

## State Diagrams: a New Visual Language For PLC's

# STATE DIAGRAMS: A NEW VISUAL LANGUAGE FOR PROGRAMMABLE LOGIC CONTROLLERS

By  
FANFAN HUANG, B.ENG

A Thesis  
Submitted to the School of Graduate Studies  
in partial fulfilment of the requirements for the degree of

M.A.Sc  
Department of Computing and Software  
McMaster University

© Copyright by FanFan Huang, April 26, 2010

MASTER OF APPLIED SCIENCE(2009)

McMaster University  
Hamilton, Ontario

TITLE: State Diagrams: A New Visual Language For Programmable Logic Controllers

AUTHOR: FanFan Huang, B.Eng(McMaster University)

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: viii, 82

Ladder Logic has been the dominant defacto method for programming programmable logic controllers for over 30 years. Primarily designed as a replacement for electronic relay boxes Ladder Logic uses the analogies of circuits and wires. With the changes in education and training Ladder Logic has not kept up with the times. Ladder Logic is difficult to understand for a software background trained operators. In addition with increasingly difficult control logic strategies, Ladder Logic has become cumbersome and difficult to use.

This thesis seeks to examine a different approach to visual programming that is more suitable to modern software trained individuals. A visual programming language will be established based on finite state machines. We then define both the syntax and semantics of our visual language, demonstrate the correctness of operation and execution. This thesis also defines a reference hardware platform and shows our graphical tool used to construct programs in this new language.

The major contributions of this thesis include the development of a prototype programming language, a graphical integrated development environment tool, and a prototype hardware environment.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Abbreviations</b>	<b>1</b>
<b>1 Preface</b>	<b>2</b>
1.1 Structure of this Thesis . . . . .	2
<b>2 Overview of Existing Technology</b>	<b>4</b>
2.1 PLC Hardware Controller Implementations . . . . .	4
<b>3 Introduction to Ladder Logic</b>	<b>12</b>
3.1 Background of Ladder Logic . . . . .	12
<b>4 Software</b>	<b>19</b>
4.1 Goals . . . . .	19
4.2 Overview of State Machines . . . . .	22
4.3 Logic Control Chart . . . . .	26
4.4 Syntax of Logic Control Chart . . . . .	29
4.5 Implementation . . . . .	32
4.5.1 System Overview . . . . .	32
4.5.2 GUI and IDE . . . . .	33
4.5.2.1 Introduction . . . . .	33
4.5.2.2 Using The GUI: Parts of the GUI . . . . .	33
4.5.3 Data Flow . . . . .	40
4.5.4 Generated Code Structure . . . . .	40
4.5.5 Compiled Structure . . . . .	41
4.5.5.1 Start Block . . . . .	41
4.5.5.2 Delay Block . . . . .	42
4.5.5.3 Input Block . . . . .	43
4.5.5.4 Output Block . . . . .	43
4.5.5.5 Store Block . . . . .	44

4.6	Correctness . . . . .	46
4.6.1	Atoms . . . . .	48
4.6.1.1	Start Block . . . . .	48
4.6.1.2	Delay Block . . . . .	48
4.6.1.3	Output Block . . . . .	49
4.6.1.4	Input Block . . . . .	50
4.6.1.5	Store Block . . . . .	50
4.6.2	Constructors and Transitions . . . . .	51
4.6.3	Correctness of Translation . . . . .	56
4.6.3.1	Table Representation . . . . .	56
4.6.3.2	Start Diagram Execution . . . . .	58
4.6.3.3	Delay Diagram Execution . . . . .	58
4.6.3.4	Output Diagram Execution . . . . .	59
4.6.3.5	Input Diagram Execution . . . . .	61
4.6.3.6	Store Diagram Execution . . . . .	63
<b>5</b>	<b>Hardware</b> . . . . .	<b>68</b>
5.1	Platform . . . . .	68
5.2	Hardware Framework . . . . .	70
5.2.1	Purpose . . . . .	70
5.2.2	Hardware Framework Overview . . . . .	70
5.2.3	Sub-Component Implementations . . . . .	71
5.2.4	Hardware Specific Definitions . . . . .	72
5.2.5	Hardware Specific Implementations . . . . .	72
<b>6</b>	<b>Summary</b> . . . . .	<b>75</b>
6.1	Conclusions . . . . .	75
6.1.1	Conclusions . . . . .	75
6.1.2	Summary of Contributions . . . . .	76
6.1.2.1	Ladder Logic Analysis . . . . .	76
6.1.2.2	Development of <i>Logic Control Chart</i> Language . . . . .	76
6.1.2.3	Development of A Prototype Tool and Hardware Environment . . . . .	76
6.1.2.4	Development of A Hardware Platform . . . . .	77
6.2	Future Research . . . . .	78
<b>A</b>	<b>Appendix</b> . . . . .	<b>82</b>
	Stepper Motor Examples . . . . .	82

# List of Tables

3.1	Semantics for Fig 3.2 . . . . .	14
3.2	Semantics for Fig 3.3 . . . . .	14
3.3	Semantics for Fig 3.4 and Fig 3.5 . . . . .	15
3.4	Semantics for Fig 3.4 and Fig 3.5 . . . . .	16
4.1	Start Diagram shown in Figure 4.23 . . . . .	57
4.2	Start code execution for compiled code from Figure 4.23 . . . . .	57
4.3	Start code execution for compiled code from Figure 4.23 extended . .	57
4.4	Start code execution combined table. For Figure 4.23 . . . . .	58
4.5	Delay code execution combined table. For Figure 4.24 . . . . .	59
4.6	Output code execution combined table. For Figure 4.25 . . . . .	61
4.7	Input code execution combined table. For Figure 4.26 . . . . .	62
4.8	Store code execution combined table. For Figure 4.27 . . . . .	65

# List of Figures

2.1	Mitsubishi PLC All In One Unit [6]	5
2.2	3D Diagram of A Modular PLC [9, 10]	5
2.3	PLC Execution Loop	6
2.4	3D Diagram of A Modular PLC With One Module Being Inserted [11, 12]	8
3.1	Togglable Light in Ladder Logic	12
3.2	Basic Ladder Logic Diagram	13
3.3	Simple AND Logic Diagram	14
3.4	Branching Rungs	15
3.5	Branching Rungs (Alternative)	15
3.6	State diagram conversion for Table 3.2	16
3.7	State diagram conversion for Table 3.3	17
3.8	State diagram conversion for Table 3.4	17
3.9	Latched Ladder Logic circuit.	18
4.1	Simple Toggle Light	23
4.2	Moore (left) and Mealy (right) State Machines	23
4.3	UML State Diagram of a Toggle Light	24
4.4	UML2 State Machine Diagram Syntax[1]	25
4.5	Example of Basic Transitions	27
4.6	Incorrect Transitions	27
4.7	Example Of A Store Block As Implemented In PLCEdit	30
4.8	5 Tap Fir Filtering	31
4.9	System Overview	32
4.10	The Drawing Canvas	34
4.11	The Tools Palette	34
4.12	Compile Preview Window	37
4.13	Simulator Window Starting Configuration	38
4.14	Simulator Window Program Running	39
4.15	Code Transformations	46



