Pseudo State Machine Programming Language

by

Campbell Malcolm Strachan

A thesis submitted to the Murdoch University
to fulfil the requirements for the degree of
Bachelor of Engineering Honours
In the discipline of
Electrical Power and Industrial Computer Systems

Perth, 2017

© Murdoch University 2017
Author’s Declaration

I declare that this thesis is my own account of my research and contains as its main content work which has not previously been submitted for a degree at any tertiary education institution.

_______________________
Campbell Malcolm Strachan

WORD COUNT: 15398
Abstract

Programmers today have many tools at their disposal to increase their effectiveness, use of time, code integrity and more. These tools can be anything from the Interactive Development Environments (IDEs) that attempt to fill in code, document particular functions and point out possible errors. Other tools include programming methods or processes that help the developer in constructing and designing their code quicker and to a higher degree of robustness. The main aim of this thesis is to create an additional tool for the students completing the Industrial Computer Systems Engineering course at Murdoch University. This tool is a Pseudo State Machine Programming Language that allows for easy development and use of the State Machine method. Subsequent to this was, the creation of a simulation application for the debugging and development of the language with the intention of it also being used by the students for their project developments. An additional goal was an investigation into the feasibility of the Arduino Uno replacing or being an addition into the ICSE units. In this thesis, the Pseudo State Machine Programming Language was developed by extending the FORTH compiler in SwiftX. It works in the intended way by making the development of simple and complicated state machine comparatively easy. The comparison has been made to an implementation of code that achieves the same end results and factored in the development time and the complexity of reading and understanding the code. The simulation created also provided the help required in completing the language within the timeframe of this project. It is also considered to be adequate for use by students for their project development. After testing the Arduino Uno, it was found that it was not a suitable replacement but may be a valuable addition to the ICSE unit. The Pseudo Language and the Simulator enhance the learning experience by making development of projects easier. The Arduino Uno exposes the students to a different type of microcontroller which increases their knowledge and experiences on the subject.
Acknowledgements

I would like to thank my supervisor, Graeme Cole, for his guidance and supervision on this thesis. Also, thank you to the Murdoch university technicians, Will Stirling and Iafeta Laava (Jeff) for their assistance throughout this thesis.

To all the friends I have made at university, I thank you for being there to make me laugh and get through the hard times. A special thanks to Harold, Ben and Allison for making the mechatronics room a somewhat bearable place to live for the past year. Thanks for keeping me on track and lifting my spirits and providing constant motivation.

Finally, and perhaps most importantly I have to thank my families who have supported me throughout my whole engineering degree. Thank you to my siblings for feeding me and letting me crash at their houses. A special thank you to my parents. Thank you for believing in me and providing encouragement in moments of need and for making this possible.
# Table of Contents

Author’s Declaration ........................................................................................................ iii
Abstract ............................................................................................................................ v
Acknowledgements ........................................................................................................ vii
Table of Contents ............................................................................................................. ix
List of Figures .................................................................................................................. xi
List of Tables ................................................................................................................... xiii
List of Abbreviations ....................................................................................................... xv

Chapter 1  Introduction ..................................................................................................... 1
  1.1  Objectives ............................................................................................................... 2
  1.2  Thesis Reading convention .................................................................................... 2

Chapter 2  Background ................................................................................................... 5
  2.1  State Machines ....................................................................................................... 5
  2.2  Linked lists ............................................................................................................. 8
  2.3  Compilers and Forth .............................................................................................. 9
  2.4  Lolly Machine ....................................................................................................... 10

Chapter 3  Simulator ........................................................................................................ 13
  3.1  Requirements ........................................................................................................ 14
  3.2  Method .................................................................................................................. 14
  3.3  Result .................................................................................................................... 15
  3.3.1  Goals 1 2 and 3 ............................................................................................... 16
  3.3.2  Goal 4 .............................................................................................................. 19
  3.3.3  Goal 5 .............................................................................................................. 19
  3.3.4  Goal 6 .............................................................................................................. 20

Chapter 4  Language Development ................................................................................. 23
  4.1  Method .................................................................................................................. 24
  4.2  Individual components ........................................................................................ 25
  4.2.1  Variables .......................................................................................................... 25
  4.2.2  User Variables .................................................................................................. 27
  4.2.2.1  Result of User Variables .............................................................................. 28
  4.2.3  Machine Diagnostics ...................................................................................... 29
  4.2.3.1  Results Machine Diagnostics .................................................................... 32
  4.2.4  Defining States ............................................................................................... 32
  4.2.4.1  Requirements and limitations ................................................................... 34
  4.2.5  Transitions ...................................................................................................... 34
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pacman State Machine Example</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Lolly Machine Apparatus</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Rectangle Coordinates in Application</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Developed Simulation</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Structure of the State Data Block (SDB)</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Transition Data Node (TDN) Structure</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>TDN Creation</td>
<td>36</td>
</tr>
<tr>
<td>8</td>
<td>TDN Linked List Structure</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Pseudo Code for Machine Loop</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>Lolly Machine with Identified Components</td>
<td>53</td>
</tr>
<tr>
<td>11</td>
<td>Lolly Machine with Components Named – Upper Section</td>
<td>54</td>
</tr>
<tr>
<td>12</td>
<td>Lolly Machine with Components Named - Lower Section</td>
<td>55</td>
</tr>
<tr>
<td>13</td>
<td>Lolly Machine Dispenser Actuator Location</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>Sorting State Machine Diagram</td>
<td>61</td>
</tr>
<tr>
<td>15</td>
<td>Dispensing State Machine Diagram</td>
<td>65</td>
</tr>
<tr>
<td>16</td>
<td>Equipment Set-up for Re-Burning the Arduino Uno</td>
<td>70</td>
</tr>
</tbody>
</table>
## List of Tables

Table 1 Rectangle Array ........................................................................................................16
Table 2 Colour for Each Byte Used by CREATESOLIDBRUSH.............................................18
Table 3 Variables and Their Descriptions ..........................................................................27
Table 4 Constants and Their Descriptions ........................................................................31
Table 5 Transition True or False Outcome........................................................................38
Table 6 Stop Words and Their Descriptions ......................................................................44
Table 7 Timing Results .........................................................................................................47
Table 8 Example for Error.....................................................................................................48
Table 9 Lolly Machine Linear Actuators and Their Actions ..............................................56
Table 10 Lolly Machine Rotary Actuators and Their Actions ............................................57
Table 11 Lolly Machine Reject Air Supply Actions............................................................57
Table 12 Lolly Machine Sensors and Indications ...............................................................57
Table 13 Results of 7 Segment Display Test .....................................................................59
Table 14 Rotary Actuator Positioning Results ....................................................................64
Table 15 Summary of Objectives ........................................................................................67
List of Abbreviations

TDN – Transition Data Node

SDB – State Data Block

ICSE – Industrial Computer Systems Engineering

USB – Universal Serial Bus

AVR – a family of Atmel developed microcontrollers

ISP – In-System Programmer

RAM – Random Access Memory

IDE – Interactive Development Environments
Chapter 1  \hspace{1cm} Introduction

The aim of this thesis project is to develop a pseudo state machine programming language for students in the Industrial Computer Systems Engineering (ICSE) course at Murdoch University. The language should help students completing the third-year units in the development of their projects for assessment while also further teaching them the concept of state machines. The word ‘pseudo’ is used in the title of this thesis because it is not a language in its own right. The ‘language’ was written in another language called Forth [1] which allows for the compiler to be extended. This created language can be thought of as a unique instruction set that performs state machine operations. In order to develop the state machine language within the thesis timeframe, the creation of a simulator application was advised by the supervisor. This simulator should visually indicate the state of the state machines during operation.

One of the requirements of the language is that it should run on multiple microcontrollers that can run the SwiftX environment [2] without the need of re-burning the kernel. The Arduino Uno [3] was selected to test this feature of the language and also to investigate the possibility of this microcontroller replacing or being an addition to the microcontrollers in the ICSE laboratory. The current microcontrollers used in the ICSE units are old and more expensive than the Arduino Uno. It would be beneficial to the university to use a cheaper alternative. Also, The Arduino Uno has a lot of easily attachable extendable components [4]. This would give the students greater exposure to a variety of microcontrollers that will enhance their learning experience.

The approach of this thesis is outlined in order here:

1. Creation of the simulator
2. Development of the State Machine Language
3. Testing of the language
4. Installing the SwiftX environment on the Arduino Uno and testing the language in this alternative environment.

1.1 Objectives

1. Create a Simulator application
2. Create a State Machine Language
3. SwiftX and state machine language implementation on the Arduino Uno

These objectives will be discussed after Chapter 2 Background that introduces concepts that are used and involved in this thesis.

1.2 Thesis Reading convention

To explain some of the concepts and program some code will be represented by pseudo code. Pseudo code is used in the design of programs but also to illustrate and convey the logic of the code that can be applied somewhat universally [5].

The pseudo code is customised and unique to this thesis. There are two types. The first explains how a word, similar to a function or method in modern languages, is defined:

: WORD1

<Value, Object> WORD2 <value, object>

;
The second explains how a word is used:

<value, object> WORD3 <Value, Object>

The first example starts with the opening of the word using the colon ‘:’ character and ends in the semicolon character ‘;’. WORD1 is the word being defined. <value, object> is some value or reference, the words inside of the angle brackets explain what it is that that value represents. The second example is very similar except it is not within the two colons. This is because it is used in the terminal at run time rather than a word being defined. WORD2 is a word already defined that will run on the execution of WORD1. WORD3 is a word that is already defined and will run from the terminal.
Chapter 2  

Background

2.1  

State Machines

State machines are an abstract mechanism, or digital device, which traverses a set of states when certain criteria have been met for the current state [6]. The two main components of a state machine are the states and the transitions. States are points in the machine where the machine takes some action. Transitions are a set of conditions that are required to be met for a machine to transition from the current state to another. Importantly, the state machine can only be in one state position at a time. This means that the machine is only required to check the conditions of the state it is in and find what state to go to next. The actions of a state can be anything from changing a variable in software to actuating a relay through registers on a microcontroller, essentially any change in any parameter is an action [7]. These actions can be subdivided into three types listed and explained below:

- **Entry Actions**: An action that occurs once upon the entry of a state, e.g. turning on a motor.
- **Normal Actions**: An action that continues to occur while the state is active, e.g. flashing a light.
- **Exit Actions**: An action that occurs once upon exiting the state when a transition occurs, e.g. turning off a motor.

