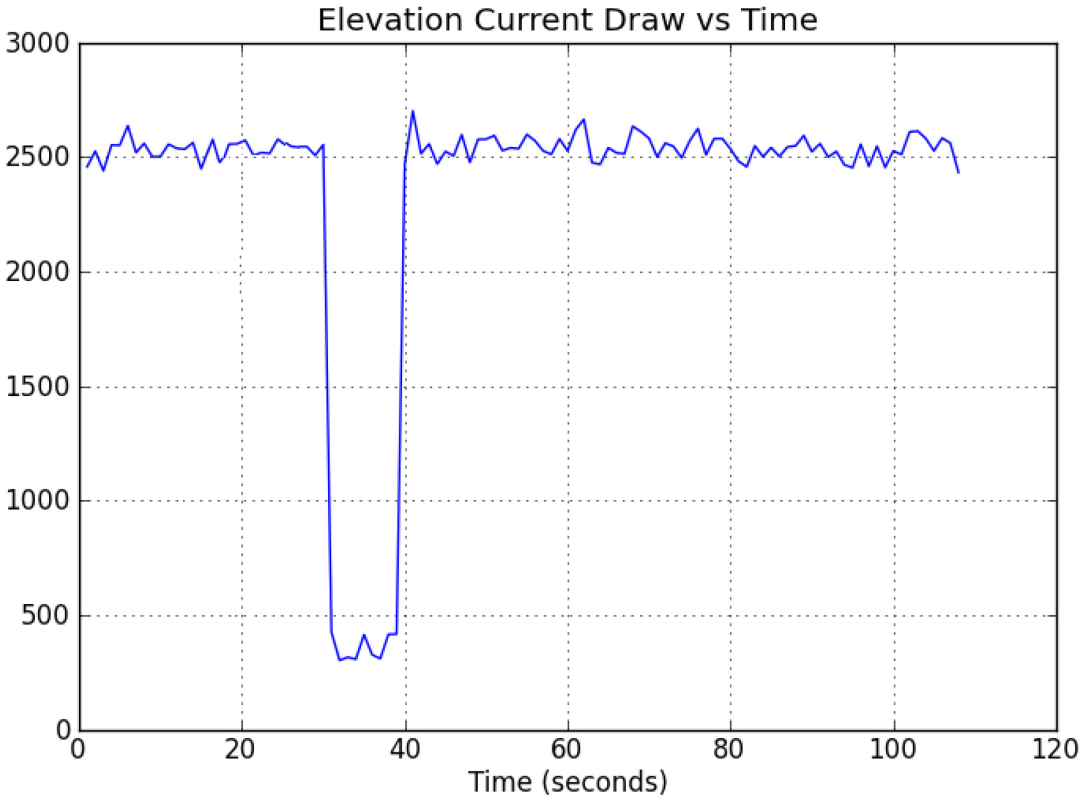


FIGURE 5.1-2 SAMPLE ELEVATION CURRENT DRAW VS. TIME GRAPH. X AXIS = VOLTAGE OUTPUT FROM THE CURRENT SENSOR A1 (IN MILIVOLTS) Y AXIS = TIME IN SECONDS.

In the figure above, the supervisor enabled the "EL_CW_MOVE" command to trigger at time 30 seconds. In this setup, the pan-tilt assembly was physically obstructed, leading to the sharp increase in current draw visualized above. It should be noted that the Y axis represents the voltage output (in millivolts) from the current sensor. The amount of current measured is proportional to the voltage output by the sensor. Steady state (motor idle with no obstruction) is 2.5V, or 2500 mV as indicated above. When an obstruction is encountered, the current increases sharply and the output voltage of the sensor drops sharply.

5.2 INEXACT SYNCHRONIZATION

The speed at which a microcontroller can detect the occurrence of an event is extremely important. In all digital implementations there exists a fundamental time difference between the occurrence of an event and the time at which it is detected by system.

Digital systems can only observe changes in their environment at periodic time slices relative to their clock speeds. This is a phenomenon known as Inexact Synchronization.

The issue of inexact synchronization cannot be illuminated entirely. As such, this issue manifests itself in the implementation outlined in this thesis. However, the architecture presented in this thesis can be used to minimize the effects of inexact synchronization.

An example of inexact synchronization for this implementation would be if a cloud passes overhead. The illumination present on the solar panel will diminish instantaneously but in order for a system to perceive a change in its environment, the voltage sensor must be polled and the microcontroller must trigger the associated event. The time required for the sensor to detect a change in its environment is referred to as the Acquisition Time. This time gap includes the polling of the sensor, the conversion from analog to digital, and the storing of that information into the system's memory. The time required for the Microcontroller to trigger an event is referred to as the Triggering Delay. A visualization of this can be seen below:

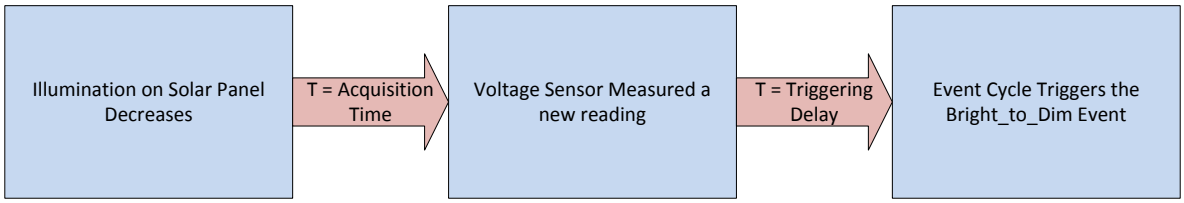


FIGURE 5.2-1 ACQUISITION TIME AND TRIGGERING DELAY

The inexact synchronization between the plant and the supervisor can therefore be expressed as the sum of the acquisition time and the triggering delay. For supervisory controllers this time delay is theoretically zero, however, this is impossible in practice. In all cases, the triggering delay should be as close to zero as possible, but the acquisition time may vary from implementation to implementation. It is therefore necessary to take into account the requirement of the system's behaviour when determining what an acceptable acquisition time delay is. This obviously varies from device to device, for example: a missile guidance system will require drastically lower time delays when compared to toaster oven. For this implementation the acquisition frequency is set to 4 Hz, yielding a worst case acquisition time of 250ms.

PLCs execute a scan cycle that periodically polls inputs, evaluates logic, and drives outputs. Typical scan cycles are 10ms [2] and represents both the acquisition time and the triggering delay.

In the microprocessor based implementation outlined in this thesis, the sensor values are updated four times per second (4hz) and the evaluation of the event logic is performed in the Evaluation Cycle. The acquisition time is not an important consideration for this thesis, since (as mentioned above) the speed required will vary greatly from implementation to implementation. The focus of this implementation is to outline a method that drives the Triggering Delay as close to zero as physically possible.

The Triggering Delay represents the maximum amount of time between the changes in an event's source data to the triggering of the associated event. For example, if the solar panel is in the "Dark" state and a new Voltage measurement indicates a value of 7 Volts, how long will it take for the Evaluation Cycle to trigger the event and transition to the next state?

To measure the triggering delay, the microprocessor sets a discrete output high at the beginning of each State Transition Phase. When each of the phases end, the associated discrete output is set back low. The following figure represents the time required to complete the State Transition phase.