---

4.16	Code Transition Structure . . . . .	47
4.17	Start Atom . . . . .	48
4.18	Delay Atom . . . . .	48
4.19	Output Atom . . . . .	49
4.20	Input Atom . . . . .	50
4.21	Store Atom . . . . .	50
4.22	Transitions . . . . .	53
4.23	Singular Start Block . . . . .	56
4.24	Delay Block . . . . .	58
4.25	Output Block . . . . .	60
4.26	Input Block . . . . .	61
4.27	Store Block Example . . . . .	63
5.1	Hardware Framework Components . . . . .	70
A.1	Running a Stepper Motor with <i>Logic Control Chart</i> . . . . .	83
A.2	Running a Stepper Motor with <i>Ladder Logic</i> part 1 of 3 . . . . .	84
A.3	Running a Stepper Motor with <i>Ladder Logic</i> part 2 of 3 . . . . .	85
A.4	Running a Stepper Motor with <i>Ladder Logic</i> part 3 of 3 . . . . .	86

# Abbreviations

<b>BLUID</b>	Block Unique Identifier, 41
<b>CPU</b>	Central Processing Unit, 19
<b>GND</b>	Ground (or negative in electronics), 12
<b>GUI</b>	Graphical User Interface, 32
<b>IDE</b>	Integrated Development Environment, 20
<b>IL</b>	Intermediate Language, 40
<b>LCC</b>	Logic Control Chart, 2
<b>PLC</b>	Programmable Logic Controller, 2, 19, 68
<b>TTL</b>	Transistor-transistor logic, 10
<b>VCC</b>	Voltage at the Common Collector (or positive voltage in electronics), 12

# Chapter 1

## Preface

### 1.1 Structure of this Thesis

Chapter 2: *Overview of Existing Technology*. We will begin by introducing the reader to existing PLC implementations. We will go into the history behind programmable logic controllers and their usages. We will also give the user an idea of what kinds of modules manufacturers have created in the industry over time.

Chapter 3: *Introduction To Ladder Logic*. In this chapter we will expose the reader to the existing language (Ladder Logic) that is currently in use in the industry. The reader will be exposed to the syntax and semantics of this language. In addition the user will also obtain some background insight as to how the language came into looking as it does today.

Chapter 4: *Software*. The software section covers all of the software implementation information of the proposed language Logic Control Chart. We begin by defining the goals in constructing LCC. Next the language is then defined for LCC. In the final two sections we go into implementation details of the software package as well as show correctness of the diagram to code translation.

---

Chapter 5: *Hardware*. In this section we go into detail about the hardware reference platform. We introduce the hardware framework which allows multiple microcontrollers to be utilized. We finish by showing parts of the hardware driver code that must be implemented should the reader wish to implement their own hardware board.

Chapter 6: *Summary*. This chapter present all the conclusions of the work. In addition we also recommend future directions this thesis can go if continued.

Appendix: Contains examples and diagrams in Ladder Logic.

# Chapter 2

## Overview of Existing Technology

### 2.1 PLC Hardware Controller Implementations

Programmable Logic Controllers (PLC) have been around for over 30 years and as a result there have been many iterations and designs. The original Programmable Logic Controllers were a quick way for automotive manufacturers to replace traditional relays with digital control. These relays were hooked up to power rails and inputs, and allowed for basic mechanical logic [4]. Due to their mechanical nature relays wore out over time causing the logic they were performing to fail. In addition, because hundreds to thousands could be used in a cabinet, it was also difficult to isolate the worn out part. Relays also proved to be inflexible when a small change was required to be added to the program, the entire plant was required to be taken offline in order to make the change. Halting large production plants is often extremely costly and thus eventually the relays were migrated out in favour of microcontrollers that can be reprogrammed on the fly. To this day modern PLC's still use graphical analogies of circuits and relays in order to construct their programmable logic. This visual language is now referred to as ladder logic due to the the finished program having a similar visual structure to a ladder.

Mitsubishi Automation see Figure (2.1), Siemens, and Omron are a few of the big producers of industry standard PLC's although the shape and form factor dif-

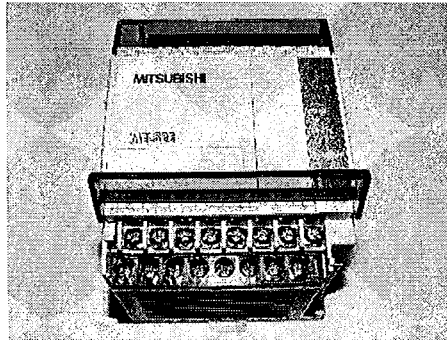


Figure 2.1: Mitsubishi PLC All In One Unit [6]

fer between manufacturers PLC's always consist of three distinct parts: The input module, the main controller unit and the output module (see Figure 2.2). This separation exists due to varying requirements for analog inputs and different output capacity requirements in order to drive heavy machinery. I/O modules may consist of thermal-sensors, ambient light sensors, resistive sensors, or a direct connection to the external circuitry. The output module may also be composed of both analog and digital output pins.

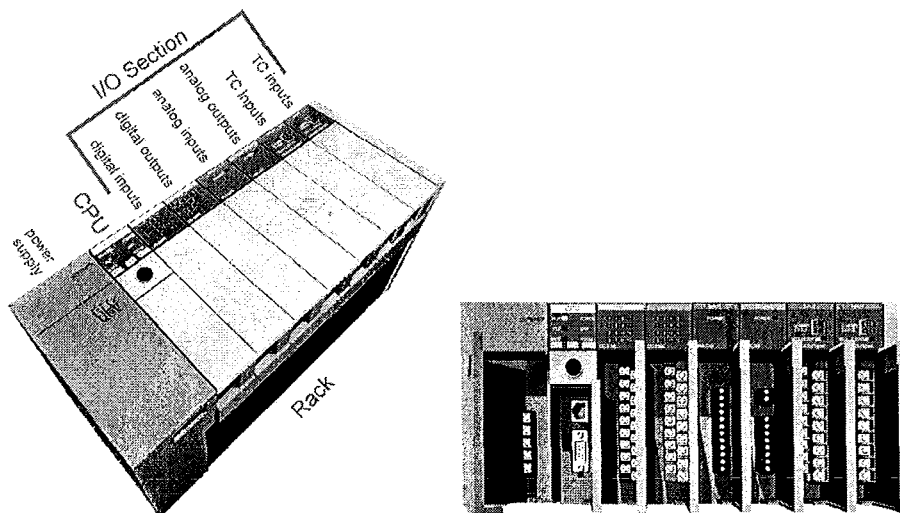


Figure 2.2: 3D Diagram of A Modular PLC [9, 10]

Executions of programs on a PLC are always done in a loop. An iteration of execution is referred to as a scan. Each scan is further broken up into 4 phases: Self-Test, Input scan, Logic solve / scan, and Output scan. Figure 2.3 shows the order in which each of these steps are executed. The jobs performed in each step is described in more detail below:

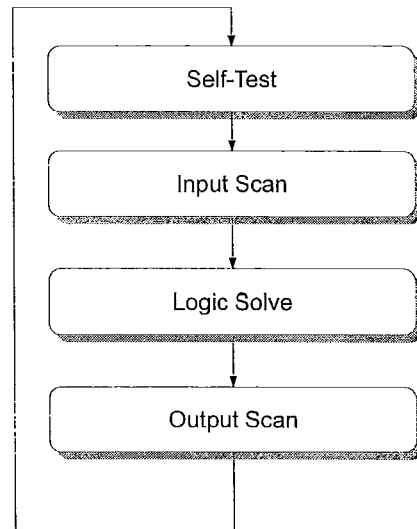


Figure 2.3: PLC Execution Loop

- **Self-Test:** All PLC's contain self diagnostic routines, this includes communication checks between the main control unit and the I/O modules. If a fault is found it is handled here before any of the execution is allowed to proceed.
- **Input Scan:** All inputs both from the input modules and from the internal memory are read. This is done in a single step to make sure that all future calculations for the currently executing scan has consistent data. You may note that updates are not read during logic solve and will be delayed until the next input scan.

- **Logic Solve / Scan:** Calculations and computations from the user programs are computed in this step. If values are to be stored back into internal registers they are now put into temporary registers. Similarly if external output is required it is written to a temporary internal register that will hold the output until the output phase is executed.
- **Output Scan:** Internal temporary registers are written to their destination registers in one step. External outputs take on the values held by their associated temporary registers. All outputs also take place in one step and the outcome is that it appears that all outputs change simultaneously.

In each scan operations are modelled as if executions happen concurrently thus, the order of individual instructions in each phase is not important. All of this is done since Ladder Logic (we refer the reader to Section 3.1) executes concurrently on multiple rungs in order to emulate the behaviour of electrical circuitry. Internally however PLC's are sequential machines with a deterministic order in which instructions are executed. This is a side effect of using microcontrollers in the main control unit instead of relays and circuitry. Many temporary registers are used to store inputs and outputs so that at the beginning of each phase they can be latched in one step. This makes the reading of multiple inputs and outputs occur concurrently.

The input and output modules generally connect to the main module via serial links however some manufacturers also include network communication over standard shielded Ethernet [2, 3]. Generally, serial communication is used more often when the input and output modules are at a close distance to the controller unit as in modular PLC designs (see Figure 2.4). The network interface on the other hand is used when the input or output module needs to be located far away from the main controller unit [3] as is often the case in automated production facilities. Output modules are generally relay driven and the driving current is provided by a transistor connected to logic pins of the main controller. This is done to isolate the internal circuitry from the high current demands of driving heavy machinery [7]. Alternatively some circuits employ an opto-isolator circuitry to achieve the same effect. The trade off in this



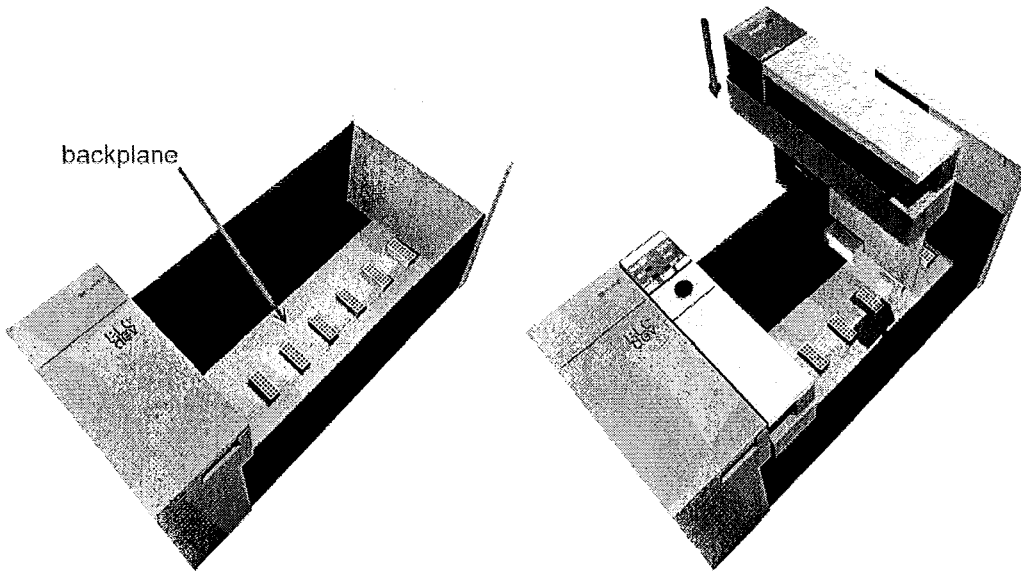


Figure 2.4: 3D Diagram of A Modular PLC With One Module Being Inserted [11, 12]

design is it will accommodate less current under load but has faster switching and overall better service life [7]. Analog outputs are obtained by passing a binary value through a digital to analog converter (DAC). A reference voltage is usually required in such configurations.

All input and output modules contain some common features which allow for modularity. Each I/O module is assigned a unique address so that the controller unit running the PLC program can access it. Each controller also has what is referred to as a backplane which contains the necessary connectors to connect to the bus so the CPU can access it. Most I/O modules have multiple channels where each channel is either a single ended wire or a differential pairs. Differential pairs can be commonly seen in analog input modules where it is preferable to compare a signal with the sensor's reference ground. This is done to avoid problems with the ground mismatch between the PLC's CPU and the sensor. Such a mismatch would produce an undesirable floating ground and would induce a permanent error on the input channel. Most modules are designed so that the external inputs and outputs are completely isolated from the CPU circuitry. As such, the CPU is connected to common ground via the

serial bus while the input and output modules are usually connected to the power supply via a ground screw (usually marked “common”). In analog input modules conversion circuitry must exist to convert the sensed input values into quantized values to be sent over the serial bus. The same is true for analog output modules but in reverse. In digital inputs conversion is not needed however circuitry still exists for isolation of the internal bus from the external input, and where applicable the module itself may perform noise correction before the actual input enters the PLC itself. In addition to individual modules some PLC manufacturers opted to have the I/O modules embedded into the main CPU unit. These are generally more commonly found in the micro sized PLC’s and are generally less expensive than their expandable counterparts. Although externally they are not expandable internally the configuration of the input modules are actually identical.

Due to the varying requirements of manufacturing over the years a wide variety of I/O modules now exist to fill in every need. A short list of common I/O modules for the Allen-Bradley SLC500 series of PLC’s is given below [8].