The transitions can be any logical operation or physical input that provides a true or a false, such as a microcontroller input. For state machines to run they require memory and the ability to perform logical operations. The logical operations are what determines the actions of a state and also calculates the conditions that need to be met to transition to the next state. For the machine to know what state to go to next it must also know where it is and remember parameters that can be used for the logic, so the machine must have memory. State machines
are a powerful tool used for developing the logic and code for controlling the operation of machinery and software application. Due to their simplicity and accuracy when performing sequential and logical operations, they are used in various fields. For example, they can be used to control traffic lights, programs in video games, operations in CPU controllers and transmission protocols [6].

State machines contain a combinatorial logic element and a memory element. This combinatorial logic element is a combination of two separate elements that complete logical calculations [6]. One of them is called the ‘Next State Decoder’ which decides what the next state of the machine is to be. The decoder determines which state to go to next by looking at several inputs to the system as well as checking to see if any other software or internal parameters have been met, such as a timer. The other, called the ‘Output Decoder’, is the set of logical steps that determine the type of outputs that will be performed by the current state, also known as actions [6].

There are two types of state machines, known as Mealy and Moore. They vary slightly in their implementation, specifically at the combinatorial logic element. A Mealy state machine performs actions on the transition between states. The Moore state machine performs actions within the state while the state is selected [6]. However, the design of a state machine does not have to strictly follow one type over the other. Instead, it can be a combination of both. State machines are useful in that they are easy to design, modify and visualise while also being very robust in their design. They also convey to the designer a clear and concise way on how the machine works especially when they are portrayed as a ‘state machine diagram’ [7]. A simple state machine diagram is shown in Figure 1. It describes the functionality of a ghost in the famous game Pacman. These diagrams make it easy to visualise the logic and conditions to be met throughout the operation of a project. They also make it easier to catch potential problems in the design. Depending on the tools used, such as different programming languages, a set of rules can be developed by a team to be then followed in the development
process of the state machine. This makes development easier for teams to quickly understand each individual part of the design since the rules and sequence of operations are clearly outlined [7]. Additionally, while designing state machines, it is simple to adjust, rearrange and add functionality to the design. This means that design and integration of a project can be done at the same time with much less burden on the designer when there are sudden changes in the requirements. This modification process is made easier because modifying one area or adding in new ones, if careful, generally doesn’t affect the rest of the state machine. For these reasons, state machines allow for a quicker “design to working prototype” period [7]. In general, it is a good thing to save time and money on a project, which state machines accomplish.

![Pacman State Machine Example](image)

To elaborate on the diagram, the behaviour of a ghost in the iconic game Pacman can be described and coded as a state machine. The actions of the ghost in the example are analogous to any action performed by a machine, Programmable logic controller or microcontroller. For instance, the ghost’s action to ‘Flee Pacman’ can be representative of turning on a relay from a
microcontroller to actuate some valve, piston, rotor or any other appropriate type of actuator.

The ghost in this game has 4 basic modes of behaviour, states. These are:

- Wander the Maze
- Chase Pacman
- Return to Base
- Flee Pacman

There are several transitions for this example which are easiest to show in the state machine diagram. The rectangles shown are states that contain within them the actions that are to be performed when the state is active. The arrows show the transition conditions that need to be met for the state of the machine to transition to a different state, according to the direction of the arrows. The action of a state may also be another state machine in themselves [7].

### 2.2 Linked lists

One of the features of the language is the ability to add new states, transition conditions and actions spontaneously. A typical array structure could not be used to achieve this feature.

Arrays are usually static data structures that contain a fixed number of elements [8]. The arrays parameters, such as element memory type and element amount, have to be assigned at the time of its creation. Elements cannot easily, or at all, be appended or removed. The element values within an array can still be changed or cleared, but the memory location and array size stay the same [8].

Linked lists are a straightforward and elegant solution to solving this limitation of storing data at any time as needed, a dynamic data structure, in what appears as a single structure [9] [8]. Depending on the linked list ‘creator’ the elements within the linked list will be of a fixed size or they can be as long as the user specifies, within computer architecture and memory limitations. To explain how this is possible: firstly the structure of the elements and of the
linked list structure needs must understood. Each element contains two essential items; the data and the reference, or link, to the next element [9] [10]. The data is the information that must be stored in the structure. The reference, tagged onto either the beginning of the element structure or at the end, contains the memory location of the next element within the linked list. There are several ways a linked list can be created. For example, when an element is added to the linked list the specified amount of memory is assigned at whatever point in memory the element can be placed. The address of this location is stored into the previous element’s reference area. This allows the linked list to read the first element’s data, then jump to the reference area and find out where in memory the other data point is stored along with reference to the next element [10]. The last element in the structure has some value in the reference link that indicates it is the last element in the linked list [9].

2.3 Compilers and Forth

Forth is an old postfix language that runs can be used in the SwiftX and SwiftForth environments [2] [11]. In most languages, the compiler is an extensive program in its self that translates the written code into machine code. Forth has a much smaller compiler compared to other languages of today [12]. Some of the unique features of Forth’s compiler is for that of the ability of words to compile themselves, like conditionals and loops [12]. It has much simpler approach to compiling compared to most other languages, but it allows for the compiler to be extended [12]. Extending the compiler can be done by creating defining words. These can be defined similarly to other Forth words. This makes for a very powerful language. For the purpose of this thesis, it makes it especially easy to make what looks like a new ‘language’.

In Forth there are two distinct operations of code, execution and compilation. Execution is the code that occurs at ‘run time’ where compile time refers to things that happen when a word is
compiled [12]. To separate and explain the difference it is easier to explain further what a defining word is. A defining word is a word that compiles another word, the compile time code, and gets that word to run a particular instruction, the run-time code. For example, the word CONSTANT in forth is a defining word. To declare a constant that has a value of 40 in forth is done with the following line of code.

40 CONSTANT EXAMPLE

The compiling behaviour of constant is to create a new constant type of name EXAMPLE which stores the number 40 into EXAMPLE’s parameter field, its memory location. When EXAMPLE is invoked, the run-time code of EXAMPLE is executed and places the value 40 onto the stack [12]. Forth allows people to make a new word that defines another, essentially extending the compiler. To do this, the word CREATE has to be invoked within the definition of a word which then makes it a defining word. This allows this new defining word to create an ‘object’, like a constant, that has the run-time code defined in the defining word after the word DOES>. The definition of CONSTANT is similar to the following pseudo-code:

: CONSTANT
    CREATE Compiler time code: “Store number on stack into the memory location here where the word is created”
    DOES> Run-time code: “Retrieve number from memory and leave on the stack”.

; 2.4 Lolly Machine

The Lolly machine was purchased and modified by Graeme Cole in 1997. Its primary purpose was for demonstration to the public during university open days. An additional purpose was for the machine to be used by students for learning and developing projects throughout their ICSE course. It is relevant to the course since it incorporates many of the aspects taught in ICSE. The two primary functions of the lolly machine are to sort lollies by colour and dispense a selected amount of specific coloured lollies to the user [13]. Figure 2 shows the lolly machine
This apparatus was used to test the developed Pseudo State Machine Language to see if it could work in a practical sense.

Figure 2 Lolly Machine Apparatus
Chapter 3  Simulator

To develop the language within the thesis timeframe, it was reasoned that a debugging tool would be worth developing. It was expected that the language would be complicated enough to warrant a simulator to help visualise what the state machine language was doing while developing it. The simulator would, hopefully, not only be helpful in the development of the language but also for students when creating state machines using the language. The simulator can even be used as a standalone program in its current state to test other code or for development into something completely different.

Since there was thought to be the utilization of an extensive linked list and multiple tasks, which are structures that run FORTH software multitasking feature which is explained further in Appendix A.2, during the development and testing of this project it was deemed necessary to create a simulator to complete the project within the timeframe. Memory manipulation and jumping is not something that has an output so nothing can be clearly seen. Often entering, writing or reading from the wrong memory location can cause a system crash with no explainable reason because the program doesn’t show an error report. One useful feature the simulator can perform is to place markers throughout the code that indicate a successful operation or run. If the program crashes and it hasn’t reached marker D out of A B C D then the problem is occurring between marker C and D.

Research went into Forth’s ability, like most languages, to instruct the Windows operating system to run some of its functions. These include creating new windows, which are called ‘apps’ in forth. These apps can display text, images and are even capable of running simple games. The goal is to create an app that can help in the development and for a user to visualise where their state machine is, what it has done or indicate if it is running at all.
3.1 Requirements

The final outcomes desired for the simulator are the following:

- Display visual information to the user;
- Be easy to adapt to the user’s code;
- Contain enough features that can be developed upon later to make a more extensive simulator;
- Be robust;
- Usable by multiple state machines.

The priority is to get any visual indicator possible on the screen, quickly. The emphasis on this part of the thesis was the speed in which this simulator could be built. The aesthetics of the simulator were a lower priority.

3.2 Method

The first step was to make sure that it was possible to display something on the computer’s main display in an easy to understand way, using Forth. The SwiftForth environment provides a set of examples, with source code, of different types of apps. One of them produced randomly sized and randomly coloured rectangles every second. Other examples included moving objects, like a ball, that would bounce off the edges of a window and a mouse click counter.

The most appealing example was the rectangle example. If these rectangles could be controlled so that they can portray to the user information visually, it would be a way of satisfying the visual information requirement. The idea was to look into the code that runs this example and modify it to generate rectangles at specific locations in the window when desired. These functions should be able to be called within the state actions of a state machine to represent some operation.
To modify the programs the window’s ‘system-defined messages’ needed to be somewhat understood. These messages are what Forth uses to provide information or instruct Windows applications to perform functions. For example, the word WM_PAINT acts like an event. If the application has something drawn on it with any of the drawing tools then the WM_PAINT function will run words designated to it. SwiftForth contains a help feature called ‘Win32 Programmer’s Reference’ which does provide some context and help on the system-defined messages in the programs. To modify the rectangle code required reading through and referring to the help feature to understand how particular sections worked.