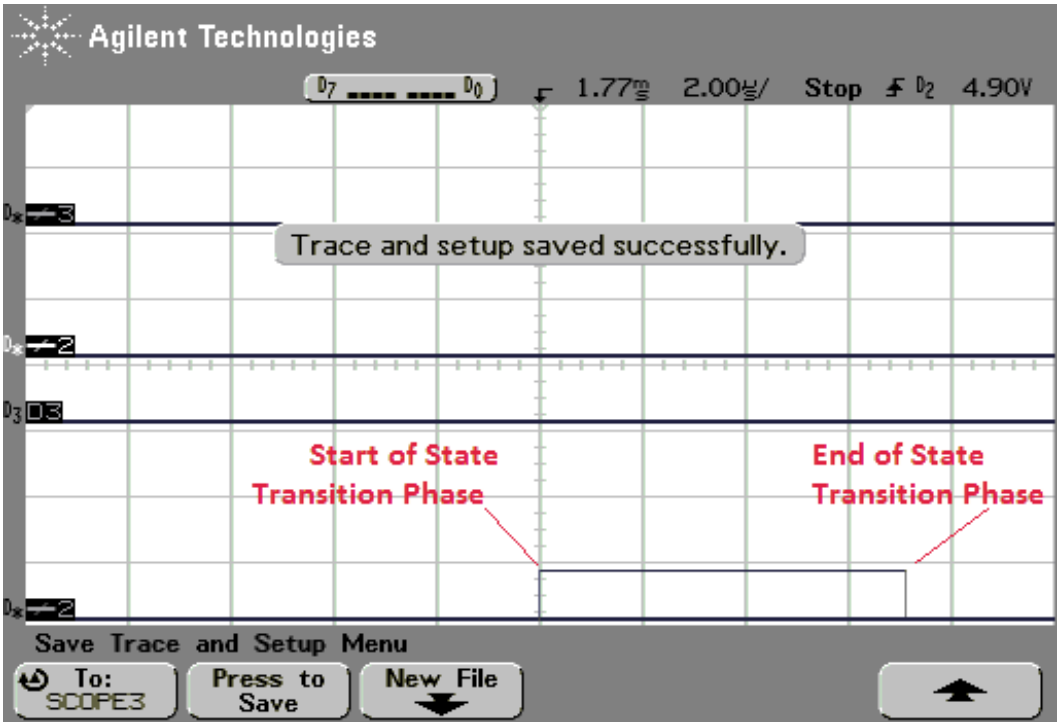


FIGURE 5.2-2 STATE TRANSITION PHASE DURATION

From the above measurement, the State Transition Phase takes roughly 7 microseconds to execute *(Recall that this phase uses the last triggered event to determine the next state to transition to)*.

This phase only begins when an event has been triggered. A logic analyzer was used to measure the time difference between two successive state transition phase pulses, which is equivalent to the amount of time required for an iteration of the event cycle. The figure below illustrates the results measured by the logic analyzer after a successful "AZ_Poll_Range" and "AZ_RANGE_OK" events were triggered.

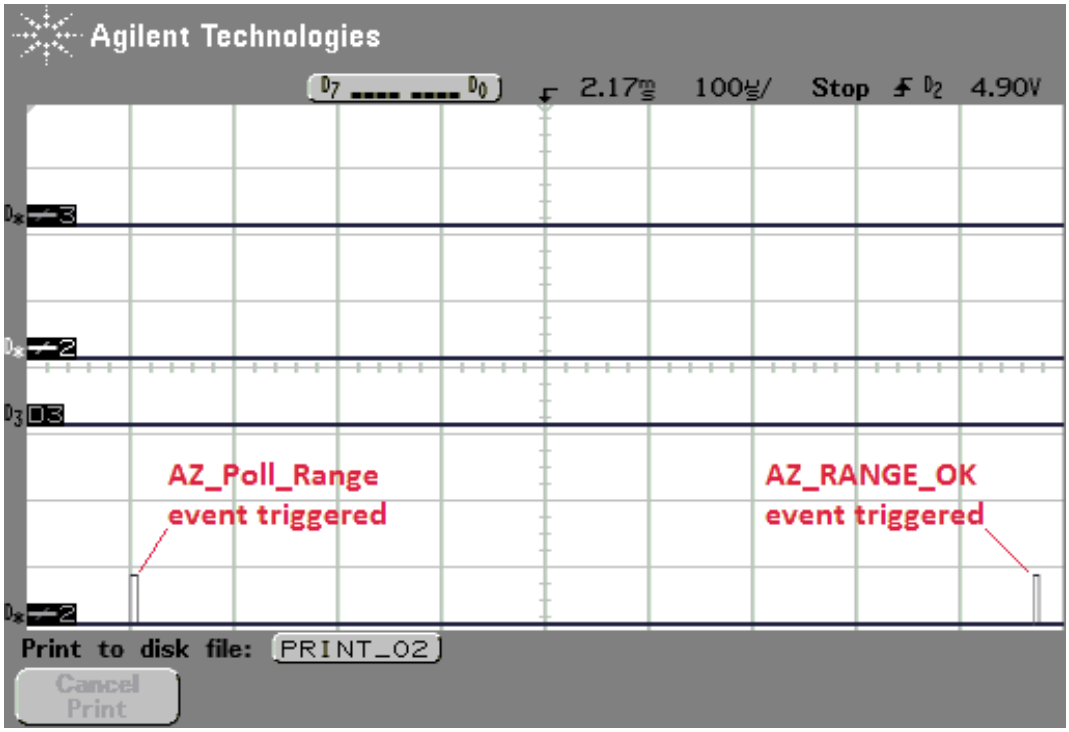


FIGURE 5.2-3 EVENT CYCLE DURATION

From the figure above, the time required for the Event Cycle to execute is approximately 880 microseconds (or 0.88ms). It is important to note that there is 0 acquisition time required between the Azimuth Poll Range controllable event and the Azimuth Range OK uncontrollable response as the data used in the triggering of this event is stored in memory and not measured by a sensor. It should also be noted that the priority associated to the AZ_RANGE_OK event is number 16 of 28. Events that are lower in priority will experience slightly longer delays. However, the positioning of 16 of 28 is roughly halfway through the priority list and represents a reasonable average time. It is therefore safe to conclude that the total triggering delay for the implementation outlined in this thesis is approximately 880 microseconds.

Comparing the triggering delay value of 0.88ms to that of PLC scan cycle is not directly compatible. As mentioned above, the 10ms scan cycle time for a PLC takes into account the acquisition time and the triggering delay, where this implementation measures only the triggering delay. However, time difference between 10ms and 0.88ms is significant. Unless the acquisition time for a PLC scan cycle consumed more than 92% of the total scan cycle time, the implementation outlined in this thesis represents a decrease in the overall triggering delay.

The effect of inexact synchronization is therefore mitigated when using the implementation outlined in this thesis.

In order for the above statement to be true, a few caveats need to be outlined. First, the speed of the microprocessor is important. The microprocessor selected for this implementation operates on 48Mhz. If the implementation is executed on a faster microcontroller, the triggering delay may be further reduced. The opposite is also true, a slower microprocessor clock speed will result in a longer triggering delay.

The second caveat relates to the amount of information stored in the state transition table. As discussed in the Chapter 4.5.2, the Evaluation Cycle sequentially evaluates all legal events associated to the current state. As the list of legal events grows, so to will the worst case triggering delay. The example illustrated above shows the time required for the AZ_RANGE_OK event to be triggered, which is priority number 16 in the list of events. Events lower in the priority list will experience slightly longer triggering delays while events higher in priority result in shorter triggering delays.

5.3 AVALANCHE EFFECT

The avalanche effect is a consequence of the sequential evaluation of the Boolean expressions within the PLC [2]. The Figure below outlines an example of the avalanche effect. The left hand side of the figure shows the finite state machine while the area on the right shows implementation in a ladder diagram (LD). This example is borrowed from the paper outlined in [2].

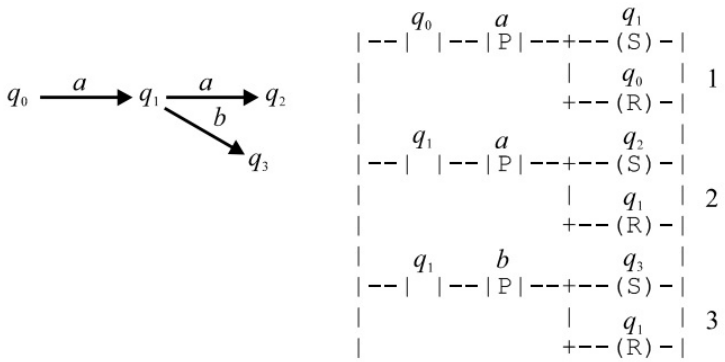


FIGURE 5.3-1 AVALANCHE EFFECT EXAMPLE [2]

The sequential nature of the PLC is such that the ladder diagrams are evaluated from top to bottom. When the event “a” occurs it will be detected during the next scan cycle. At which time the PLC will begin evaluating the LD starting at rung 1 down through to

rung 3. Rung 1 will evaluate Rung 1 to True, as the system is in state q0 and the event "a" has occurred. The PLC will then immediately begin to evaluate Rung 2, which will also evaluate to True as the system is now in state "q1" and the event "a" is still True. This phenomenon is known as the Avalanche Effect and causes the system to skip from State 0 to 2 on a single occurrence of "a".