- **Analog Input Module:** Reads in analog voltages, converts them into digital values and sends the information to the control unit.
- **Analog Output Module:** Takes in digital values and produces an appropriate analog voltage level.
- **ASCII Input Module:** These modules take in character information usually via serial links (RS-232) and convert it into a form the PLC controllers can understand[8].
- **ASCII Output Module:** Takes PLC information converts it to an character representation and sends it out via RS-232.
- **Barrel-Temperature Module:** This module monitors four zones of an auto-tuned PID heating or cooling unit for temperature control. Molding machines and extruder employ these modules for controlling barrel temperature while injecting material [8].

- **BCD Input Module:** This module reads in inputs from devices that output binary-coded decimal (a method of representation where each decimal digit is represented by four bits). An example of such a device is a thumb wheel [8].
- **BCD Output Module:** Essentially does the reverse of the input module, usually used for compatibility when devices expect data in BCD format.
- **Discrete Input Module:** These are isolated digital inputs.
- **Discrete Output Module:** These are isolated digital output modules.
- **Encoder Counter Module:** Keeps track of angular positioning of shafts.
- **Grey Encoder Module:** Receives grey-code signals from a device that produces grey-code.
- **High-Speed Counter-Encoder Module:** This type of module allows counting and encoding at a much faster rate than is normally possible inside a PLC program.
- **PID Module:** These modules enable the user to perform closed loop automatic control based on proportional integral and derivative values. If properly tuned the module can hold a process at the desired set point.
- **Synchronized Axes Module:** Provides logic to synchronize machines like hydraulic tailgate loaders, forging machines, and rolling-position operations.
- **Thermocouple Modules:** These modules allow the PLC to read temperatures of a process and will output usually with Celsius or Fahrenheit.
- **TTL Input Module:** Allows reading inputs from TTL devices such as most integrated circuits.

Programmable logic controllers can be classified into three categories: Integrated such as a Mitsubishi Melsec FX-32MR seen in Figure 2.1, Modular as seen in Figure 2.2,

and Large Scale automation. The integrated category includes small one board solutions generally better suited for low power or embedded applications. The modular category consists of PLC's that have a rack that houses the power supply, and several modular slots for both the microcontroller and various input and output modules.

Compact units such as the Melsec previously mentioned commonly have an input output port range from 8 - 24, these are not expandable due to its all-in-one nature. The Melsec unit has 2000 steps of execution memory. Each step can be loosely thought of as an instruction. However, the nature of Ladder Logic programming tends to obscure what a step is after it is compiled. The output capacity of these compact units can range anywhere from milliamps to a few amps. As an example the Melsec FX-32MR in Figure 2.1 can output a maximum of 8A on 4 of its ports and only 2A on the remaining ports. Modular units on the other hand are able to scale to the diverse demands of operating heavy machinery by fitting them with high capacity output units.

# Chapter 3

## Introduction to Ladder Logic

### 3.1 Background of Ladder Logic

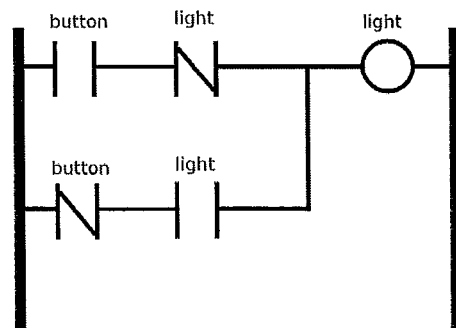


Figure 3.1: Toggable Light in Ladder Logic

Ladder logic was originally developed to replace physical relays used in PLC's. As a result the "language" resembles a circuit diagram. The left most and right most "rung" represent power rails analogous to GND (or negative) and VCC (or positive) what's placed in between those rungs are the load components [4]. In the case of programming, the entire logic is created from loads you place in between these power rails. Each horizontal "rung" is then its own independent electrical circuit.

Several conventions exist, for example power always flows from left to right along each rung. Power also flows from top to bottom along the rails. This is counter

intuitive since Ladder Logic is suppose to be analogous to a circuit schematic and a circuit is simply energized with no implicit power flow direction. In addition each run must start with inputs and end with exactly one output. Any device that is on a rung is shown in its initial state. This can be open or closed for inputs.

Modern PLC's operate more like a traditional micro controller and thus the original schematic based language can prove to be awkward to work with.

The inputs in Ladder Logic are referred to as loads. A normally open input is represented by the symbol  $-| \text{ } |-$ , and a normally closed input by  $-|/|-$ . In addition an address is usually assigned to each input referring to which port on the physical PLC the input is connected to. Logical AND can be formed by having two logical loads on one rung [4]. Similarly logical OR can be formed by creating a branch along one rung as shown in Figure 3.5.

We can define the language of Ladder Logic as follows  $Q = \langle M, S, C, R, P \rangle$

- M: set of monitored variables.
- S: set of state variables.
- C: set of controlled outputs.
- R: set of rungs.
- P: set of power rails.



Figure 3.2: Basic Ladder Logic Diagram

The most basic structure of Ladder Logic is shown in Figure 3.2. We have

$$M = \{in_1\}, S = \{out_1\}, C = \{out_1\}, R = \{rung_1\}, P = \{L, R\}.$$

Note that  $rung_1$  is not explicitly labelled in Ladder Logic but we give it a name here so we may refer to it and complete our model.

The semantics of Figure 3.2 is then given by:

Action	Result
$@T(in_1 = true)$	$out_1 := true$
$@T(in_1 = false)$	$out_1 := false$

Table 3.1: Semantics for Fig 3.2

Where  $@T(< condition >)$  is used to denote the positive edge of a condition becoming true. We also assume negligible delay between the action occurring and the result being asserted. It is important to note that our function table must be complete that is have an entry for all possible combinations in the input domain.

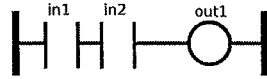


Figure 3.3: Simple AND Logic Diagram

When multiple inputs are connected on the same rung it is interpreted as a logical AND expression. In Figure 3.3 we can expand our model to:

$$M = \{in_1, in_2\}, S = \{out_1\}, C = \{out_1\}, R = \{rung_1\}, P = \{L, R\}.$$

We can see that both  $in_1$  and  $in_2$  are on  $rung_1$ . This is interpreted as follows:

Action	Result
<i>Initial</i>	$in_1 := false, in_2 := false, out_1 := false$
$@T(in_1 = true \wedge in_2 = true)$	$out_1 := true$
$@T(in_1 = false \vee in_2 = false)$	$out_1 := false$

Table 3.2: Semantics for Fig 3.3

The conditions  $in_1$  and  $in_2$  are combined to form our composed action  $@T(in_1 = true \wedge in_2 = true)$  as seen in Table 3.2. We also note that there is no action for each individual condition becoming true nor do we need to individually calculate the timing on  $in_1$  or  $in_2$ .

In addition to multiple inputs connected to the same rung, inputs can also be branched. A branched rung as show in Figure 3.4 behaves like a logical OR. In

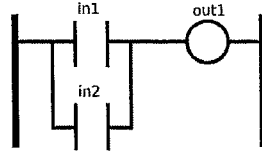


Figure 3.4: Branching Rungs

addition two or more rungs can be joined as shown in Figure 3.5. The semantics are equivalent for both Figure 3.4 and Figure 3.5.