After analysing the example code, a list of goals was created in order of apparent difficulty:

1. Get the rectangles to appear in the desired colour to indicate status.
2. Get the rectangles to appear in the desired location.
3. Get the rectangles to appear only when requested, via a word input.
4. Have the ability to clear rectangles.
5. Have the ability to move rectangles around.
6. Add text to the program for identifying what each rectangle represents.

This list did prove to be correct regarding the level of ascending difficulty of each task. This is because the later features tended to rely on the previous ones with additional functions.

### 3.3 Result

The simulator did take some time to work out due to new methods and functions used to control windows with forth. The words and methods used were very foreign and required a lot of investigation. In the end, all of the 6 goals listed above were accomplished. Each of these is discussed below.
3.3.1 Goals 1 2 and 3

An imported optional function called ‘RECTANGLE’ is used to create rectangles on an application screen. This function is within the Forth optional source files, explained in Appendix A.1. This function requires the coordinates of the upper left and bottom right of the rectangle relative to the top left of the application in pixels as shown in Figure 3.

Within the developed app there are 100 pre-defined rectangles for the users to use. They are stored within a large array as 4 reference points; X1, Y1, X2 and Y2 which can be seen in Table 1.

Table 1 Rectangle Array

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>X1</th>
<th>Y1</th>
<th>X2</th>
<th>Y2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>90</td>
<td>20</td>
<td>140</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>160</td>
<td>20</td>
<td>210</td>
<td>70</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>650</td>
<td>650</td>
<td>700</td>
<td>700</td>
</tr>
</tbody>
</table>
In order to create any of the rectangles with this function, several processes must be completed. A crucial part is to refer to the created application with its window handle called ‘HWND’. This HWND word grabs the window’s handle of the application it is used in. It is required by many functions to determine what window they are to perform their actions. The simulation applications’ HWND is saved on the creation of the application into a variable by the WM_CREATE system-defined message. This word is an event that runs allocated words on the creation of a new window. When the HWND is called inside of the words ran by WM_CREATE it returns the value of the newly created window handle instead of the SwiftForth application’s HWND.

Three words have been created to provide the user with the ability to change the state of the 100 rectangles easily. They only require the number of the rectangle they desire to change. These words are:

1. `<rectangle number> STATEON`: creates a green rectangle
2. `<rectangle number> STATEOFF`: creates a blue rectangle
3. `<rectangle number> STATEblue`: creates a blue rectangle

These words have to perform several steps in order to create or change the colour of a rectangle, outlined below. Process:

1) Create a drawing brush:

`<3 byte number> CREATESOLIDHBRUSH`

This creates a brush object that is used to colour an area on the application when used by drawing functions, such as the RECTANGLE function. The colour is specified with a 3-byte number with the structure shown in Table 2.
Table 2 Colour for Each Byte Used by CREATESOLIDBRUSH

<table>
<thead>
<tr>
<th>Byte</th>
<th>Colour (Value 0 – 255)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Lower)</td>
<td>Red</td>
</tr>
<tr>
<td>2 (Middle)</td>
<td>Green</td>
</tr>
<tr>
<td>3 (Final)</td>
<td>Blue</td>
</tr>
</tbody>
</table>

A value of 0 indicates that that colour is off and a maximum of 255 indicates that colour of the pixel being 100% on. So, a higher number causes the pixel of that colour on the screen to be brighter.

This word leaves the object reference on the stack, <object reference>.

2) Obtain the “Display device context”

This is required for building and selecting objects by the simulation app. It is obtained by using the HWND of the simulation app and the function ‘GETDC’.

<stdio_window> GETDC

3) Get the simulation application to select the created object

<stdio_window> <Display device context> SELECTOBJECT

This gets the simulation application to select the brush object which is then used later.

4) Get the area within the simulation app that is to be updated.

<stdio_window> <Storage area, address> GETCLIENTRECT

This obtains the coordinates within the specified client that can be drawn to. These are stored into storage area defined by an address which can be the PAD [14], a memory location of indefinite size at a location relative to the last dictionary entry, or a variable of the appropriate size. The values are the top left of the application which is (0,0) to the bottom right, relative to the top left.

5) Draw the rectangle on the applications screen
This is where the imported rectangle function is used. It instructs the
application to draw the rectangle the coordinates provided by the RECTALLOT
array using the created brush.

<display Device Context> <X1> <Y1> <X2> <Y2> RECTANGLE

6) The brush object is deleted and the simulation app is released.

These words satisfy the first 3 goals of the simulation application. The last three required more
steps with more manipulation to work. They are explained separately below.

3.3.2 Goal 4

This goal was achieved with the use of the word INVALIDATERECT which can be used to clear
an area on the application. It functions by obtaining the address that is used by
GETCLIENTRECT to select the region in the application that is to be updated. The area obtained
by the GETCLIENTRECT functions is stored into a variable where it is modified to cover a
rectangle specified before the INVALIDATERECT word is invoked. The user simply uses the
created word ClearRect to clear a specified rectangle as shown:

<rectangle number> ClearRect

3.3.3 Goal 5

With the ability to clear it was then possible to create some rudimentary motion. This is
accomplished with the following steps

1) Clear the rectangle with the above word

2) Update the coordinates in the RECTALLOT array

3) Redraw the rectangle
To avoid inputting the wrong coordinates and changing the shape of the rectangles drawn, a word MoveRect was created. This requires the value to move in pixels for the x and y plane.

\[ <X> <Y> <Rectangle Number> \text{MoveRect} \]

Positive values of X move the rectangle right, negative values move the rectangle left.

Positive values of Y move the rectangle down, negative values move the rectangle up.

### 3.3.4 Goal 6

The ability to draw text required a different function to be used. This is called ‘DrawText’ which required several parameters to be used.

\[ <\text{Display device context}> <\text{Address containing string}> <\text{Length of string}> <\text{Address of formatting dimensions}> <\text{Text drawing flags}> \text{DrawText} \]

Parameters:

- \(<\text{Address containing string}>\) and \(<\text{length of string}>\) are both obtained by placing the string to display in the ‘z’ text form. For example, \(z\) this is the text", which returns the two aforementioned parameters.

- \(<\text{Address of formatting dimensions}>\)

- \(<\text{Text drawing flags}>\) are a set of constants used by the drawtext word that can be OR’d together to instruct it how to draw the text such as text alignment and spacing.

Placing text at different locations was limited to the drawing flags, ‘such as top left’ or ‘center’ of the application. However, the ability to replace the rectangle coordinate from GETCLIENTRECT aided in achieving this goal. A similar process was taken as to the clearing function. The client rectangle area is re-specified to be at a point 10 pixels above the rectangle specified. The text is then written here which helps identify what the rectangle is representing.

*Figure 4* shows an example of a state machine with three states, state#, with variables being used for the transition requirements, indicated with the var# names. The values in the
variables were changed manually to simulate some microcontroller inputs. The rectangles on the bottom on the right are there to show the placement of the rectangles on the edge of the application.

![Figure 4 Developed Simulation](image-url)
Chapter 4  Language Development

The state machine language was designed with the intended functionality to run just one state machine. After the language was built and running a single state machine, it was then to be updated to handle multiple state machines in tasks. This meant that initially, all variables associated with the state machine were to be global variables and later changed to user variables, these are discussed in Chapter 4.2.1 and 4.2.2. While creating the state machine the simulation app was used. To do this the language was initially written in the SwiftForth environment. However, there were not many changes in the way the code worked, mainly around how the tasks were defined.

Requirements for simple state machine language:

1. Creates a State structure that contains the information to run Entry, Normal and Exit actions when called. It must also contain a reference to the transition requirements that must be met to leave that state;

2. Develop a way to simply check if a condition has been met for a particular or set of conditions;

3. Selectable initial state for the state machine to start at;

4. Have the option of containing a stop state where the state machine turns off;

5. Incorporate a diagnostic feature to indicate the status of the state machines;

6. Make the use of the state machines as simple as possible. The fewer instructions for the user to follow and setup parameters the better;

7. Non-microcontroller specific. No hardware specific functions are to be used for the operation of the language.
Advanced requirements:

1. Have timers that can be simply used with the state machines. This will allow for the system to have fancier functions like a light flashing at a specific frequency;

2. State machines will be running in separate tasks. This will allow for multiple state machines to be running concurrently;

3. Each state machine will have the ability to stop and start other state machines.

4.1 Method

Before any actual code could be written a basic idea of the structure was required. So the method to developing this state machine was to first begin writing quick pseudo code in a word document with notes on the operation. Sometimes, especially later on in this phase, the actual code was written for more clarity. This allowed for problems, solutions and new ideas to be found and worked upon during the design phase. This pseudo code was constantly changed and updated to include as many features in as possible. Once an underlying architecture was developed it was then properly written into the environment and tested. The testing outlined flawed logic and syntactical errors and sometimes more efficient methods were found. During the testing, the simulator created was helpful for visualising particular markers, like breakpoints in other languages, throughout the language by changing the states of rectangles. At first the most basic functionality was included, with the more complicated and unfamiliar features added later.

The following section discusses the final structure of the language and how component of the language works together. It also explains some of the problems encountered, explanations of unfamiliar concepts and reasoning for particular solutions.
4.2 Individual components

The language was broken up into the following components for developments.

- Variables
- Machines Status Constants
- Machine control
- Timers
- Defining states
- Transitions
- Machine Loop
- Machine Diagnostics
- Machine creations
- Machine start

While some of these components are used together, it was possible and easier to separate these components in the code. This allowed for easier navigation of the code which made development both simpler and quicker. It will also provide the same benefit to future students.