It should be noted that the effect of the avalanche effect in PLCs can be avoided by arranging the rungs in reverse order. However, this is a step that does not need to be taken in the implementation outlined in this thesis as it does not suffer from the effects of the Avalanche Effect.

The Avalanche Effect is completely avoided in this implementation as the Event Cycle phase immediately terminates when an event is evaluated to have occurred.

5.4 SIMULTANEOUS EVENTS

The implementation outlined in this thesis polls the various sensors at a frequency of 4Hz. It's possible that after the measuring of the sensors that two or more events may eligible for triggering. The implementation's architecture is such that only 1 event will be triggered and that the system will immediately transit to the state associated to the triggered event. The selection of which event should be triggered is known as the issue of multiple (or simultaneous) events. It is critical that only one event be triggered at any given time. "When the controller generates some events only one of the possible events must be generated. Generating several may be contradictory and catastrophic" [2]. Regardless of the implementation, a SCT based implementation must "choose and transit; and only a single event must be chosen." [1].

For the microcontroller based implementation outlined in this thesis, a priority list is created that allows a clear and concise definition over which events should take priority above others. Guidelines for the prioritization process are outlined in Chapter 4.5.2.1.

For PLC based implementations the ordering of the rungs dictates the priority that the events will be evaluated in. This requires careful analysis of the entire ladder diagram, which can be long, tedious and error prone – especially with larger scale implementations. "If the choice is not explicitly made by the implementor, the PLC itself will make the choice, determined by the ordering of the rungs." [1].

The implementation outlined in this thesis shows a clear advantage in the clarity of the priority in which events are evaluated and triggered. While the implementation does not mitigate the effects of Simultaneous Events it does offer a greater degree of visibility over the selection process. Once enforced, designers can rest easy knowing the priority list will be followed in all cases. In PLC based implementations, error in the rung ordering may be present in the implementation that forces the system to behave in an unexpected manner.

It should be noted that the issue of selecting between multiple controllable events is known as the issue of "Choice" and is not covered in this thesis. Section 6.2 offers potential solutions to this problem that are made available through the implementation of the architecture presented in this thesis.

5.5 SCT IMPLEMENTATION VERSUS CODE SIZE

This Chapter outlines a method for implementers to evaluate the approximate memory required to store the STT generated in DECK. Embedded systems have limited onboard memory. In the case of this system, 256kB of data is available for storage.

The supervisory controller generated by the DECK is expressed as a state transition table and is stored within the microcontroller's onboard memory. Chapter 4.2 outlines the manner in which the N by 3 state transition table is reassembled as an array of C structs.

The re-arrangement of the N by 3 state transition table to an "N" sized array of structs represents an increase to the amount of memory required to store the state transition table. However, the speed at which the events are located increases immensely. Onboard memory is becoming increasingly large, for this reason the tradeoff of memory size for event triggering speed is justified.

Recall that each state has the following data structure:

```
struct state_elements{
        uint16_t len;
        const uint16_t (*stt)[2];
};
```

Each data structure consists of a 2 byte sized "length" variable. Additionally, a 4 byte sized "stt" pointer to the 2 dimensional array that stores the enabled events and their transition state values is also required. This 2 dimensional array is of size 4 (2 x 16 bit integers) and increases in size for each legal event associated to the state. A static value of 10 bytes is required as this implementation keeps array location 0 empty, requiring 1 extra state and 1 extra entry in the STT.

With this in mind, the following formula was derived to determine the amount of bytes required to store the state transition table generated from DECK:

$$Required\ Memory\ (bytes) = 4(size\ of\ the\ STT) + 6(\#\ of\ states) + 10$$

The implementation defined in this thesis resulted in three separate specifications that resulted in three separate supervisors. These have been tabled below:

TABLE 5-1 DECK IMPLEMENTATION VS. CODE SIZE

Supervisor based on Spec	Number of States	State Transitions	Code Size (Bytes)
Individual Sweep	3213	27,064	133,970
Full Sweep	1143	10326	50458
Full Sweep with Motor Failure	1620	7,142	41,538

With this formula, designers are capable of determining the required memory size to store a supervisor generated in DECK. This allows implementers to accurately determine the memory required for the physical implementation.

5.6 IMPLEMENTATION DISADVANTAGES

One of the major disadvantages with this implementation is also shared with PLC based implementations. If two events are capable of being triggered at the same time, the highest priority event will be triggered first and the second will need to wait until the next evaluation cycle. This also occurs in PLC based implementations as only one event will ever be selected and transitioned upon; the other signal must wait for the next Scan cycle.

In the microcontroller implementation, the plant will therefore be out of synch with the supervisor for an additional 0.88ms. This phenomenon is unfortunately unavoidable, but can be minimized with shorter triggering delays.

Possible issues may also arise if the triggered event transitions to a state where the second un-triggered event is no longer possible in the plant model. This can occur if the first event triggers an interaction that prevents the second event from being present in the plant model. For this reason, it's important to properly select the event priority. Refer to Chapter 4.5.2.1 for additional details.

CHAPTER 6. CONCLUSIONS

This section outlines a brief summary of the results achieved in this thesis and a brief discussion on potential future work.

6.1 SUMMARY

A physical implementation of a supervisory controller for a Dual Axis Solar Tracker executing on a microcontroller was outlined in this thesis. The supervisor monitors the system (plant) by periodically measuring the sensors and implements the behaviour outlined in the specifications. A detailed list of all processes used to successfully implement the microcontroller based supervisory controller has been documented.

Multiple issues occur during the implementation of a supervisory controller, as explained in other papers such as [2]. The implementation outlined in this thesis removes, mitigates, or grants higher visibility over them. These have been detailed in the table below:

TABLE 6-1 RESULTS OF IMPLEMENTATION FINDINGS

Issue	Status	Associated Chapter
Inexact Synchronization	Mitigated	5.2
Ladder Diagram Avalanche Effect	Removed	5.3
Simultaneous Events	Higher Visibility	5.4

By using the Discrete Event Control Kit, designers may code component, interaction and specification models that fully define the behaviour of the plant and the behaviour that must be enforced. Once the automata have been created, a maximally permissive, non-blocking supervisory controller is automatically generated and stored as a state transition table. The state transition table may be quickly converted into C code by using post processing techniques such as python based scripts. The converted state transition table may then be uploaded to the microcontroller and the system will execute an Evaluation Cycle that has been measured to take approximately 0.88ms to complete.

6.2 FUTURE WORK

While some of the issues inherent to PLC implementations have been either removed or mitigated, some important issues still remain. This sub-chapter outlines a rough idea of

how new technology can help mitigate or remove the issues explored in this thesis even further.

This implementation does not take into account the selection between multiple controllable events (known as the issue of Choice) enabled in the same state. The current implementation simply selects the highest priority event. In Chapter 4.5.2.2, the code required to trigger a controllable event is defined as:

```
if(AZ_CW_MOVE_ENABLED){
        AZ_CW_MOVE();

        //Reset Variables, Find Next State
return;
}
```

This definition can be expanded to include a priority structure. By adding an additional function that returns a Boolean value, users may further expand upon the issue of choice by applying a selection policy. The event must then be enabled and the higher priority in the queue in order to be executed.

The microcontroller used in this implementation is limited to a single CPU core. By increasing the amount of cores, it's possible to have one core dedicated to the execution of a supervisory controller, while another core samples and converts the input signals required for the triggering of events. A shared memory can be used between the two devices to ensure a proper communication path exists. Additionally, this implementation limits the periodic polling of inputs to 250ms. By decreasing this value, smaller delays between the occurrence of events and their triggering can be achieved.

Field Programmable Gate Arrays (FPGAs) have been used in ever more increasing frequency. By programming a set of codes in VHDL, an implementer is capable of transferring certain software implementations into hardware implementations. As a general rule, hardware executes much faster than its software counterpart. The Plant and Supervisor explored in this thesis are automatons that can be expressed as finite state machines. In FPGAs, any finite state machine may be transformed into an equivalent hardware state machine. It is therefore logical to conclude, that an FPGA implementation of a supervisory controller operating over a plant may yield even faster triggering delays. This would further decrease the inexact synchronization between the supervisor and the plant.

CHAPTER 7. BIBLIOGRAPHY

[1] A. Brandin, "The Real-Time Supervisory Control of an Experimental Manufacturing Cell," *IEEE Transactions on Robotics and Automation,* vol. 12, no. 1, pp. 1-14, 1996.