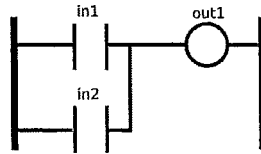


Figure 3.5: Branching Rungs (Alternative)

Action	Result
<i>Initial</i>	$in_1 := false, in_2 := false, out_1 := false$
$@T_n(in_1 = true \vee in_2 = true)$	$out_1 := true$
$@T_n(in_1 = false \wedge in_2 = false)$	$out_1 := false$

Table 3.3: Semantics for Fig 3.4 and Fig 3.5

Since the semantics are the same for both Figure 3.4 and Figure 3.5 we can express both outcomes with function Table 3.3. As before in Table 3.2 in Table 3.3  $in_1$  and  $in_2$  are composed to form our composed action  $@T(in_1 = true \vee in_2 = true)$ . However it is also possible to represent this in another way as we will show below.

In Table 3.4 we choose to represent  $in_1$  and  $in_2$  as separate inputs. This matches Figure 3.5 more closely but also makes any verification harder than Table 3.3. For smaller examples Table 3.3 make more sense since you can verify relatively simple smaller functions quite fast. If a system becomes reasonably large however there might be motivation to use the style shown in Table 3.4 since it will allow more complex functions to be decomposed into simpler inputs. Since semantically the two are equivalent this thesis will focus to the first convention.



Action	Result
<i>Initial</i>	$in_1 := false, in_2 := false, out_1 := false$
$@T(in_1 = true)$	$out_1 := true$
$@T(in_2 = true)$	$out_1 := true$
$@T(in_1 = false \wedge in_2 = false)$	$out_1 := false$

Table 3.4: Semantics for Fig 3.4 and Fig 3.5

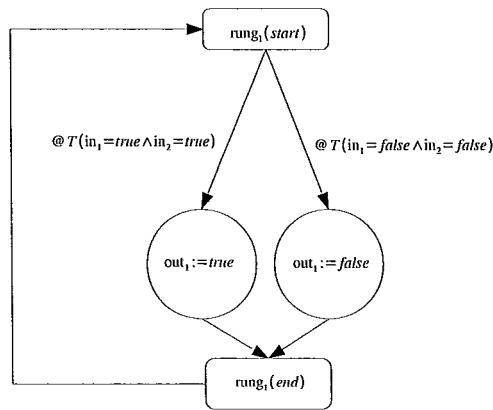


Figure 3.6: State diagram conversion for Table 3.2

A rung can be defined as a directed graph with exactly one  $rung_1(start)$  and one  $rung_1(end)$ . The state variables form guard conditions along the edges. A branch in this case represents 2 edges leaving one node. For example Figure 3.3 can be easily converted into a state diagram by observing the results in Table 3.2. Each row of Table 3.2 is directly converted into an edge with the appropriate guard conditions. Each output assertion is given their own state. Finally in Figure 3.6 we observe the start state of the rung, and the end state for the rung. In Ladder Logic each rung is executed continuously with each scan, therefore it is necessary to also add an loop from  $rung_1(end)$  to  $rung_1(start)$ . We can similarly take Figure 3.4 observe its Function Table 3.3 and produce a state diagram from its function table. Figure 3.7 represents said construction.

Thus, each rung can be converted into an equivalent state diagram by examining

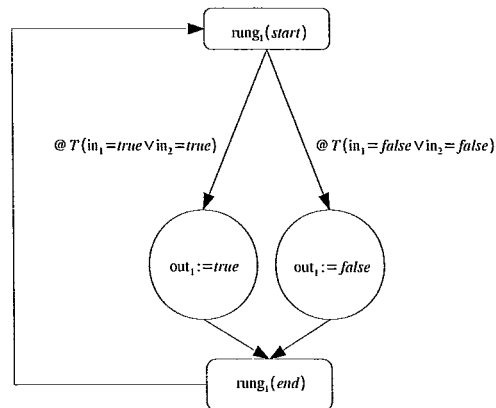


Figure 3.7: State diagram conversion for Table 3.3

its function table, assigning the state variables to guard conditions, and creating a state for each of the outputs. For thoroughness we complete this procedure to produce a state diagram shown in Figure 3.8 which is a representation of Figure 3.5 and its corresponding Table 3.4.

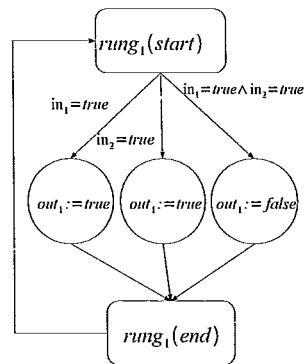


Figure 3.8: State diagram conversion for Table 3.4

Another concept in ladder logic is latching. Suppose you have a light that you want to be able to turn on and stay on after you push a button. With any of the ladder diagrams shown above the light would go out as soon as the buttons were all

released. In order to keep the light on and constantly on after the button is pressed we introduce the latch. We modify our diagram shown in Figure 3.5 making the output feed back into one of the inputs to our OR circuit. The result is show in Figure 3.9.

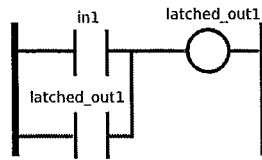


Figure 3.9: Latched Ladder Logic circuit.

The latched circuit operates the same way as the behaviour show in Table 3.4; the key difference is the feedback of the OR circuit replaces the second row of the function table. This causes *latched\_out1* to stay on after the initial input is removed.

All Ladder Logic programs are built of these basic primitives in addition each manufacturer offers their own convenience functions and additional blocks. These allow operations like add, subtract, and taking a measurement to be done in one block rather than attempt to construct it through the circuit diagram.

# Chapter 4

## Software

### 4.1 Goals

This project aims to improve on current industrial programmable logic controllers by introducing a more natural graphical programming method. In addition we will evaluate how to create a cost effective alternative using off the shelf parts to construct our own hardware PLC. In the process we aim to produce a final prototype that will have the same basic feature set of modern PLC's. This includes a main controller unit, with input and output capabilities, and a prototype of a basic IDE that will work in our new visual language. In this project we propose state machine diagrams as the method of choice. It is also important for this project to understand the deficiencies of Ladder Logic (the current method). In addition we will evaluate the original use of PLC's and if the old methodologies are still applicable to their modern application. This analysis will provide further understanding on how the original programming methods have been outpaced by more recent technology.

Due to the scope of this project we must deliver several components in order to achieve an acceptable proof of concept. The components and their sub-goals are as follows:

- We must deliver a main controller unit with an on-board embedded OS. This main controller will serve as our CPU and will include the input and output

units in addition to running our program. The main controller must be able to execute adequately fast in order to compete against commercial PLC's. Since the user is not concerned with the execution speed of each instruction, fast refers to the time it takes for an input to trigger an output. We hope to utilize cheaper hardware but still equivalent speeds by allowing the user to specify instructions more closer to the actual chip supported instructions. This reduces the actual number of pseudo instructions required to perform a task. Our hardware will aim to not require executing in scans (see Figure 2.3) and in doing so we hope to make better use of available hardware.