4.2.1 Variables

As discussed in the background section of State machine, for there to be proper functionality there needs to be the ability for the machine to at least know what position, i.e. state, it is in. This means that there will most definitely have to be at least some variables assigned to the state machine so that it can ‘remember’. The only way is to declare these at the beginning of the code before they are called in later defined Forth words. In order to facilitate the future development and legibility of the language, the variables are defined together at the top of the state machine language code with comments explaining their purpose and usage. Prior
planning for specific variables was not necessary for development since when the need for them arose they could simply be added with no hindrance. However, the variable type was important and they were defined as normal global variables for initial development. Global variables are readily available by any word or task by simply invoking the variable name alone [15]. Global variables were chosen in the initial development because the state machine language was only built to run one state machine at a time at first, for simplicity. For multiple state machines to run at the same time each would require its own set of variables. Two machines can not be using the same memory location to store, for example, the address of the current state. This would cause the state machines to wrongfully think they are in another state resulting in the wrong actions being performed. So, later the global variables were replaced by ‘user’ variables which allow for multiple state machines to run at the same time. The reason for adding user variables later in development was to avoid delay because the creation of these and use of them was unknown. The hope was that replacing the global variables with these would not affect the functioning of the language. This was deemed acceptable because multitasking was an additional goal of the language. If problems did arise due to their addition then the global variables version could still be used to fulfil the basic requirements. The thesis priority was to create a language that can support at least one state machine. The variables created are shown in Table 3, they all have a size of 2 bytes (1 cell).
Table 3 Variables and Their Descriptions

<table>
<thead>
<tr>
<th>User Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreviousState</td>
<td>Holds the address of the previous state</td>
</tr>
<tr>
<td>CurrentState</td>
<td>Holds the address of the current state that is being executed by the state machine</td>
</tr>
<tr>
<td>Machinetimeout</td>
<td>Holds the value of the amount of machine loops to be passed before the machine is instructed to timeout</td>
</tr>
<tr>
<td>Machinestateloops</td>
<td>Holds the value of the machine loops that have occurred since entering a new state. It gets reset when a new state is entered.</td>
</tr>
<tr>
<td>MachineStack</td>
<td>This holds the value of the stack depth. It is used in checking for errors when a normal, exit and entry action are performed as well as transition checks.</td>
</tr>
<tr>
<td>MachineWait</td>
<td>Holds the value for the time the machine is to wait per state loop of the machine.</td>
</tr>
<tr>
<td>MachineStatus</td>
<td>Holds an indicator value that is used to represent the status of the machine. The values of which are discussed in the ‘machine status’ section of the thesis.</td>
</tr>
<tr>
<td>MachineTimer</td>
<td>Holds the value of the machine timeout time in milliseconds. However, the size of this variable isn’t always large enough.</td>
</tr>
</tbody>
</table>

### 4.2.2 User Variables

The state machines need to have their own set of variables that perform the same function as the global variables perform for a single state machine. They are only defined once but exist and are unique to each task defined. That means if a user variable is defined then the variable
exists in both tasks A and task B's user memory. Windows programmers would see these as equivalent to what is called a “thread local storage” [16].

Unlike normal Forth variables, user variables exist in a user list located in a dedicated machine register area. There is a pre-defined set of variables and some are vital to the operation of the tasks in Forth as they provide information about the status of the machine, its stack pointer and more [16]. Adding the variables requires the use of the word +USER [17]. The word’s definition was not apparent and adding user variables took a lot of trial and error. A particular ‘error’ that was found was the ability for user variables to be allocated to the same memory location. This meant that some of the pre-existing user variable memory locations that are not required by the state machines can be “re-used” to save space. After further investigation and testing the proper method of allocating user variables at a particular section in the list was found. Testing was performed by writing to and reading from the variables from multiple tasks and checking to see if the operation was correctly functioning.

4.2.2.1 Result of User Variables

The introduction of user variable for multitasking was ultimately a success and was accomplished in the following way.

The +USER word requires the relative position from the beginning of the user list and the size to be allocated to the variable. This position is the number of bytes from the beginning so the first variable, which is called STATUS, is at position 0. The actual address of STATUS in the 68HC11 used for the Lolly machine is $2 776. After allocating the variable, the word leaves the position of the end of this variable on the stack. To avoid disturbing the operation of the task, the location of variables essential to the operation of the task were not used for the language’s variables. To do this, the user variables were allocated starting from cell 14 to cell 30 relative to the start of the user variable list. The end position of the last state machine variable is used to create the user memory portion of the state machines. This is done by storing the value
after the last user definition, 30, into a single constant called #STATEMACHINE_U. The pseudo
code that helps explain this is provided below:

\[
\begin{align*}
&\text{< relative position from beginning of list >} \\
&\text{< variable size > +user < variable name >} \\
&\text{... (repeat above line for all variables)} \\
&\text{EQU #STATEMACHINE_U}
\end{align*}
\]

This method of adding new user variables is different in SwiftForth compared to SwiftX and can
be seen in Appendix A.4.

4.2.3 Machine Diagnostics

The machine status is identified by a set of unique constants, shown in Table 4, that are stored
into the MACHINESTATUS user variable to help the user identify the operational status of the
machine. They are stored in different parts of the language under different circumstances.
Some are stored after action words by states while others are inside the language to indicate
errors.

There is no way to tell the user that a change in the machine status has occurred. If the
machine has stopped because of an error then the user has to recognise this and check
MACHINESTATUS manually. The reason for this is because tasks in forth are not able to access
terminal IO commands. Since the state machines are a task they also can not run these
commands. For reference, these terminal IO commands include displaying things on the
terminal lines, like an error message. To further help the user identify the operation of the
machine a word called MACHINESTATUS? was created. This word checks the MACHINESTATUS
variable and prints out a text in the format:

“Status Code - Status: description.”

For example, if the machine is running normally then the printed statement after running this
word is:
“1 - Machine Working: Normal operation.”

The full list of the MACHINETATUS? Messages can be viewed in the appendix A.3. This method of identifying the status of the machine and errors is similar to the low-level error handling throw codes in the Forth system [18].

The machine status has limitations in that it does not identify errors outside of this language’s scope meaning the messages are limited to misuse of the state machine language but not errors that can occur by the improper use of Forth itself. Some of these errors, such as overflowing memory location or entering infinite loops, can cause the Swift environment to crash and or the microcontroller to freeze. This requires a restarting the microcontroller which clears the RAM so no indication of what the error is can be shown.
Table 4 Constants and Their Descriptions

<table>
<thead>
<tr>
<th>Machine Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Neverstarted</td>
<td>0</td>
<td>This is stored when the state machine is instantiated by the user. It indicates the machine either failed to start up or simply hasn’t been started.</td>
</tr>
<tr>
<td>#Normal</td>
<td>1</td>
<td>Indicates that the machine is currently working normally.</td>
</tr>
<tr>
<td>#Endstate</td>
<td>2</td>
<td>Indicates that an end state has been reached within the state machine. End states are discussed in Chapter 4.2.8 Stopping the machine.</td>
</tr>
<tr>
<td>#Errorstate</td>
<td>3</td>
<td>Very similar to the use #endstate but is meant to be used to signify an incorrect operation that has been identified and incorporated into the state machine by the user. Explained further chapter 4.2.8 Stopping the machine.</td>
</tr>
<tr>
<td>#Halted</td>
<td>4</td>
<td>This constant indicates that the machine has been stopped either by a user in the terminal or by another state machine.</td>
</tr>
<tr>
<td>#EntryActionError</td>
<td>5</td>
<td>This value indicates that the stack has increased after the running the Entry Action of the state.</td>
</tr>
<tr>
<td>#NormalActionError</td>
<td>6</td>
<td>Same as above but after the Normal Action</td>
</tr>
<tr>
<td>#ExitActionError</td>
<td>7</td>
<td>Same as above but after the Exit Action</td>
</tr>
<tr>
<td>#Transitions</td>
<td>8</td>
<td>This indicates that a single flag has not been left on the stack after a transition condition has been checked.</td>
</tr>
<tr>
<td>#MachineTimeout</td>
<td>9</td>
<td>Indicates the machine has timed out because the machine has not transitioned within the timeout period.</td>
</tr>
</tbody>
</table>
4.2.3.1 Results Machine Diagnostics

This component proved to be a valuable and worthwhile addition to the language. The MachineStatus variable helped during the development of the language and also for using the language when developing the Lolly machine’s state machines, discussed in Chapter 6.

4.2.4 Defining States

The defining states component contains the defining word for creating states and also the words used for allocating actions to the states. Defining words that create the states have to be written in the interpreter dictionary. The interpreter is the list of words that are used during compiling of target words. Target words are the words that run on the target microcontroller, see Appendix A.5. States are one of two essential parts of the state machine. The code that is in the language allows for users to define a state and assign actions to them. The state does not need to do anything at run-time except to provide its address, much like a variable. In essence, the states are exactly like a variable but with more room that is to be used for a specific purpose. The word DefineState creates a state by allocating 4 cells of memory with an initial value of 0 in each cell. The value of 0 in each cell means that even if the state was assigned to a state machine, it wouldn’t affect anything. The language looks into these cells to find an execution token that performs some operation. The value of 0 is the same as executing an ‘NOOP’, a ‘no operation’ where nothing occurs [19]. Similarly, a transition link of zero indicates the end of the transition checks. The state’s structure is referred to as a State Definition Block (SDB) the structure can be seen in Figure 5.
The first memory location contains the address of the first Transition Data Node, these are explained in Chapter 4.2.5 Transitions. The second, third and fourth memory locations contain the execution tokens of the Entry, Normal and Exit actions of the state. Execution tokens are the reference to the words executable code. To improve the readability of the code the following words were made to jump to the corresponding memory location within the cell:

- **SkipToTransitionList**: does not do anything. It was purely created to help with the readability of the code to let the coder know what part of the state is being accessed. It is defined inside the Interpreter.
- **SkipToEntryAction**, **SkipToNormalAction** and **SkipToExitAction**: These are words that jump to the location of the state address by 1, 2 and 3 cells respectively.

These four words discussed in the list above are collectively referred to as ‘skiptoyyy’ words.

The last four words explained require the address of the state on the stack before being executed. All of the words in the list above are defined inside both the interpreter and Target. This is because they are also used in the definitions of other forth words that are not defining words, see Appendix A.5 for more details on the difference between the Interpreter and Target dictionaries.

To assign actions to the state the following words were created:

- **EntryActionIs**
- **NormalActionIs**
• And, ExitActionIs.

These three words are collectively referred to with the word ‘xxxActionIs’. Before running they require the address of the state the actions are to be assigned to on the stack. They then jump to the appropriate memory location first with the words mentioned above. The word “ ‘ ” (tick) within their definition grabs the execution token of the word that is after the xxxActionIs words and stores it in the appropriate cell of the SDB.