[2] M. Fabian and A. Hellgren, "PLC-Based implementation of supervisory control for discrete event systems," in *Proc 37th IEEE Conf. Decision Control*, Tampa, FL, USA, pp. 3305-3310, Dec 1998.

[3] R. J. Leduc, "PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective," *M.S. Thesis, Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, ON, Canada,* 1996..

[4] W. Wonham and P. Ramadge, "On the supremal Controllable Sublanguage of a Given language," in *SIAM Journal of Control and Optimization, vol. 25, no 3, 637-659*, 1987.

[5] "PLC Solutions," http://plc-solutions.blogspot.ca/p/block-diagram-of-plc.html, 2 Dec 2016. [Online].

[6] "PLC Academy," http://www.plcacademy.com/ladder-logic-examples/, 14 07 2016. [Online].

[7] "PLC Real," http://controlreal.com/en/memory-and-scan-cycle/, 14 07 2016. [Online].

[8] "Circuits Today," http://www.circuitstoday.com/microprocessor-and-microcontroller, 17 09 2016. [Online].

[9] A. D. Vieira, E. A. Santos, M. H. de Queiroz, A. B. Leal, A. D. P. Neto, and J. E. R. Cury, "A Method for PLC Implementation of Supervisory Control of Discrete Event Systems," *IEEE Transactions on Control Systems Technology,* vol. 25, no. 1, pp. 175-191, 2017.

[10] S. Balemi, "Control of discrete event systems: Theory and application," *Ph.D. dissertation, Autom. Control Lab., Swiss Federal Inst. Technol.,* May, 1992.

[11] R. Kamphuis, "Design and real-time implementation of a supervisory controller for baggage handling at Veghel Airport," *M.S. Thesis, Dept. of Mech. Eng. Systems Eng. Group, Eindhoven University of Technology,* 2013.

[12] M. M. Wood, "Application, Implementation and Integration of Discrete-Event Systems Control Theory," *M.S. Thesis, Dept. Elec. Comp. Eng, Queen's University, Kingston, Ontario, Canada,* 2005.

[13] Y. L. Kaszubowski, A. B. Leal, R. Rosso Jr. and E. Harbs, "Local Modular Supervisory Implementation in Microcontroller," *9th International Conference of Modeling Optimization and Simulation, Bordeaux, France,* 2012.

[14] C. G. Cassandras and S. Lafortune, Introduction to Discrete Event Systems,, 2008, Springer.

[15] S. Hashtrudi Zad, Discrete Event Control Kit (DECK 1.2013.11) User Manual, Montreal, 2013.

[16] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete-event systems," *SIAM Journal of Control Optimization,* vol. 25, no. 1, pp. 206-230, 1987.

[17] "Flex Solar Cells," http://www.flexsolarcells.com/, 5 December 2016. [Online].

[18] "Spark Fun," https://cdn.sparkfun.com/assets/6/7/4/7/9/5112a224ce395fb479000003.png, 6 12 2016. [Online].

[19] J. Geurts, "Supervisory control of MRI subsystems," *M.S. Thesis, Dept. of Mech. Eng. and Systems Eng. Group, University of Eindhoven,* 2012.

[20] B. A. Brandin and W. M. Wonham, "Discrete-event systems supervisor control applied to the management of

manufacturing workcells," *7th International Conference on Computer Aided Manufacturing Engineering,* pp. 527-536, 1991.

[21] L. Grigorov, "Control of dynamic discrete-event systems," *M.S. Thesis, School of Computing, Queens University, Kingston, Ontario, Canada,* 2004.

[22] H. Schildt, C, The Complete Reference, Fourth Edition, Berkeley, California: Osborne, 2000.

[23] J. Doland and J. Valett, "C Style Guide," NASA, Software Engineering Laboratory Series, Greenbelt, Maryland, 1994.

[24] S. Schmit, "Getting Started with EFM32 Zero Gecko ARM Cortex-M0+," 8 Dec 2016. [Online]. Available: https://eewiki.net/pages/viewpage.action?pageId=33882195.

[25] S. Balemi, J. Hoffmann, P. Gyugyi, H. Wong-Toi and G. F. Franklin, "Supervisory control of a rapid thermal multiprocessor," *IEEE Trans. Autom. Control,,* vol. 38, no. 7, p. 1040–1059, 1993.

[26] S. C. Lauzon, J. K. Mills and B. Benhabib, "An implementation methodology for the supervisory control of flexible manufacturing workcells," *J. Manuf. Syst,* vol. 16, no. 1997, pp. 91-101, 1997.

[27] F. Charbonnier, H. Alla and R. David, "The Supervised Control of discrete-event dynamic systems," *IEEE Trans. Control Syst. Technol.,* vol. 7, no. 2, pp. 175-187, March, 1999.

[28] M. Summerfield, Rapid GUI Programming with Python and Qt, Prentice Hall, October 2007.

[29] D. B. Silva, E. A. P. Santos, A. D. Vieira and M. A. B. de Paula, "Application of the supervisory control theory to automated systems of multi-product manufacturing," *Proc. 12th IEEE Int. Conf. Emerg. Technol. Factory Autom.,* pp. 689-696, Sept. 2007.

[30] D. B. Silva, E. A. P. Santos, A. D. Vieira and M. A. B. de Paula, "Application of the supervisory control theory in the project of a robot-centered, variable routed system controller," *Proc. 13th IEEE Int. Conf. Emerg. Technol. Factory Autom.,* pp. 751-758, Sept. 2008.

[31] R. Kumar and V. K. Garg, Modeling and Control of Logical Discrete Event Systems., Boston, MA: Kluwer, 1995.

[32] K. Rohloff, "Sensor failure tolerant supervisory control," *44th IEEE Conference on Decision and Control,* pp. 3493-3498, 2005.

[33] P. J. Ramadge and W. M. Wonham, "The Control of Discrete Event Systems," *Proceedings of the IEEE,* vol. 77, no. 1, pp. 81-98, 1989.

[34] F. Basile, P. Chiacchio and D. Gerbasio, "On the implementation of industrial automation systems based on PLC," *IEEE Trans. Autom. Sci. Eng.,* vol. 10, no. 4, pp. 990-1003, 2013.

[35] F. Basile and P. Chiacchio, "On the implementation of supervised control of discrete event systems," *IEEE Trans. Control Syst. Technol.,* vol. 15, no. 4, pp. 725-739, 2007.

[36] U. Murat, K. Burak, G. Gokhan and B. H. Aksebzeci, "Asynchronous implementation of discrete event controllers based on safe automation Petri nets," *Int J Adv Manuf Technol,* vol. 41, pp. 595-612, 2009.

[37] "Discrete Event Control Kit (DECK)," Department of Electrical and Computer Engr. Concordia University, [Online]. Available: http://www.ece.concordia.ca/~shz/deck.