- We must also design and formalize a visual programming language to be used as a replacement for Ladder Logic. This language will require precise definitions on how to interpret diagram elements. The language should also be designed to be easily understandable and generate efficient code. It should also be advanced enough to construct basic control programs. To practically utilize this language we will deliver a proof of concept IDE that will allow the user to enter programs using the visual programming language instead of Ladder Logic. This new IDE should work like a flow chart drawing program allowing the user to add and remove logic blocks at will. We aim to make this interface as intuitive as possible and also provide a fast efficient translation into machine code. In addition a simulator is also required to help developers visualize a program's execution. The simulator should contain basic step features to allow the programmer to step through each transition, and allow the developer to examine the contents of the variables in memory. The main goal of the IDE will be to provide an easy to understand visual environment where the user may enter their program and to minimize error where possible.
- The final layer that would be added is a software to hardware compiler that will take user generated programs from the IDE and produce actual machine code. This would be achieved in two parts. First the IDE will compile the diagram into intermediate code. This intermediate code would be code that would stay

---

hardware neutral and would resemble C. The intermediate code would contain many calls to functions that would be supported on the chip. These functions would be part of our hardware framework that would take the calls placed by intermediate language and translate them into hardware specific calls. From this design choice hardware specific code will stay on the hardware portion of this project and compiled software will stay hardware independent with specific support implemented by the hardware framework.

By delivering an initial proof of concept software and hardware system this project would allow for further development for modernizing programmable logic controllers. In addition it may also serve as a practical example of visual programming languages that are used to program embedded devices.

## 4.2 Overview of State Machines

Our proposed visual programming language is modeled by state machines with guarded edges. The language to describe the state machines differs in a few areas from the *UML2 State Machine Diagram*[1] syntax in order to support features of the hardware. To better understand these differences we will first introduce the syntax and semantics of more traditional *Finite State Machines*[5] and *UML2 State Machine Diagrams*[1].

**Definition 4.2.1** *A Finite State Machine is defined as  $M = \{Q, I, Z, \delta, \omega, q_0\}$*

- $Q$ : Set of states.
- $I$ : Set of input symbols.
- $Z$ : Set of output symbols.
- $\delta$ : A state transition function:  $I \times Q \rightarrow Q$ .
- $\omega$ : An output function:  $Q \rightarrow Z$
- $q_0$ : Starting state.

A state represents an operating condition. For example in Figure 4.1 the states are “Light On” and “Light Off”.

Given a state  $q_1 \in Q$  an input  $i \in I$  a transition is then a function  $\delta(i, q_1) = q_2$  where  $q_2$  is the next state and  $q_2 \in Q$ . Outputs are formalized to  $\omega(q_1) = z_1$  where  $z_1 \in Z$ . Machines that satisfy Definition 4.2.1 are referred to as *Moore machines*[5]. A *Mealy machine* is obtained by adjusting outputs to be dependent on inputs as well as states. To obtain a *Mealy machine* from Definition 4.2.1 we change our output function to  $I \times Q \rightarrow Z$ . Although Figure 4.2 contains outputs for all edges, we can define a null or a no change output in order to simulate an edge having no output as well. For purposes of the *Logic Control Chart* language we do not require the power of a Mealy machine and thus we chose to keep our definitions closer to a Moore machine. More details of this decision can be found in Section 4.3.

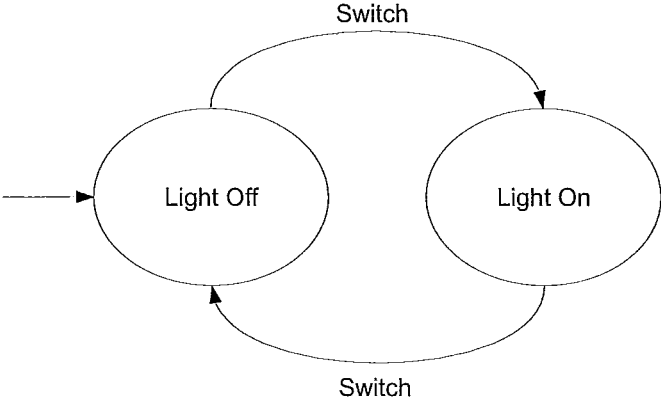


Figure 4.1: Simple Toggle Light

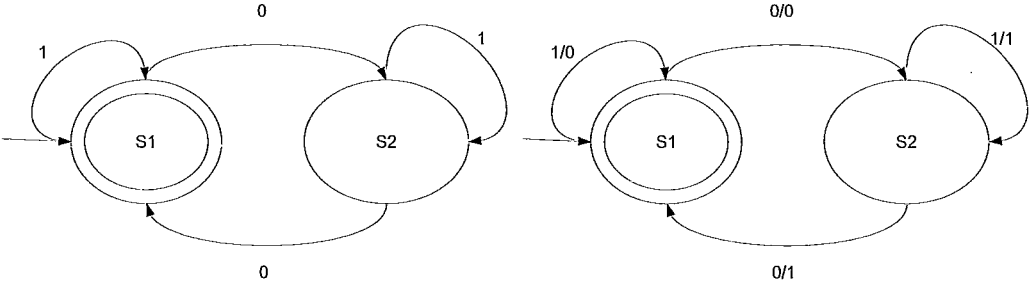


Figure 4.2: Moore (left) and Mealy (right) State Machines



To describe our state machines we use diagrams such as adopted by *UML*[1]. There are several ways a starting state can be drawn as seen in Figure 4.2. One such way is to draw a edge that has no state connected to its tail. In our system we choose incorporate a symbol similar to the *UML*[1] symbol where the start state edge has a solid dot connected to the tail as shown in Figure 4.3. *Logic Control Chart* utilizes a special *start* state that is functionally identical to *UML*'s solid dot symbol.

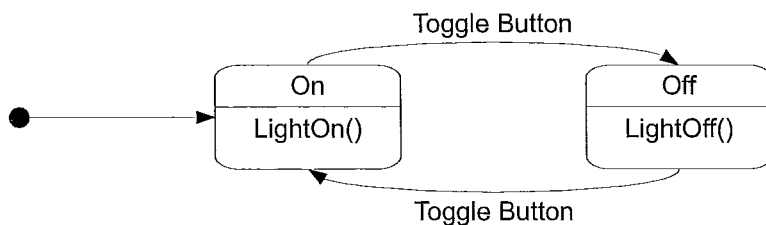


Figure 4.3: UML State Diagram of a Toggle Light

The *UML State Diagram*[1] as shown in Figure 4.3 also allows for state titles in each state seen at the top of each state. In addition, each state may contain actions that are executed. In Figure 4.3 “LightOn()” and “LightOff()” refer to executed actions that are called once the “On” and “Off” states are reached. The behaviour is read as once the state “On” is reached “LightOn()” is executed right away. More than one instruction to be executed can be listed and is understood that each instruction is executed sequentially[1].