4.2.4.1 Requirements and limitations

Only one word can be assigned to each state action type of the state. If a user wants to run two different words for any action, then they need only make another word that contains both of those desired words in the definition. They would then assign the new word to the state action. No execution token of a word observed ever exceeded the cell size. If this occurs, then there would be an error and the solution would be to change the SDB size and adjust the words that rely on its structure, such as the skip words above. However, the language has worked on three out of three different microcontrollers, so it is not expected ever to be a problem.

4.2.5 Transitions

4.2.5.1 Creation and structure

Along with states, transitions are the other of the two most important parts of a State machine. This is the part of the language utilises the linking list method. It allows for multiple transitions to be created and then assigned to states easily and dynamically, which means the user can append new transitions throughout the design phase of their state machine and also
easily update and modify the transitions. This makes it much easier to fix, debug and modify the machine.

This part of the language links two states together with at least one transition requirement. The ‘next state’ is the state that the machine will jump to when there is a true flag left on the stack after the checking process. The ‘from state’ is the state that the transition is assigned to and running while the transitions are being checked. The linking between two states is accomplished by the Transition Data Node (TDN) the structure of which can be seen in Figure 6.

![Figure 6 Transition Data Node (TDN) Structure](image)

The transitions are sorted in a linked list for each individual state. It can have its own set, or the same or shared set of linked list transitions. The linked list is created inside the ConditionIs word. This word requires the address of the ‘from state’ and the ‘to state’ on the stack, in that order and is then followed by the word that runs the transition check.

```
<from state> <to state> ConditionIs <transition word>
```

When the TDN is added to the memory its address is stored into the transition link cell of the from state’s SDB. The value that was in the SDB’s transition link cell, which is initially 0, is stored into the address of the TDN’s transition link cell. This means the state can now refer to this TDN. Inside this TDN the execution token of the word after the conditionIs word is stored.
into the condition cell and the address of the ‘to state’ is stored into the ‘to state’ cell. Figure 7 shows what happens in the memory when a TDN is created.

The same process occurs when a second, third and nth transition condition is added. So the linked list of TDN’s can have the following structure outlined in Figure 8. For this diagram, the values inside the boxes are what is stored in the address. Arrows pointing to the top of a box indicate that that cell is at address N#. Dashed lines indicate the address pointers used in the TDN linked list structure for navigation.
Notice that the last transition to be added will have its address stored into the state’s transition link. This means the last transition that is defined is the first transition to be checked at the end of each machine loop after the normal action executes. This can have some effect on the operation of the state machine if some transitions are intended to be checked before others. It is much easier for the user to create a condition word that handles this rather than incorporating a ‘specified preferential treatment structure’ into the transition checking system part of the language. This would increase complexity and computation time of the transition check word. Since it is easier to be implemented by the user and most probably not a common requirement this feature was not implemented in the language.

Since ConditionIs is a defining word it has to be defined in the interpreter dictionary, after this ConditionIs word there is a set of words that are similar to the skiptoyyy words. They make the code easier to understand when writing and for future students. They all require the address of the TDN on the stack before running.

- SkipToNextTransition, doesn’t do anything. It is there to improve the readability of the code.
- SkipToCondition, adds one to the address to get to the condition cell
- SkipToState, adds two to the address to get to the Next State cell

4.2.5.2 Condition Checking

After the transition list is created for a state the ability to look through and check the conditions needed to be created. This occurs in the Transition? word, performed in the machine loop, which is discussed in Chapter 4.2.6 Machine Loop, after the normal action. This Transition? word contains a ‘begin while’ loop which runs through each of the TDNs in the linked lists assigned to the current state. By using the CurrentState user variable the word obtains the address of the first transition link. It then jumps to the condition word and runs it to receive a true or false which would cause the following operation seen in Table 5.
Table 5 Transition True or False Outcome

<table>
<thead>
<tr>
<th>Condition Result</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>The value stored in the next state cell is fetched. Then the loop is exited. This value is then used by the machine loop word transition to the new state.</td>
</tr>
<tr>
<td>FALSE</td>
<td>Nothing occurs and the loop is repeated.</td>
</tr>
</tbody>
</table>

The “begin while” loop allows the language to detect if the end of the transition link has been reached. The loop will only execute if the value from the Transition link cell is not zero. Since the last transition to be checked contains a zero in its linked list it will exit the loop. Note also, if a state had no transitions added, indicating an endstate, it has a value of 0 in its transition link cell by default so the loop is immediately exited. In these two cases, the value of 0 is left on the stack to be used by the machine loop to indicate no transition condition was true.

4.2.6 Machine loop

The Machine loop is the heart of the state machine language where multiple components come together to run the state machine. The machine loop is, as implied by the name, a loop that continues to run the state machine assigned. It is actually made up of two loops, a ‘begin until’ loop inside of a ‘begin again’ loop that performs the following tasks:

- Runs all actions of the states
  - Entry Action
  - Normal Action
  - Exit Action
- Machine timeout function
- Checks the Transition conditions of the states
• Error checking
  • Stack increases
  • Initialisation of the machine

The machine loop is what keeps the state machine running. Before the word jumps into the infinite loop an initialization of the machine occurs. At this point, the MachineStatus variable is set to 1 to indicate it has been started. Once it enters the loop, it should never leave unless the state machine is stopped and re-activated with a new initial state. The ‘begin again’ loop only runs once per state. It contains the call for the current state and performs the Entry action and the Exit action. In-between the Entry and Exit action is the ‘Begin Until’ loop which runs the normal action of the state and also checks for the Transition conditions. The placement of this loop here means that the machine will continue to run the normal action until a transition is met. At that point, the loop is exited and the exit action of the state is immediately run. The address of the next state is then placed used for the new state.

This function is quite involved some pseudo code can be seen in Figure 9.
Figure 9 Pseudo Code for Machine Loop

Further Description:

1) Before the machine starts it is initialised by setting the machinestatus variable to the
#normal value and the currentstate and machine loops user variables both to zero. The
reason the #normal value is stored and not some ‘starting up’ value is because if the
program is up to this point then things should be working correctly, i.e. normally.
There should be no errors from this point to the next set of words where the state
machine actually. The ‘currentstate’ is set to zero just to be safe. It does, almost
immediately, get updated to hold the address of the initial state left on the stack on
start up.
2) The new state is important because it stores the address of the new state in the CurrentState variable which is used throughout the language for operation on this state. Without it the state's address on the stack would have to be duplicated and manipulated for other words to refer to the state and to avoid incorrect operations.

3) , 5) and 9) perform the actions of that particular section of the state. Before they do this the task's stack depth is stored and compared to the stack depth after the execution of the actions has not changed. If it has then this could cause the task's stack to continue to increase which will cause it to write outside of its memory bounds. This causes a system crash that does not provide any error reports. For this reason, it took a while to find out why in some cases the system was crashing.

4) The Machinestateloops is reset to zero before entering the next loop so that the timeout function will work correctly.

6) This is where the machinewait value is used. It is taken and used by the MS word which contains the necessary words to instruct the task to relinquish control of the CPU while waiting for the time to elapse. Even if a value of 0 is entered into machine wait the state machine will still give up control but take it back as soon as it can and continue running.

7) This has to be placed after the wait to be correct for obvious reasons. If the value of the machinetimeout is greater than 0 then machinestateloops is incremented by one. Even though the loop has not finished, because the transitions still have to occur, It will stop if the time has exceeded regardless. It is stopped if the value in machinetimeout and machinestateloops are equal.

8) The transition condition link is checked. The value of the flag is 0 if false and the address of the next state if true. For this reason, if it is true the value has to be duplicated before getting checked by the until word to avoid it getting lost.
4.2.7 Starting the machine

Starting the machine is done with a word that only requires the initial state and the name of the state machine followed by the word ‘SM_GO’.

<Initial state> <state machine name> SM_GO

This word contains the ACTIVATE word which starts the task that runs the state machine. To pass the initial state value into the CurrentState user variable it first has to be stored into a normal global variable. It is then called after the ACTIVATE word and stored into the CurrentState user variable. This is because the word ACTIVATE starts up the task which no longer looks at the same parameter stack. Therefore, it does not see the initial states address left on the terminal stack.

It is desired that there would be as little ‘control’ on part of the user to get the state machines to work correctly. Forth does not automatically clear the contents of memory when variables are created. This is important when trying to convey the status of the machine to the user with MachineStatus. One of the parameters as mentioned above is the #neverstated value. This needs to be done before the ACTIVATE word which runs within the SM_GO word which in turn starts the machine. It would be desirable to automatically have the address of the created state machine available to define the areas at the time of creation. This can not be done because of how the system is designed to build and download the code, see Appendix A.6. The memory for the variables, in essence, does not exist until the code is downloaded onto the target system so there is no easy way to save the values. It is also because the defining word is in the interpreter dictionary and there is no link to the created object, the state machine, after the word’s use.

There is a way to do it but it was recommended by the supervisor not to progress down that path as it would be too much work for such little gain. So, to fix the problem of initial values the user must run an instantiation word before running the machine. This can be done by using the INSTANTIATEMACHINE word inside the DOIT word in the user file. Its purpose is to
set the machine up for starting and to store the status code that will tell the student that the
machine has been built but never started.

INSTANTIATE_MACHINE requires the name of the state machine on the stack and does the
following:

1) Builds the memory area for the machine
2) Stores the status code #neverstarted into the machinestatus variable
3) Defaults the machinewait value to 100 ms
4) Defaults the machine timeout function to disabled.