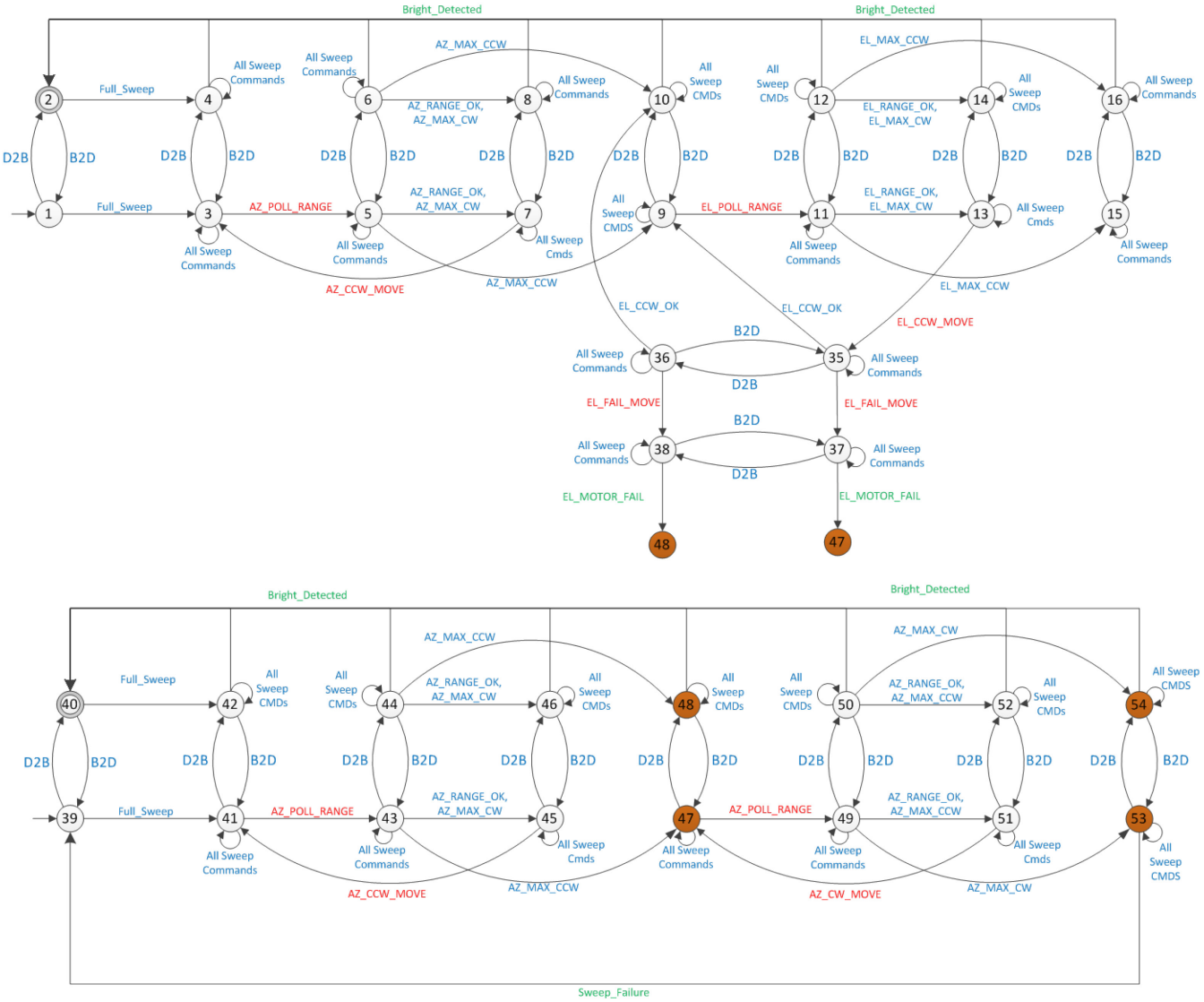
[38] W. Wonham, "Supervisory Control of Discrete Event Systems," Systems Control Group, Edward S. Rogers Sr. Dept. of Electrical and Computer Engineering, University of Toronto, Canada, 2016. [Online]. Available: http://www.control.utoronto.ca/DES..

[39] P. Dietrich, R. Malik, W. M. Wonham and B. A. Brandin, "Implementation considerations in supervisory control," in *Synthesis and Control of Discrete event Systems*, New York, NY, Springer, 2002, pp. 185-201.

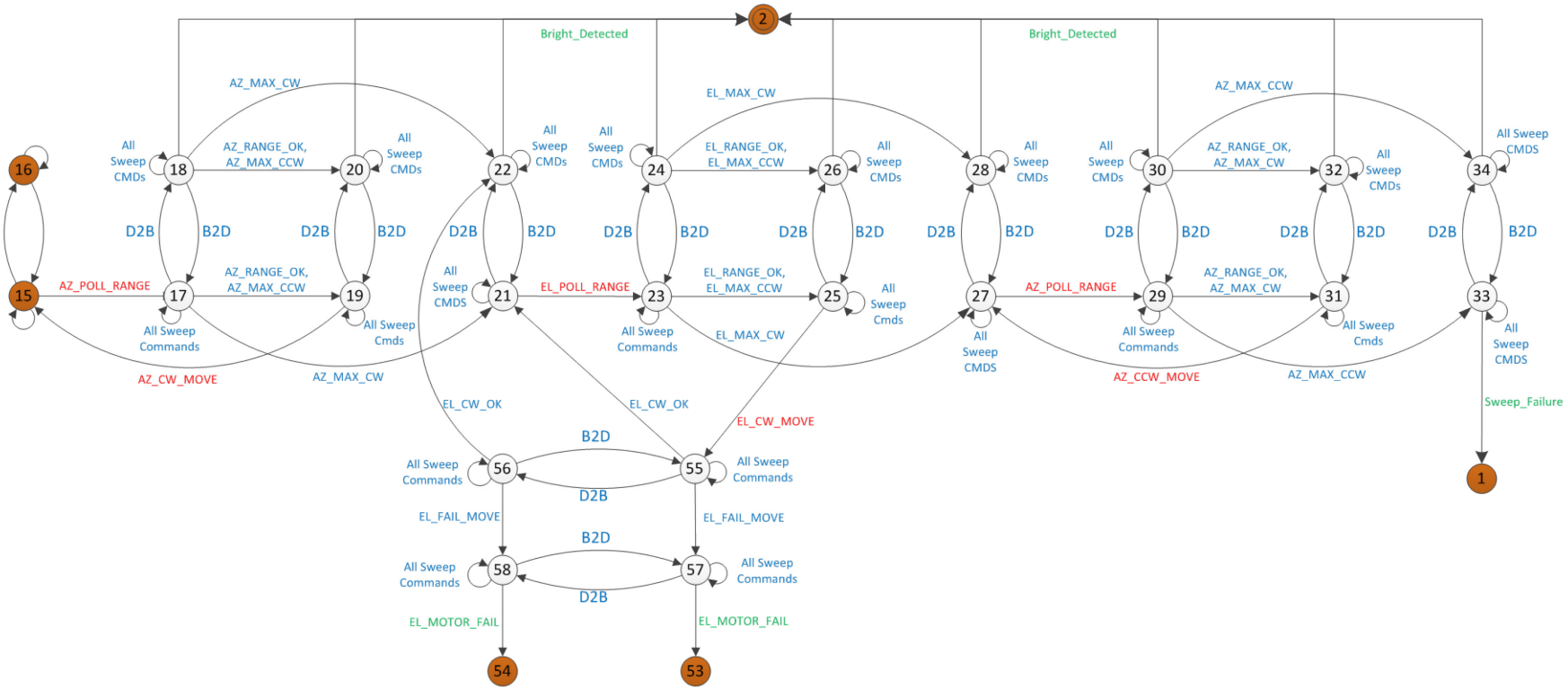
[40] S. R. Mohanty, V. Chandra and R. Kumar, "A Computer Implementable Algorithm for the Synthesis of an Optimal Controller for Acyclic Discrete Event Processes," in *Proc. 1999 IEEE Int. Conf. on Robotics & Automation*, Detroit, MI, USA, pp. 126-130, May 1999.

[41] C. G. Cassandras and S. Lafortuna, Introduction to Discrete Event Systems, New York: Springer, 2008.

APPENDIX A FULL MOVEMENT SPECIFICATION (WITH FAILURE)



Part 1


2
Bright_Detected
Bright_Detected
AZ_MAX_CW
EL_MAX_CW
AZ_MAX_CCW
16
18
20
22
24
26
28
30
32
34
All Sweep CMDs
AZ_RANGE_OK, AZ_MAX_CCW
EL_RANGE_OK, EL_MAX_CCW
AZ_RANGE_OK, AZ_MAX_CW
D2B
B2D
15
17
19
21
23
25
27
29
31
33
AZ_POLL_RANGE
EL_POLL_RANGE
All Sweep Commands
All Sweep Cmds
EL_MAX_CW
AZ_CW_MOVE
AZ_MAX_CW
AZ_CCW_MOVE
AZ_MAX_CCW
Sweep_Failure
1
EL_CW_OK
EL_CW_OK
EL_CW_MOVE
56
55
58
57
EL_FAIL_MOVE
EL_MOTOR_FAIL
54
53

Part 2

APPENDIX B DISCRETE EVENT CONTROL KIT (DECK)

DECK is a toolbox written in MATLAB for the analysis and synthesis of supervisory control problem of discrete-event systems. In this appendix we review the functions of this toolbox [15]

1. Automaton: Creates an automaton model for use by the toolbox (DECK). Number of states, the matrix of transition list and a vector of marked states are the inputs to this function.
2. Automatonchk: This function verifies the validity of the automaton model. It gets the automaton model as input.
3. Complement: Returns the complement of the input deterministic automaton.
4. Controllable: This function determines if the input automaton is controllable with respect to the second input automaton and the input vector of uncontrollable events.
5. Deterministic: Converts the nondeterministic input automaton to a deterministic automaton.
6. Isnondet: This function determines if the input automaton is nondeterministic.
7. Product: Computes the product of a finite number of input automata.
8. Project: Finds a deterministic automaton to represent the projected model of the input automaton and the input vector of unobservable events.
9. Reach: Computes the reachable states of a transition list matrix and the vector of source states.
10. Reachable: This function computes the reachable subautomaton of an input automaton.
11. Seloop: Adds seloop to the input automaton.
12. Supcon: Finds the supremal controllable sublanguage of the first input automaton with respect to the second automaton and the input vector of uncontrollable events.
13. Sync: This function computes the synchronous product of a finite number of input automata.
14. Trim: Finds the trim (reachable and coreachable) subautomaton of the input automaton.