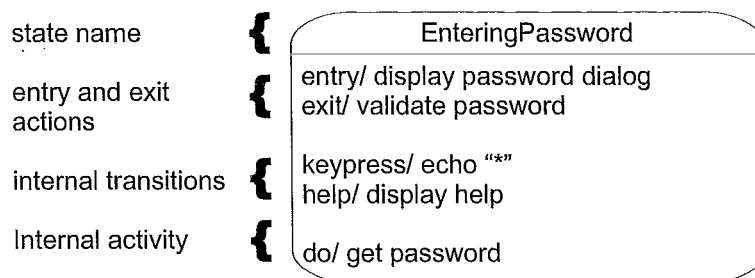


Figure 4.4: UML2 State Machine Diagram Syntax[1]

The syntax shown in Figure 4.3 is that of *UML2 State Machine Diagram*[1]. Each state bubble in *UML2 State Machine Diagram* contains the information given by Definition 4.2.2 and shown in Figure 4.4. The additional information makes UML2 more useful for representing programs more conveniently.

**Definition 4.2.2** *UML2 State Machine Diagram by convention is given by the following:*

- **state name:** *The state name is mandatory for each UML2 state in cases where this is the only information the horizontal divider may be omitted.*
- **entry and exit actions:** *Entry and exit actions are optional, they are commonly used for initializers and finalizers that may occur in each state.*
- **internal transitions:** *These optional fields refer to simple transitions that may happen in the state itself generally these are simple enough to not require a separate diagram.*
- **internal activity:** *These are optional and refer to activities that occur or are executed while in the state.*

### 4.3 Logic Control Chart

The language used for our project was purposely modelled after *UML2 State Machine Diagram*. As discussed in section 4.2 UML2 is similar to state machines with additions to describe behaviours internal to states. The motivation in using *UML2 State Machine Diagram* as a basis for *Logic Control Chart* are as follows: First *UML2 State Machine Diagram* is quite well known and popular, this improves the potential acceptance of our tool. Secondly *UML2 State Machine Diagram* state machine syntax is more concrete than the other forms of state machines, giving a strong basis to construct our language around. Finally *UML2 State Machine Diagram* has the concept of internal activity or internal execution necessary for modelling the behaviour of an actual useful program.

The model of Logic Control Chart is expressed mathematically as follows:

**Definition 4.3.1** *Logic Control Chart*

- $Q$ : Set of modes
- $V$ : The state space  $V = \langle V_0, V_1, V_2, \dots, V_n \rangle$  where  $V_i \in \{\mathbb{Z}_{128}, \mathbb{Z}, \mathbb{B}, \mathbb{R}\}$
- $\mathbf{v}_{init}$ : vector of initial values  $\mathbf{v}_{init} = \langle v_0, \dots, v_n \rangle$  where  $(v_i \in V_i)$
- $G$ : Set of guard conditions  $V \rightarrow \mathbb{B}$
- $A$ : Set of assignments  $V \times Q \rightarrow V$
- $\tau$ : Set of transitions  $G \times Q \rightarrow Q$
- $q_0$ : The initial starting mode

Many states in the system can belong to the same mode. The state space in our system is used for model variables. All variables in *Logic Control Chart* are typed. The initial value in the system is given by a state  $\mathbf{v}_{init}$  in the state space  $V$ . Guard conditions are mappings from states spaces to boolean values.

Assignments in our system take the form  $V \times Q \rightarrow V$ . Suppose we the state  $\langle 1, 2 \rangle$  meaning  $v_1 = 1, v_2 = 2$  suppose we are in mode 5 an assignment  $v_1 := v_2, v_2 := 0$  can be described by:  $(\langle 1, 2 \rangle, 5) \rightarrow (\langle 2, 0 \rangle)$ . Assignments to variables are always represented by changes to the state.

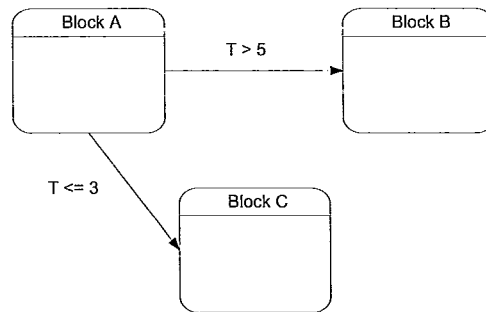


Figure 4.5: Example of Basic Transitions

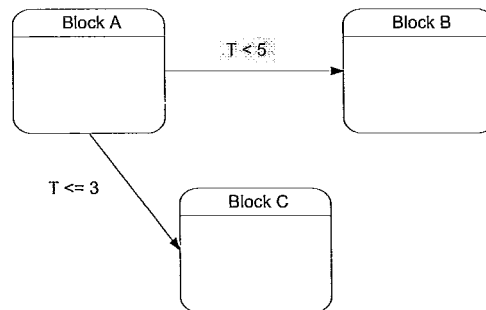


Figure 4.6: Incorrect Transitions

Transitions in our system must start from an block object and must terminate at another block. Since *Logic Control Chart* does not support non-deterministic execution, transition conditions must be mutually exclusive to be valid. All edges departing a mode must have  $\forall q(q, g_1, q') \in \tau \wedge (q, g_2, q'') \in \tau \ q' \neq q'' \rightarrow g_1 \wedge g_2 = false$ . In Figure 4.5 we see three possible outcomes  $(T <= 3, \text{"Block A"}) \rightarrow \text{"Block C"}$ ,

$(T > 5, \text{"Block A"}) \rightarrow \text{"Block B"}$ , and for  $3 < T \leq 5$  we stay in "Block A" since there is no transition to take us out. An incorrect usage of transitions is shown in Figure 4.6. Values of  $T \leq 3$  will cause both guard conditions to become true; violating our requirement for mutual exclusion. *Logic Control Chart* considers the outcome of multiple possible transitions as undefined and is invalid in the language.

## 4.4 Syntax of Logic Control Chart

Syntax of Logic Control Chart language:

**Definition 4.4.1** *Logic Control Chart* is given by the following grammar:

- $Eq \rightarrow = | \neq | < | \leq | > | \geq$
- $Op \rightarrow + | - | * | / | \% | \&\& | // | ^$
- $Expression \rightarrow Expression Op Expression | Op Expression | Variable | \mathbf{Constant}$
- $Type \rightarrow \mathbf{byte} | \mathbf{int} | \mathbf{long} | \mathbf{bool} | \mathbf{float}$
- $Variable \rightarrow [A - Za - z]^+ [A - Za - z0 - 9]^*$
- $Condition \rightarrow Expression Eq Expression | Variable$
- $StoreBlock \rightarrow Type Variable := Expression | Type Variable := Expression; StoreBlock$
- $DelayBlock \rightarrow Expression$
- $OutputBlock \rightarrow \mathbf{PORTOUT} := Variable$
- $InputBlock \rightarrow Type Variable := \mathbf{PORTIN}$
- $Block \rightarrow StoreBlock | OutputBlock | DelayBlock | InputBlock$
- $Transition \rightarrow \mathbf{EMPTY} | Block Condition Transition$
- $Program \rightarrow \mathbf{StartBlock} Transition$