4.2.8 Stopping the machine

The state machines are stopped in the event of catching an error or can be stopped simply
because they have completed their task. Since the machines are within tasks they can be
controlled and stopped using the Forth words that control these. However, three different
words were created to help the users easily implement a stop into their state machines in a
variety of ways. Table 6 describes each of these words. The word NOD performs the actual
stopping of the machine. This puts the TASK to a sleep state where it does not perform any
action and waits to be restarted by the ACTIVATE word. The only way to restart the machine is
by starting the state machine using the same process as discussed previously by instantiation
and using the SM_GO word.
# Table 6 Stop Words and Their Descriptions

<table>
<thead>
<tr>
<th>Stopping Word</th>
<th>Description</th>
</tr>
</thead>
</table>
| **EndState**  | Used in an action of a state to stop the machine.  
  The #EndState value is stored into the MACHINESTATUS variable. |
| **ErrorState** | Used in an action of a state to indicate that a condition/s has been met that should not have occurred. To clarify, this does not mean an incorrect operation has happened as a result of the language. Rather, this word is used by the user to indicate that the machine encountered a condition that requires it to stop. For example, it can be used if a sensor should always be off but in a given state but is not. Perhaps to indicate a jam in the system.  
  The #ErrorState indicator value is stored into the MACHINESTATUS variable. |
| **HaltMachine** | This word halts a state machine that has its address left on the stack after being invoked. It can be used by a user in the terminal or be performed in the action of another state machine.  
  The #Halted indicator value is stored into the MACHINESTATUS variable. |
**4.2.8.1 Timeout feature**

Additional to these words is a machine timeout function built into the machines. This Timeout function can be activated if the user requires the machine to automatically turn itself off if it has not met any transition condition for the specified ‘timeout time’. This can be useful in situations when one would expect a state transition to occur within a certain time throughout all stages of the machine. For instance, if nothing is occurring it could indicate, again, a jam in some equipment used by the machine. This feature may not be something that is always required by the user but if it were it would require adding timers in every state. This would be tiresome to the user and prone to errors so it was thought to be worth implementing into the language.

By default, the machine timeout feature is disabled at the machines instantiation. It is enabled, and also re-disabled, with the use of the MACHINETIMEOUT_TIME word. This word requires a value for the timeout time in milliseconds within limitation that is dependent upon the value of the MachineWait time. To understand why this is, the operation of the timeout needs to be understood.

The timeout feature works by comparing the current machinestate loops variable with the MACHINETIMEOUT variable. To do this, the timeout time is converted into the number of loops the machine would need to execute to reach that time. Equation 1 outlines what the MACHINEWAIT_TIME word does. To clarify, the words in brackets are indicating the ‘units’ of the variables.

**Equation 1 Summary of MACHINEWAIT_TIME Operation**

\[
\frac{\text{Timeout Time (time)}}{\text{MachineWait (time/loop)}} = \frac{\text{MACHIENTIMEOUT (time)}}{\text{MACHINETIMEOUT (loops)}}
\]

The limitations from Machinewait is to do with size of the values that can be stored in MACHINETIMEOUT. All the variables are 2 bytes in size which means MACHINETIMEOUT can only hold and represent 65 535 loops of the machine. So, a small machinewait time of 1
millisecond would mean the timeout time could be a maximum of 65 535 milliseconds and a minimum of 1 millisecond. If the machinewait is 100 milliseconds then the timeout time can be a maximum of 6 553 500 milliseconds, around 107 minutes, and a minimum of 100 milliseconds. In the manual created for the students, they are instructed on how to implement this feature properly to avoid errors and also the limitations. This function, as well as the other timers created in the language, are software driven which means there timing is dependent on the software execution time which can vary.

4.2.8.2 Results

When these words stop the machine it no longer performs any operations but the memory locations within the state machine are still accessible. This means information can be obtained about the state of the machine. The user may also use normal Forth words to stop the state machines but they would have to manually place the appropriate signals into the user variables, especially if that information is necessary to the function of others. For example, a state machines transition condition could be relevant to another state machines status, from MACHINESTATUS.

The machine timeout function does have a slight problem that was not able to be fixed. Small state machinewait times can cause inaccuracy since the function only relies on the number of loops that have occurred. As a result, it doesn’t take into account the execution time of the state actions and transition checks of its own or other state machines. To test this feature, a state within the lolly machine, discussed later in Chapter 6, had its transition requirements changed so that they would never be true. This would then cause the state machine to timeout. Different timeout times were tested and the period it took to timeout was determined with a stopwatch while watching the machine status from an infinite loop. Also, the machine wait time was varied to find out the affects it could have on the function. The
results can be seen in Table 7. These results were gathered by starting the timeout feature at different times and calling the MachineStatus? Word within an infinite loop on the terminal.

### Table 7 Timing Results

<table>
<thead>
<tr>
<th>Machine wait (milliseconds)</th>
<th>Timeout Time (milliseconds)</th>
<th>Actual Time before timeout (seconds)</th>
<th>Inaccuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10 000</td>
<td>10.5</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>5 000</td>
<td>5.3</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>1 000</td>
<td>1.16</td>
<td>16</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
<td>0.69</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>10 000</td>
<td>10.7</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>5 000</td>
<td>5.32</td>
<td>6.4</td>
</tr>
<tr>
<td>10</td>
<td>1 000</td>
<td>1.33</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>0.7</td>
<td>40</td>
</tr>
<tr>
<td>1</td>
<td>10 000</td>
<td>41.5</td>
<td>315</td>
</tr>
<tr>
<td>1</td>
<td>5 000</td>
<td>20.94</td>
<td>318</td>
</tr>
<tr>
<td>1</td>
<td>1 000</td>
<td>4.46</td>
<td>346</td>
</tr>
<tr>
<td>1</td>
<td>500</td>
<td>2.5</td>
<td>400</td>
</tr>
</tbody>
</table>

#### 4.2.8.3 Discussion

Factoring for human error, the results have an uncertainty of +/- 0.2 seconds [20]. A non-quantifiable factor affected the results which was the ability to determine the starting point of the timeout function. Communication from the terminal to the microcontroller takes time which causes latency issues. Depending on the current operation of the microcontroller this can take varying times to execute the sent commands, which in this case was the activation of
the timeout feature and a constant polling of the machine’s status. This, as well as the reaction time, led to longer times recorded that may have actually been encountered by the state machine.

The results of a MachineWait of 10 and 100 milliseconds are within the degree of uncertainty from each other. These values are also very close to the actual timeout time, at least for the longer 10 and 5 second times. However, it is assumed that the execution time of the loops is about the same for different timeout times. The constant 0.3 to 0.7-second delay indicates that this is because of the factors described above. Smaller the values would have a larger inaccuracy because this delay relative to the timeout time is relatively larger. For example, if you were to assume a 0.1-second delay in recording the results for 10 and 1 second you get different inaccuracies; even though the system is performing the same. Table 8 shows an example to highlight this point.

**Table 8 Example for Error**

<table>
<thead>
<tr>
<th>Timeout Time (milliseconds)</th>
<th>Theoretical actual time (seconds)</th>
<th>Inaccuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 000</td>
<td>10.1</td>
<td>1</td>
</tr>
<tr>
<td>1 000</td>
<td>1.1</td>
<td>10</td>
</tr>
</tbody>
</table>

Since the results are reasonably close to the timeout times the method for this feature was kept. However, the most extreme value at 1 milliseconds resulted in an error of nearly a factor of 4. This is not a result of the same inaccuracy which comes from human error. This is most probably due to the Forth system architecture not being able to run the loop at such a small MachineWait interval. This indicates that the fastest the machine can run is at 4 milliseconds. To be safe, the students are instructed not to use a MachineWait less than 10 milliseconds when using this feature. They also, for the same reason, can not rely on the normal action executing at the MachineWait for values smaller than 10 milliseconds.
Chapter 5  Language Usage

The language makes it very easy for a user to create and run a state machine. The steps involved in making a simple two state state machine are shown below:

1. **CreateMachine** <Machine Name>
2.  <Machine Name> **InstantiateMachine**
3.  **DefineState** <state 1> **DefineState** <State 2>
4.  <State 1> **xxxActionIs** <Action 1>
5.  <State 2> **xxxActionIs** <Action 2>
6.  <State 1> <State 2> **ConditionIs** <Condition A>
7.  <State 2> <State 1> **ConditionIs** <Condition B>
8.  <State1> <Machine Name> **SM_GO**

The bolded words are some of the created language words.

Outline of each step:

1. Creates a name for referencing the state machine.
2. Creates the memory that will hold the machine and initializes some variables to be used for diagnostics such as the Machinestatus.
3. Creates the state data blocks.
4. Execution token of Action1 is stored into Entry, Normal or the Exit Action cell depending on the selected word. The words are:
   a. **EntryActionIs**
   b. **NormalActionIs**
   c. **ExitActionIs**
5. Same as above but for State2 and Action2.
6. Creates a transition from State1 to State2.
7. Creates a transition from State2 to State1.
8. Starts the machine.

After the last step, the state machine will begin running. This is an extremely easy and time-saving method to developing code for a project due to its simplicity.
Chapter 6  Testing the Language with the
Lolly Machine

The lolly machine was the most rigorous test of the state machine language. It is an apparatus that has several inputs and outputs to sort and dispense lollies. There is human input front panel with buttons that determine the actions of the actuators to perform the appropriate actions to dispense a selected colour and amount. This provides a chance to create a large and practical example to test the state machine language.

Part of the previous thesis student’s project was to write the lolly machine code in a state machine but due to time restriction, they were not able to complete this particular objective [13]. It provided the opportunity not only to test the language but to also complete one of the future work goals from the previous project. This was to code the lolly machine using the state machine method.

6.1  Design

The operation of the lolly machine has to be as robust as possible since it is a demonstration project. As stated before, the lolly machine was meant to be programmed using a state machine method which will now be less complicated with the state machine language.