APPENDIX C DECK CODE

```
% This implementation is a dual motor (azimuth + elevation) implementation.
% This implementation ONLY includes the "Full_Sweep" ROBUST control model.
% X states
% Y double STT entries
%
%
%Author: Kevin J. Searle, October 1st, 2016.
%


%
% Components
%

%Battery State of Charge

Safe_to_Full = 601;
Full_to_Safe = 602;
Crit_to_Safe = 603;
Safe_to_Crit = 604;

Bat_SOC_Marked_States = [2];
Bat_SOC_STT = [1 Safe_to_Full 2 ; 2 Full_to_Safe 1 ; 3 Crit_to_Safe 1 ; 1
Safe_to_Crit 3];
Bat_SOC = automaton(3, Bat_SOC_STT, Bat_SOC_Marked_States);

%
% PV Cell Illumination
%

Dark_to_Dim = 301;
Dim_to_Bright = 302;
Dim_to_Dark = 303;
Bright_to_Dim = 304;

PV_Marked_States = [3];
PV_STT = [1 Dark_to_Dim 2 ; 2 Dim_to_Bright 3 ; 3 Bright_to_Dim 2 ; 2
Dim_to_Dark 1];
PV_Cell = automaton(3, PV_STT, PV_Marked_States);
```

```
%
% Motor Motion
%

% Azimuth
AZ_CCW_OK = 401;
AZ_CW_OK = 402;
AZ_CCW_MOVE = 403;
AZ_CW_MOVE = 404;

AZ_Motor_Motion_Marked_States = [1];
AZ_Motor_Motion_STT = [1 AZ_CW_MOVE 2 ; 2 AZ_CW_OK 1 ; 1 AZ_CCW_MOVE 3 ; 3
AZ_CCW_OK 1];
AZ_Motor_Motion = automaton(3, AZ_Motor_Motion_STT,
AZ_Motor_Motion_Marked_States);



%Elevation
EL_CCW_OK = 451;
EL_CW_OK = 452;
EL_CCW_MOVE = 453;
EL_CW_MOVE = 454;
EL_FAIL_MOVE = 455;



%With Failure Implementation (Adds EL_FAIL_MOVE and a fourth state)
EL_Motor_Motion_Marked_States = [1,4];
EL_Motor_Motion_STT = [1 EL_CW_MOVE 2 ; 2 EL_CW_OK 1 ; 1 EL_CCW_MOVE 3 ; 3
EL_CCW_OK 1; 2 EL_FAIL_MOVE 4; 3 EL_FAIL_MOVE 4];
EL_Motor_Motion = automaton(4, EL_Motor_Motion_STT,
EL_Motor_Motion_Marked_States);

%
% Motor Range
%

%Azimuth
AZ_MAX_CW = 410;
```

```
AZ_MAX_CCW = 411;
AZ_RANGE_OK = 412;
AZ_POLL_RANGE = 425;

AZ_Motor_Range_Marked_States = [1,2,3];
AZ_Motor_Range_STT = [1 AZ_POLL_RANGE 4; 4 AZ_RANGE_OK 1; 4 AZ_MAX_CW 2; 4
AZ_MAX_CCW 3; 3 AZ_POLL_RANGE 4; 2 AZ_POLL_RANGE 4];
AZ_Motor_Range = automaton(4, AZ_Motor_Range_STT,
AZ_Motor_Range_Marked_States);



%Elevation
EL_MAX_CW = 460;
EL_MAX_CCW = 461;
EL_RANGE_OK = 462;
EL_POLL_RANGE = 435;

EL_Motor_Range_Marked_States = [1,2,3];
EL_Motor_Range_STT = [1 EL_POLL_RANGE 4; 4 EL_RANGE_OK 1; 4 EL_MAX_CW 2; 4
EL_MAX_CCW 3; 3 EL_POLL_RANGE 4; 2 EL_POLL_RANGE 4];
EL_Motor_Range = automaton(4, EL_Motor_Range_STT,
EL_Motor_Range_Marked_States);



%
% Master Controller
%

Full_Sweep = 504;
Bright_Detected = 510;
Sweep_Failure = 511;
EL_MOTOR_FAIL = 512;

%With Failure Implementation (Adds EL_MOTOR_FAIL signal to inform Master
%Control that we have encountered a motor failure.

MC_Marked_States = [1];
MC_STT = [1 Full_Sweep 1; 1 Bright_Detected 1; 1 Sweep_Failure 1 ; 1
EL_MOTOR_FAIL 1];
```

```
MC = automaton(1, MC_STT, MC_Marked_States);



%
% Interactions
%

%The Motors cannot move when the battery state of charge is Critical.
Mot_Motion_f_Bat_SOC_Marked_States = [1, 2];
Mot_Motion_f_Bat_SOC_STT = [Bat_SOC_STT; 2 AZ_CCW_OK 2; 2 AZ_CW_OK 2; 2
AZ_CW_MOVE 2; 2 AZ_CCW_MOVE 2; 3 AZ_CCW_OK 3; 3 AZ_CW_OK 3; 1 AZ_CCW_OK 1; 1
AZ_CW_OK 1; 1 AZ_CW_MOVE 1; 1 AZ_CCW_MOVE 1];
Mot_Motion_f_Bat = automaton(4, Mot_Motion_f_Bat_SOC_STT,
Mot_Motion_f_Bat_SOC_Marked_States);

%The Battery State of Charge varies as a function of the brightness on the
%PV Cell.

%Battery State of charge as a function of Solar Cell Brightness

Bat_SOC_f_PV_Marked_States = [3];
Bat_SOC_f_PV_STT = [PV_STT; 1 Safe_to_Crit 1; 1 Full_to_Safe 1; 2 Safe_to_Crit
2; 2 Full_to_Safe 2; 2 Crit_to_Safe 2; 2 Safe_to_Full 2; 3 Safe_to_Crit 3; 3
Full_to_Safe 3; 3 Crit_to_Safe 3; 3 Safe_to_Full 3];
Bat_SOC_f_PV = automaton(3, Bat_SOC_f_PV_STT, Bat_SOC_f_PV_Marked_States);



%The Battery State of Charge varies as a function of the motor state (both
%EL and AZ)
%Battery State of charge as a function of Solar Cell Brightness

[Bat_SOC_f_Motor_Motions, states] = sync(AZ_Motor_Motion, EL_Motor_Motion);
for i=1:size(states,1)

   if (states(i,1) == 1  && states(i,2) == 1)
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Safe_to_Full
i];
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Full_to_Safe
i];
```

```
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Crit_to_Safe
i];
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Safe_to_Crit
i];

   elseif (states(i,1) == 1  && states(i,2) == 4)
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Safe_to_Full
i];
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Full_to_Safe
i];
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Crit_to_Safe
i];
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Safe_to_Crit