Two state machines were to be created and run concurrently to test the multiple state machine feature. They were the sorting and dispensing state machines. The state machines were designed using a state machine diagram and have been documented as well as commented throughout the code. This is required to make it as easy as possible for other students to understand and if needed to modify or extend the lolly machines functionality.
6.2 Method

Before any work could be done on the coding the components of the lolly machine had to be identified and labelled to find out what input and output need to be controlled to design the state machine. The lolly machine and its components are indicated in Figure 10, Figure 11, Figure 12 and Figure 13. Table 9, Table 10, Table 11 and Table 12 describe the function of each of the components. This would allow for the creation of an easy to understand state machine diagram. It would also help to transfer the state machine diagram into the state machine language. Testing and debugging of the state machine was done by observing the operation of the lolly machine and noting where it would incorrectly operate. Since the code is separated into states with unique actions it was easier to identify where in the code the problem was occurring.
6.3 Lolly machine components

Figure 10 Lolly Machine with Identified Components
Figure 11 Lolly Machine with Components Named – Upper Section
Figure 12 Lolly Machine with Components Named - Lower Section
Figure 13 Lolly Machine Dispenser Actuator Location

Table 9 Lolly Machine Linear Actuators and Their Actions

<table>
<thead>
<tr>
<th>Actuator</th>
<th>Extended Action</th>
<th>Contracted Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collector</td>
<td>Grabs Lolly</td>
<td>Drops the captured lolly</td>
</tr>
<tr>
<td>Detector</td>
<td>1. Holds lolly for colour detection</td>
<td>1. Grabs lolly to move to sorting (L3 must be contracted)</td>
</tr>
<tr>
<td></td>
<td>2. Drops lolly for sorting (L3 must be contracted)</td>
<td>2. Allows lolly to drop to reject tube (L3 must be extended)</td>
</tr>
<tr>
<td>Sorter</td>
<td>Drops lolly to reject area</td>
<td>Drops lolly to sorting area</td>
</tr>
<tr>
<td>Green</td>
<td>Grabs green lolly</td>
<td>Drops lolly to dispenser</td>
</tr>
<tr>
<td>Red</td>
<td>Grabs red lolly</td>
<td>Drops lolly to dispenser</td>
</tr>
<tr>
<td>Blue</td>
<td>Grabs blue lolly</td>
<td>Drops lolly to dispenser</td>
</tr>
</tbody>
</table>
Yellow Dispenser

<table>
<thead>
<tr>
<th>Grabs yellow lolly</th>
<th>Drops lolly to dispenser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispenses lollies to the user.</td>
<td>Collects dispensed lollies (must be contracted when lollies are being dropped to dispenser)</td>
</tr>
</tbody>
</table>

**Table 10 Lolly Machine Rotary Actuators and Their Actions**

<table>
<thead>
<tr>
<th>Actuator</th>
<th>Activated Action</th>
<th>Deactivated Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>Directs Lolly to Green and Blue filter</td>
<td>Directs lolly to Red and Yellow filter</td>
</tr>
<tr>
<td>GreenBlue</td>
<td>Directs lolly to Green Bay</td>
<td>Directs lolly to Red Bay</td>
</tr>
<tr>
<td>RedYellow</td>
<td>Directs lolly to Blue Bay</td>
<td>Directs lolly to Yellow Bay</td>
</tr>
</tbody>
</table>

**Table 11 Lolly Machine Reject Air Supply Actions**

<table>
<thead>
<tr>
<th>Actuator</th>
<th>Activated Action</th>
<th>Deactivated Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reject Air Supply</td>
<td>Supplies air to reject tube to return lolly back to the mixed lolly storage</td>
<td>Supplies no air to reject tube</td>
</tr>
</tbody>
</table>

**Table 12 Lolly Machine Sensors and Indications**

<table>
<thead>
<tr>
<th>Sensor</th>
<th>‘Triggered’ value</th>
<th>‘Non-triggered’ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reject Colour Bay</td>
<td>A lolly is in the reject tube</td>
<td>No lolly in reject tube</td>
</tr>
<tr>
<td></td>
<td>A lolly is within the colour detector Bay area</td>
<td>Colour detector Bay clear</td>
</tr>
<tr>
<td>CS1</td>
<td>Value of colour</td>
<td>Nothing</td>
</tr>
<tr>
<td>CS2</td>
<td>Value of colour</td>
<td>nothing</td>
</tr>
</tbody>
</table>
6.4 Hardware testing

Before testing of the language could commence the Lolly Machines status needed to be identified. To do this the current state of operation was to be viewed and any problems occurring where to be noted. This was done by setting up the lolly machine and turning it on with the microcontroller that contained the current working version of the code in the ‘app.f’ file which enables the user to burn the code into the ROM [13]. This means the microcontroller did not need to be connected to the computer to start the machines process, this is done on start-up.

6.4.1 Problems identified

The lolly machine, for the most part, worked well however there were a few problems that were noted.

Problems:

1. Operational Keypad was not instructing the machine to do the intended operation
2. Incorrect display of the blue lolly’s 7 segment display on the front panel. The value shown was only updated on the even values. It is meant to display how many blue lollies have been selected.
3. Pneumatic actuators are not actuating reliably.

6.4.2 Fixes

Problem 1:

The incorrect operation of the keypad was a result of the array linking the keypads numbers not being assigned into memory correctly. The correct implementation of the keypad values was entered which fixed the problem.
Problem 2:

To eliminate several potential causes at once the values that could be displayed on the device were sent straight to the display device, bypassing the conditional logic in the code. Table 13 shows the results.

<table>
<thead>
<tr>
<th>Value Sent to seven segment display</th>
<th>Value displayed on seven segment display</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

This indicated that there wasn’t a logical problem occurring in the code running the machine but rather something wrong with the hardware. Leading from the microcontroller to the seven-segment display is a lot of electrical wiring and components each of which could have had issues. Due to the ease of access of components, the problem was debugged from the microcontroller towards the seven-segment display.

4 pins communicate to the seven-segment display to send a number in parallel. These four pins represent a 4-bit number which is the smallest number of bits required to send a number...
from 0 to 9, which is the only values the 7-segment device can display. It was suspected that
the pin representing the lowest bit was not being received. Since this bit is needed for the odd
values in the binary notation.
A value of %1111 was sent to the device to turn on all pins and the following components were
checked:

1. Pin connections to the microcontroller.
2. Connection to electrical isolation board.
3. Connections from isolation board to the 7-segment display.

The connection from the isolation board to the display device had zero volts, indicating a short.
This meant there was either a problem with the isolation board or the display device. Due to
the ease of access to the isolation board, it was investigated first but no problem was found.
The problem ended up being a short of the circuitry inside of the display device. This was
rectified and the device worked with no problems afterwards.

**Problem 3:**

The air supply tubes to the pneumatic actuators were checked and some were found to be
loose. They were re-attached and also the settings of the valve restrictors were adjusted to
allow more air to pass quicker into the actuators. This pushed the actuators enough to
overcome the static friction which sometimes caused the actuators to not move when
activated.

### 6.5 Sorting State machine

The purpose of the sorting state machine is to separate the colours of the lollies into the four
different colours red, blue, green and yellow while rejecting the fifth, orange. It needed to be
able to actuate the correct cylinders at the appropriate time to direct the lollies and also be
able to deal with any errors. Errors that could occur include:
A lolly not being collected at the storage
- Incorrect starting positions
- Lolly getting jammed in the collector
- Cylinders getting stuck
- Lollies getting jammed in and sliced in half
- Incorrect timing

The state machine diagram shown in *Figure 14* is the result of many iterations in order to find one that flowed well. This is a simplified version that does not show the exact actions and transitions that occur during operation since they are explained in the ‘states of the sorting state machine’ section.

![Figure 14 Sorting State Machine Diagram]
To make a more comprehensive state machine diagram the words for actuation the pneumatics and LEDs were created with unique names. The states were then separated into entry normal and exit action sections and the appropriate action were assigned to complete the task assigned to that state. The main concern of developing the state machine is to make sure that by the end of the process the actuators were all in the correct position for the next cycle of the machine. For example, if the detector and sorter slides were extended and contracted, respectively, the next lolly would fall straight to the reject area. Using the limit switches that are inbuilt into the cylinders as part of the transition requirements makes sure that the lolly will not go to the next step unless the system is ready.

All state transitions have a timer within them that were tuned to balance speed and reliability of the lolly machines flow. Too fast a timer could cause the state machine to skip a state since the sensors had not detected quick enough. Too long a timer made the observation of the lolly machine boring and ‘clunky’, which is bad for a demonstration project.

6.5.1 States of the Sorting State Machine

State 1: State one collects a lolly from the storage area by extending the collector actuator and then contracting after a timer expired.

State 2: has the most transition in the state machine but there are basically three types. These are:

1. The machine detects no lolly in the colour chamber
2. The machine detects a lolly and the colour is orange, to be rejected.
3. The machine detects a lolly and is a colour that is to be sorted into the appropriate holding bay.
   a. Purple
   b. Blue
c. Pink

d. Yellow

If no lolly is detected then it either got jammed on a component or there was no lolly fed into the collector. This occurs when there is an oversaturation of lollies in the storage area. An oversaturation of lollies in the storage area could cause gridlock, if this occurs a disturbance in the area needs to occur to allow a lolly to be placed in the collector. Or it happens when there are little to no lollies left in the storage chamber. The state machine will just continuously go back and forth between state 1 and 2 until a lolly is detected. The only action taken by state 2 is to begin a timer that just gives the capacitor and colour sensors enough time to detect the lolly after it bounces around for a short while in the detector bay.

**State 3** the detection of an orange lolly will jump the state machine to state 3 where it will contract the detection cylinder and extend the sorting cylinder. This drops the lolly into the reject chute and flashes the lights different shades of orange. The state machine will continue in this state until the lolly is detected on the reject sensor. It is common for the lolly to get stuck but there is a hole in the front panel that provides easy access to nudge the lolly onto the sensor and in the correct position to be blown it back to the storage area.

**State 4** this state turns the reject air supply on which shoots the lolly up into the storage area. It also starts a timer which when it expires the lolly is assumed to have successfully landed in the storage area and the state machines transitions back to state 1 after turning the air supply off. However, this timer gets reset until there is no lolly detected on the sensor. This gives the operator the chance to dislodge a jam in the reject chute if is not cleared.

**State 5, 6, 7 and 8:** these states all have very similar actions so they can be described together. The only difference between them is what rotating actuator is actuated to sort the lolly. The following arrangement shown in Table 14 determined what way the lolly would fall.
All of these states contract the detector at the beginning to collect the lolly from the detector bay. The normal action of the states is to flash different shades of the colour that was detected. All these states exit to state 9 only when a timer has expired, the position of the necessary actuators are in the correct position and there are no lollies in the detection chamber. If a lolly is still detected in the chamber it means it has not fallen and there is a jam. This is a rare occurrence and the machine will not continue until the jam is fixed.

**State 9:** This extends the detection slide and allows the lolly to fall through the sorter actuator into the sorting chamber where the lolly should hit the top actuator and bounce to left or right to one of the other actuators which direct the lollies left or right to the appropriate bay. There is only a timer to get to state 10.