i];
   else
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Safe_to_Crit
i];
     Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Full_to_Safe
i];
      end
end

%Specifications:

%Motor Motion and Motor Range need to be synchronized - This is performed
%as a Specification as we are imposing a set of rules on how the system
%behaves.

%The motors cannot move when CCW when in the Max CCW state.
%The motors cannot move CW when in the Max CW state.
%When the motors are "In Range" there is no limit on the motion

%Azimuth

Spec_AZ_M_Motion_f_M_Range_MarkedStates = [1,2,3];
Spec_AZ_M_Motion_f_M_Range_STT = [AZ_Motor_Range_STT; 1 AZ_CW_MOVE 1; 1
AZ_CCW_MOVE 1; 2 AZ_CCW_MOVE 2; 3 AZ_CW_MOVE 3];
Spec_AZ_M_Motion_f_M_Range = automaton(4, Spec_AZ_M_Motion_f_M_Range_STT,
Spec_AZ_M_Motion_f_M_Range_MarkedStates);
```

```
E_Not_In_AZ_M_Motion_f_M_Range_Spec = [Dark_to_Dim, Dim_to_Dark,
Bright_to_Dim, Dim_to_Bright, Safe_to_Full, Full_to_Safe, Crit_to_Safe,
Safe_to_Crit, Full_Sweep, Bright_Detected, Sweep_Failure, EL_MOTOR_FAIL,
AZ_CCW_OK, AZ_CW_OK, EL_POLL_RANGE, EL_RANGE_OK, EL_MAX_CW, EL_MAX_CCW,
EL_FAIL_MOVE, EL_CW_MOVE, EL_CCW_MOVE, EL_CW_OK, EL_CCW_OK];
Spec_AZ_M_Motion_f_M_Range_SelfLooped = selfloop(Spec_AZ_M_Motion_f_M_Range,
E_Not_In_AZ_M_Motion_f_M_Range_Spec);

%Elevation

Spec_EL_M_Motion_f_M_Range_MarkedStates = [1,2,3];
Spec_EL_M_Motion_f_M_Range_STT = [EL_Motor_Range_STT; 1 EL_CW_MOVE 1; 1
EL_CCW_MOVE 1; 2 EL_CCW_MOVE 2; 3 EL_CW_MOVE 3];
Spec_EL_M_Motion_f_M_Range = automaton(4, Spec_EL_M_Motion_f_M_Range_STT,
Spec_EL_M_Motion_f_M_Range_MarkedStates);

E_Not_In_EL_M_Motion_f_M_Range_Spec = [Dark_to_Dim, Dim_to_Dark,
Bright_to_Dim, Dim_to_Bright, Safe_to_Full, Full_to_Safe, Crit_to_Safe,
Safe_to_Crit, Full_Sweep, Bright_Detected, Sweep_Failure, EL_MOTOR_FAIL,
EL_FAIL_MOVE, AZ_CCW_OK, AZ_CW_OK, AZ_CW_MOVE, AZ_CCW_MOVE, AZ_POLL_RANGE,
AZ_RANGE_OK, AZ_MAX_CW, AZ_MAX_CCW, EL_CCW_OK, EL_CW_OK];
Spec_EL_M_Motion_f_M_Range_SelfLooped = selfloop(Spec_EL_M_Motion_f_M_Range,
E_Not_In_EL_M_Motion_f_M_Range_Spec);



%Motor Range as a function of Motor Motion
%When the Motors are Turning CW, it can generate the "AZ_CW_OK" signal, all
other signals are suppresed. (Cannot add - will make it non-deterministic)
%When the Motors are Turning CCW, it can generate the "AZ_CCW_OK" signal, all
other signals are suppresed. (Cannot add - will make it non-deterministic)
%When the Motors are Idle, it can generate the AZ_MAX_CCW and AZ_MAX_CW
signals
%When the motor has Failed all signals are suppressed.

%Azimuth
Spec_AZ_M_Range_f_M_Motion_MarkedStates = [1];
```

```
Spec_AZ_M_Range_f_M_Motion_MarkedStates_STT = [AZ_Motor_Motion_STT; 1
AZ_MAX_CW 1; 1 AZ_MAX_CCW 1; 1 AZ_RANGE_OK 1; 1 AZ_POLL_RANGE 1];
Spec_AZ_M_Range_f_M_Motion = automaton(3,
Spec_AZ_M_Range_f_M_Motion_MarkedStates_STT,
Spec_AZ_M_Range_f_M_Motion_MarkedStates);

E_Not_In_AZ_M_Range_f_M_Motion_Spec = [Dark_to_Dim, Dim_to_Dark,
Bright_to_Dim, Dim_to_Bright, Safe_to_Full, Full_to_Safe, Crit_to_Safe,
Safe_to_Crit, Full_Sweep, Bright_Detected,Sweep_Failure, EL_MOTOR_FAIL,
EL_CW_MOVE, EL_CCW_MOVE, EL_CW_OK, EL_CCW_OK, EL_FAIL_MOVE, EL_POLL_RANGE,
EL_RANGE_OK, EL_MAX_CW, EL_MAX_CCW];
Spec_AZ_M_Range_f_M_Motion_SelfLooped = selfloop(Spec_AZ_M_Range_f_M_Motion,
E_Not_In_AZ_M_Range_f_M_Motion_Spec);

%Elevation
Spec_EL_M_Range_f_M_Motion_MarkedStates = [1,4];
Spec_EL_M_Range_f_M_Motion_MarkedStates_STT = [EL_Motor_Motion_STT; 1
EL_MAX_CW 1; 1 EL_MAX_CCW 1; 1 EL_RANGE_OK 1; 1 EL_POLL_RANGE 1];
Spec_EL_M_Range_f_M_Motion = automaton(4,
Spec_EL_M_Range_f_M_Motion_MarkedStates_STT,
Spec_EL_M_Range_f_M_Motion_MarkedStates);

E_Not_In_EL_M_Range_f_M_Motion_Spec = [Dark_to_Dim, Dim_to_Dark,
Bright_to_Dim, Dim_to_Bright, Safe_to_Full, Full_to_Safe, Crit_to_Safe,
Safe_to_Crit, Full_Sweep, Bright_Detected,Sweep_Failure, EL_MOTOR_FAIL,
AZ_CCW_OK, AZ_CW_OK, AZ_CW_MOVE, AZ_CCW_MOVE, AZ_POLL_RANGE, AZ_RANGE_OK,
AZ_MAX_CW, AZ_MAX_CCW];
Spec_EL_M_Range_f_M_Motion_SelfLooped = selfloop(Spec_EL_M_Range_f_M_Motion,
E_Not_In_EL_M_Range_f_M_Motion_Spec);



%The final spec defines the behaviour of the system should take when a
%Sweep command is received.

Spec_Total_States = 58;
Sweep_Spec_MarkedStates = [2,40];
Sweep_Spec_STT = [1 Dim_to_Bright 2 ; 2 Bright_to_Dim 1 ; 1 Full_Sweep 3 ; 2
Full_Sweep 4 ; 4 Bright_Detected 2 ; 3 Dim_to_Bright 4 ; 4 Bright_to_Dim 3 ; 3
AZ_POLL_RANGE 5 ; 5 Dim_to_Bright 6 ; 6 Bright_to_Dim 5 ; 5 AZ_RANGE_OK 7 ; 5
```

AZ_MAX_CW 7 ; 7 AZ_CCW_MOVE 3 ; 7 Dim_to_Bright 8 ; 8 Bright_to_Dim 7 ; 5
AZ_MAX_CCW 9 ; 9 Dim_to_Bright 10 ; 10 Bright_to_Dim 9 ; 9 EL_POLL_RANGE 11 ;
11 Dim_to_Bright 12 ; 12 Bright_to_Dim 11 ; 11 EL_RANGE_OK 13 ; 11 EL_MAX_CW
13 ; 13 Dim_to_Bright 14 ; 13 EL_CCW_MOVE 35 ; 14 Bright_to_Dim 13 ; 11
EL_MAX_CCW 15 ; 15 Dim_to_Bright 16 ; 16 Bright_to_Dim 15 ; 15 AZ_POLL_RANGE
17 ; 17 Dim_to_Bright 18 ; 18 Bright_to_Dim 17 ; 17 AZ_RANGE_OK 19 ; 17
AZ_MAX_CCW 19 ; 19 Dim_to_Bright 20 ; 19 AZ_CW_MOVE 15 ; 20 Bright_to_Dim 19 ;
17 AZ_MAX_CW 21 ; 21 Dim_to_Bright 22 ; 22 Bright_to_Dim 21 ; 21 EL_POLL_RANGE
23 ; 23 Dim_to_Bright 24 ; 24 Bright_to_Dim 23 ; 23 EL_RANGE_OK 25 ; 23
EL_MAX_CCW 25 ; 25 Dim_to_Bright 26 ; 25 EL_CW_MOVE 55 ; 26 Bright_to_Dim 25 ;
23 EL_MAX_CW 27 ; 27 Dim_to_Bright 28 ; 28 Bright_to_Dim 27 ; 27 AZ_POLL_RANGE
29 ; 29 Dim_to_Bright 30 ; 30 Bright_to_Dim 29 ; 29 AZ_RANGE_OK 31 ; 29
AZ_MAX_CW 31 ; 29 AZ_MAX_CCW 33 ; 31 AZ_CCW_MOVE 27 ; 31 Dim_to_Bright 32 ; 32
Bright_to_Dim 31 ; 33 Dim_to_Bright 34 ; 34 Bright_to_Dim 33 ; 35 EL_CCW_OK 9
; 35 Dim_to_Bright 36 ; 36 Bright_to_Dim 35 ; 36 EL_CCW_OK 10 ; 36
EL_FAIL_MOVE 38 ; 35 EL_FAIL_MOVE 37 ; 37 Dim_to_Bright 38 ; 38 Bright_to_Dim
37 ; 38 EL_MOTOR_FAIL 48 ; 37 EL_MOTOR_FAIL 47 ; 55 EL_CW_OK 21 ; 55
Dim_to_Bright 56 ; 56 Bright_to_Dim 55 ; 56 EL_CW_OK 22 ; 55 EL_FAIL_MOVE 57 ;
57 Dim_to_Bright 58 ; 58 Bright_to_Dim 57 ; 56 EL_FAIL_MOVE 58 ; 58
EL_MOTOR_FAIL 54 ; 57 EL_MOTOR_FAIL 53 ; 39 Full_Sweep 41 ; 40 Full_Sweep 42 ;
41 AZ_POLL_RANGE 43 ; 43 AZ_RANGE_OK 45 ; 43 AZ_MAX_CW 45 ; 45 AZ_CCW_MOVE 41
; 43 AZ_MAX_CCW 47 ; 47 AZ_POLL_RANGE 49 ; 49 AZ_RANGE_OK 51 ; 49 AZ_MAX_CCW
51 ; 39 Dim_to_Bright 40 ; 40 Bright_to_Dim 39 ; 41 Dim_to_Bright 42 ; 42
Bright_to_Dim 41 ; 43 Dim_to_Bright 44 ; 44 Bright_to_Dim 43 ; 45
Dim_to_Bright 46 ; 46 Bright_to_Dim 45 ; 47 Dim_to_Bright 48 ; 48
Bright_to_Dim 47 ; 49 Dim_to_Bright 50 ; 50 Bright_to_Dim 49 ; 51
Dim_to_Bright 52 ; 51 AZ_CW_MOVE 47; 52 Bright_to_Dim 51 ; 49 AZ_MAX_CW 53 ;
53 Dim_to_Bright 54 ; 54 Bright_to_Dim 53 ; 53 Sweep_Failure 39 ; 33
Sweep_Failure 1 ; 6 Bright_Detected 2 ; 8 Bright_Detected 2 ; 10
Bright_Detected 2 ; 12 Bright_Detected 2 ; 14 Bright_Detected 2 ; 16
Bright_Detected 2 ; 18 Bright_Detected 2 ; 20 Bright_Detected 2 ; 22
Bright_Detected 2 ; 24 Bright_Detected 2 ; 26 Bright_Detected 2 ; 28
Bright_Detected 2 ; 30 Bright_Detected 2 ; 32 Bright_Detected 2 ; 34
Bright_Detected 2 ; 42 Bright_Detected 40 ; 44 Bright_Detected 40 ; 46
Bright_Detected 40 ; 48 Bright_Detected 40 ; 50 Bright_Detected 40 ; 52
Bright_Detected 40 ; 54 Bright_Detected 40 ; 3 Full_Sweep 3 ; 4 Full_Sweep 4 ;
5 Full_Sweep 5 ; 6 Full_Sweep 6 ; 7 Full_Sweep 7 ; 8 Full_Sweep 8 ; 9
Full_Sweep 9 ; 10 Full_Sweep 10 ; 11 Full_Sweep 11 ; 12 Full_Sweep 12 ; 13
Full_Sweep 13 ; 14 Full_Sweep 14 ; 15 Full_Sweep 15 ; 16 Full_Sweep 16 ; 17
Full_Sweep 17 ; 18 Full_Sweep 18 ; 19 Full_Sweep 19 ; 20 Full_Sweep 20 ; 21
Full_Sweep 21 ; 22 Full_Sweep 22 ; 23 Full_Sweep 23 ; 24 Full_Sweep 24 ; 25

```
Full_Sweep 25 ; 26 Full_Sweep 26 ; 27 Full_Sweep 27 ; 28 Full_Sweep 28 ; 29
Full_Sweep 29 ; 30 Full_Sweep 30 ; 31 Full_Sweep 31 ; 32 Full_Sweep 32 ; 33
Full_Sweep 33 ; 34 Full_Sweep 34 ; 35 Full_Sweep 35 ; 36 Full_Sweep 36 ; 37
Full_Sweep 37 ; 38 Full_Sweep 38 ; 41 Full_Sweep 41 ; 42 Full_Sweep 42 ; 43
Full_Sweep 43 ; 44 Full_Sweep 44 ; 45 Full_Sweep 45 ; 46 Full_Sweep 46 ; 47
Full_Sweep 47 ; 48 Full_Sweep 48 ; 49 Full_Sweep 49 ; 50 Full_Sweep 50 ; 51
Full_Sweep 51 ; 52 Full_Sweep 52 ; 53 Full_Sweep 53 ; 54 Full_Sweep 54 ; 55
Full_Sweep 55 ; 56 Full_Sweep 56 ; 57 Full_Sweep 57 ; 58 Full_Sweep 58; 6
AZ_RANGE_OK 8; 6 AZ_MAX_CW 8; 6 AZ_MAX_CCW 10; 12 EL_RANGE_OK 14; 12 EL_MAX_CW
14; 12 EL_MAX_CCW 16; 18 AZ_RANGE_OK 20; 18 AZ_MAX_CW 20; 18 AZ_MAX_CCW 22; 24
EL_RANGE_OK 26; 24 EL_MAX_CW 26; 24 EL_MAX_CCW 28; 30 AZ_RANGE_OK 32; 30
AZ_MAX_CW 32; 30 AZ_MAX_CCW 34; 44 AZ_RANGE_OK 46; 44 AZ_MAX_CW 46; 44
AZ_MAX_CCW 48; 50 AZ_RANGE_OK 52; 50 AZ_MAX_CW 52; 50 AZ_MAX_CCW 54];
Sweep_Spec_Automata = automaton(Spec_Total_States, Sweep_Spec_STT,
Sweep_Spec_MarkedStates);

E_Not_In_Sweep_Spec = [AZ_CCW_OK, AZ_CW_OK, Dark_to_Dim, Dim_to_Dark,
Safe_to_Full, Full_to_Safe, Crit_to_Safe, Safe_to_Crit];

Sweep_Spec_Automata_SelfLooped = selfloop(Sweep_Spec_Automata,
E_Not_In_Sweep_Spec);

%Sweep_Supervisor = supcon(Sweep_Spec_Automata_SelfLooped, Plant, Euc);
%do I create a product of the supervisors before or after we create
%"supcon"

%AZ_RANGE_OK, AZ_MAX_CCW, AZ_MAX_CW, EL_RANGE_OK, EL_MAX_CCW, EL_MAX_CW
Euc = [Full_Sweep, Dark_to_Dim, Dim_to_Bright, Dim_to_Dark, Bright_to_Dim,
AZ_CCW_OK, AZ_CW_OK, EL_CW_OK, EL_CCW_OK, EL_FAIL_MOVE, Safe_to_Full,
Full_to_Safe, Crit_to_Safe, Safe_to_Crit, AZ_RANGE_OK, AZ_MAX_CCW, AZ_MAX_CW,
EL_RANGE_OK, EL_MAX_CCW, EL_MAX_CW];

Plant = sync(PV_Cell, Bat_SOC, AZ_Motor_Motion, EL_Motor_Motion,
AZ_Motor_Range, EL_Motor_Range, MC, Mot_Motion_f_Bat, Bat_SOC_f_PV,
Bat_SOC_f_Motor_Motions);

 SPEC = product(Sweep_Spec_Automata_SelfLooped,
Spec_EL_M_Range_f_M_Motion_SelfLooped, Spec_AZ_M_Range_f_M_Motion_SelfLooped,
Spec_EL_M_Motion_f_M_Range_SelfLooped, Spec_AZ_M_Motion_f_M_Range_SelfLooped);
```

```
Supervisor = supcon(SPEC, Plant, Euc);
```

APPENDIX D PHOTOGRAPH OF THE DUAL AXIS SOLAR TRACKING SYSTEM AND THE GRAPHIC USER INTERFACE FOR THE MASTER CONTROLLER