**State 10:** this state does nothing but waits before progressing to state 1 to begin the whole cycle again. The reason for waiting is to allow people to watch the lolly dropping down without any other actuation occurring.
6.6 Dispensing state machine

This state machine can run at the same time as the as the sorting machine but later this feature was disabled. The reason for this was the supervisor wanted the machine to either be sorting or dispensing. This idea was so the user of the machine may get distracted by the sorting while the dispensing operation is occurring. It also had benefits to the operation of the machine because when multiple pneumatic actuators are activated it can cause some to not move due to lack of air pressure. The following states were made for the dispensing state machine.

6.6.1 States of the Dispensing State Machine

*Figure 15* shows the Dispensing state machine diagram. Note, the transitions are not included in this diagram to avoid clutter. A description of each state, and the transitions conditions, is after the diagram.

![Dispensing State Machine Diagram](image)

*Figure 15 Dispensing State Machine Diagram*
Buttons? This state has multiple transitions that depend on the button on the front panel that is pressed. It does not perform an action it is merely a state that is used to determine what the next action will be depending on the buttons pressed.

Add (COLOUR) where COLOUR is Green, Red, Blue or Yellow: These states are entered from the BUTTONS? State and increment a variable #COLOUR and the display for their colour. These states also wait for no button being pressed to transition back to the BUTTONS? States. This avoids the problem of one button press being counted as multiple presses and also eliminates the need of a timer.

Dispense-selector: determines which colour needs to be dispensed into the cup

Dispense (COLOUR), where COLOUR is Green, Red, Blue or Yellow. This collects the colour and increments a variable #Lollies-in-cup which is used to identify when the lolly collector cup has 5 lollies in it. If it does then the machine will dispense those lollies before continuing. These states will be activated for a value chosen via the front panel.

Dispense Lollies: this state dispenses the lollies in the cup when there are five inside and then returns back to the dispense-selector

Dispense Lollies Final: this state occurs when there are 5 or fewer lollies remaining to be dispensed into the cup. After dispensing the final group of lollies it then waits for 2 seconds before returning to the Buttons? state.

Cancel: This state occurs when the Cancel button on the front panel is pressed. It resets the values of the variables used for counting and dispensing the lollies and resets all the actuators to the starting positions before jumping into the BUTTONS?
6.7 Overall results

Each component of the language was tested in the state machines implemented on the lolly machine. All of the components and features incorporated into the language worked accordingly. The components tested in the language and the results of the testing are shown in Table 15.

<table>
<thead>
<tr>
<th>Component</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>State actions</td>
<td>Working</td>
</tr>
<tr>
<td>State Transitions</td>
<td>Working</td>
</tr>
<tr>
<td>Timers</td>
<td>Working</td>
</tr>
<tr>
<td>Timeout</td>
<td>Working</td>
</tr>
<tr>
<td>Multiple state machines</td>
<td>Working</td>
</tr>
<tr>
<td>Stopping the machine</td>
<td>Working</td>
</tr>
</tbody>
</table>

These state machines have been designed so that they automatically deal with any jamming issues much to the same extent that the previous program did. However, there is no separate error tasking loop which was a separate task looping to check for any errors [13]. The state machine language allowed for all components to be easily melded into one state machine making for a more efficient and less complicated system.

The readability of the code is also greatly improved because of how easy it is to navigate the code. For example, the code for state 3 in the lolly machine is shown below.

Actions:

S3 ENTRYACTIONIS S3-ENTRY
S3 NORMACTIONIS S3-NORMAL

S3 EXITACTIONIS S3-EXIT

Transition:

S3 S4 CONDITIONIS S3S4-TRANS

It is very clear to see that the entry action for state three is the word ‘S3-ENTRY’. Also, the transitions condition assigned can easily be seen in the ‘S3S4-TRANS’ word. The language separates the code into clear and concise segments that make for easy navigation. This is opposite to the previous code which had multiple actions happening within complicated loops and conditionals [13].
Chapter 7 Arduino and Dragon Board Test

The purpose of this objective is to prove that the language can work on multiple microcontrollers. In addition to this, it was also intended to investigate and provide feedback on the possibility of the Arduino Uno replacing or being an addition to the equipment in the ICSE units. The Arduino Uno has been under consideration by the supervisor because it is cheaper and has more components that can be easily attached to extend its capabilities. It will also provide the students with more experience with different microcontroller architectures.

The Dragon board contains a 68HC12 which is very similar to the microcontroller on the Lolly machine, a 68HC11 [21]. There was almost no difference in the procedure to downloading and using the code on the Dragon board. Before the Arduino could be used, unlike the Dragon board, the SwiftX environment had to be installed. In order to achieve this goal, the SwiftX manuals were read that had instruction for downloading the environment. The procedure to downloading code onto the Arduino Uno is also radically different.

7.1 Results

There was a lack of documentation on the subject and methods provided in the manuals did not work. The drivers that were required for the Uno to run, provided in the Forth files, did not work on the university computers. There were some drivers that were automatically installed upon plugging in the Uno. These however still did not allow for the Uno to be flashed in the way the manual specified. So again, through some research and help from the university engineering technician, a program was found called ‘Atmel Studio 7’. This allowed for the burning of the SwiftX environment through the In-System Programmer the ARVISP mkII. Figure 16 below shows the setup for burning new code onto the Arduino Uno. Additionally, new drivers were found that worked with the computer.
The only limitation with the Arduino is that it does not allow for the interactive programming features that come with SwiftX. This means words cannot be created in the terminal of the SwiftX environment and downloaded onto the target. Every time a new word is created the entire kernel has to be re-burnt. This was found to be an unacceptable process for learning and developing the Forth language. Therefore, the Arduino Uno is not a recommended replacement to any of the equipment but may be suitable to be added in. This would provide students with more experience and will teach them the different methods that microcontrollers use in the operation [22].

![Figure 16 Equipment Set-up for Re-Burning the Arduino Uno](image)
Chapter 8  Conclusion

The main goal of this thesis was to create a state machine pseudo language that was capable of creating and running state machines for students. For testing, the lolly machine apparatus at Murdoch university was also updated to complete one of its future works goals, which was to code the machine using a state machine method. To help with the debugging processes a simulator application was created that is capable of visually indicating to the user the status of a state machine. Further testing of the language was done on a second type of microcontroller. This was done to not only test the language, but to also investigate the possibility of the microcontroller replacing or being an addition to the ICSE course. These goals have all been achieved within this thesis to a standard higher than expected. All of the requirements outlined in each objective were met. There are some potential future improvements and works that could be done to extend this project.

8.1  Future Works

8.1.1  Update the Simulator

The simulation application has a further work that can be done to it. It can be extended to include functions such as down boxes, communication between other apps, moving images and more. This would make the simulation more visually stimulating and appealing to the students. It can also be extended so that it provides a level of interactivity with the state machine. For example, it would be a good feature if the users are able to click on the rectangles to change their state. This would mean they can activate particular sections of the state machine just by clicking. This would provide a more interactive and intuitive simulator. At the moment the user must type in all instructions within the SwiftForth terminal.
There is example code provided that triggers an event click. It is assumed that, among the functions, there would be an ability to determine the location of that click. Since all rectangles have their locations known, this could be used to trigger a change in a rectangle of that area. One of the most desired future implementations to the simulator would be to communicate with the SwiftX environment running a microcontroller. This would require cross application communication, that may or may not be possible in Forth. Even if there is no direct way of communicating with other applications there may be an indirect method. Both SwiftX and SwiftForth are capable of writing to text files which can be accessed by both to pass information to each other. Whatever method chosen the ultimate goal would be to represent on the simulator the various outputs and inputs of the microcontrollers in real time.

8.1.2 Further Test the Language

One opportunity to further test the language would be to create a state machine with more in-depth workings on the registers and hardware such as hardware timers, interrupts, PWM, analogue to digital conversions and more. This could be possible to test on the newly acquired “playful Puppy Robot kit” from DAGU Electronics, aka ‘robot puppy’ [23]. This is a small robot that uses an ATmega8A microcontroller to control servo motors and infra-red-light sensing for movement and tracking [23]. Developing a state machine to code this robot would not only prove challenging in its own right but would also provide even more comprehensive testing of the language to outline any flaws.
References


Appendix A

A.1 Rectangle Function

The rectangle function is imported from the Forth system files. This function will use a created brush object to draw a rectangle on the screen depending on the parameters inputted.

A.2 Forth’s Multitasking System

FORTH's multitasking system – Forth has a round robin software driven multitasking structure. This is different to multithreading which requires specific hardware to run code in parallel. Forth runs one task at a time which runs code allocated to it until it relinquishes control of the CPU. This process continues for each task in the system in a loop. Each task contains its own memory area which includes a parameter stack, register stack and user memory.

A.3 MACHINESTATUS? Results

Status code - status: description:

0 - Machine Not Started: Machine was never started"

1 - Machine Working: Normal operation"

2 - Machine Halted: A user specified ENDSTATE has been reached." 

3 - Machine Halted: A user specified ERRORSTATE has been reached." 


4 - Machine Halted: Instructed by user or other machine to halt."

5 - Machine Halted: Stack changed after running Entry Action."

6 - Machine Halted: Stack changed after running Normal Action." 

7 - Machine Halted: Stack changed after running Exit Action." 

8 - Machine Halted: A single flag was not left on the stack after transition check"

9 - Machine Halted: The machine timeout period has been exceeded"

10 - Machine Halted: The MachineLoop word has been exited. Machine halted to avoid system crash"

A.4 Allocation a new User variable

#USER <variable size> +USER <variable name> (gets the size of the user list and allocates the variable size to the variable called after +USER, the new size of the user list is left on the stack)

TO #USER (stores the size after allocating the variable #user which is used for the next variable)

A.5 Interpreter and Target

The Interpreter dictionary is used to define words that are used to compile the words on the target. The Target is the words that are actually turned into machine code and run on the Target.
A.6  Downloading Code onto the Target

The code that is developed in the Forth files is downloaded once it has all been interpreted. That means the code to make things like variables has been converted to machine code and made ready to be implemented on the target. This means that the memory of things like tasks user areas and their variables don’t exist until the target system runs the code which is after the debug.