In Figure 4.7 we see a store block represented as it is drawn by the Logic Control Chart IDE. Each line consists of a type (can be seen in Figure 4.7 we have **int**, and **byte**) an identifier, the equals symbol, and a expression. An expression can take the form of a constant as shown in Figure 4.7. Our syntax also allows for many lines in each StoreBlock. This design choice allows formulae to be expressed more

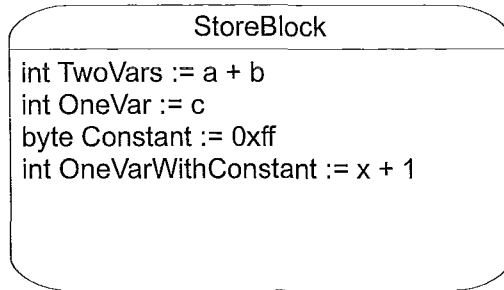


Figure 4.7: Example Of A Store Block As Implemented In PLCEdit

conveniently as it can be done in sequence rather than a sequence of blocks used together to perform a set of operations. Each of the assignment lines in a store blocks are read in sequence, and is understood to occur one right after the other. This is described in more detail below.

$$v_1 := v_0 \tag{4.1}$$

$$v_0 := v_1 \tag{4.2}$$

If assignments in *Logic Control Charts* would happen concurrently 4.1 and 4.2 would produce a swap where  $v'_1 = 'v_0$  and  $v'_0 = 'v_1$  (where  $'v$  denotes the value of  $v$  before the assignments are made and  $v'$  the value of  $v$  after the assignment is completed). We read assignments such as in equations 4.1 and 4.2 as these assignments will take place upon entry of the mode. For *Logic Control Charts* we read them in a sequential fashion for example: Upon entry of the mode, first Equation 4.1 is assigned, then Equation 4.2. That is  $v'_1 = 'v_0$  and then  $v'_0 = v'_1$  which results in  $v'_0 = v'_1 = 'v_0$  after the second assign statement.

The advantage of sequential assignments over concurrent assignments can be seen when trying to compute a 5 stage fir filter as seen in Figure 4.8. In Figure 4.8 sequential assignments are much more convenient when making many small assign-

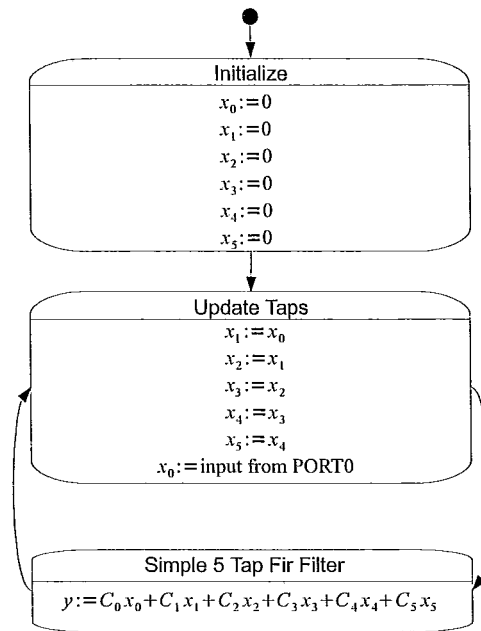


Figure 4.8: 5 Tap Fir Filtering

ments. Without sequential assignments many more mode blocks would be required to achieve the same behaviour.



## 4.5 Implementation

### 4.5.1 System Overview

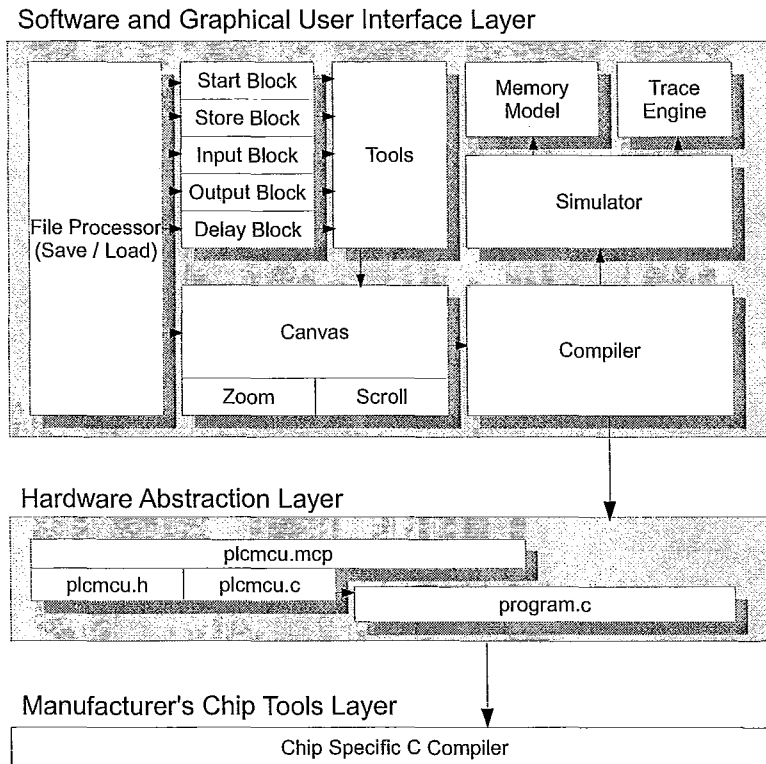


Figure 4.9: System Overview

The entire project can be logically grouped into three major sections pictured in Figure 4.9. At the upper most layer we have the high level software implementations. The main part of the GUI is implemented in this layer. Simulation tools are also implemented in this layer as they require access to the GUI.

The next layer down is the hardware abstraction layer. This allows the code generated by the IDE to be run on different hardware platforms. This layer is talked about in detail in the Hardware section.

The last layer is the chips tools layer. This layer is supplied by the manufacturer of the microcontroller.

## 4.5.2 GUI and IDE

### 4.5.2.1 Introduction

The GUI and IDE are both implemented in JAVA with the JHotDraw 7.1 framework. Early on it was decided that the GUI functions like other popular tools for drawing flow charts such as Visio, IBM's Rational Rose software, and Dia. All of these tools are interactive in the drawing of the diagram rather than compile a graph from a descriptive language. The reason for enforcing interactive drawing is that this tool is designed to simplify the original implementation of ladder logic, and building it on a textual graphical language will significantly hurt the primary goal of ease of use. JAVA was chosen for portability as it was simpler to deploy a JAVA implementation than multiple ports for each target in the the short duration of this project.

The GUI itself remains minimal; there is a toolbar in which objects can be selected from the tools palette and drawn. Properties of an object are directly editable on the object itself once drawn instead of on a separate property palette. Again this design decision was chosen as it is more immediately understandable and intuitive to the user, rather than for power and expressiveness.

### 4.5.2.2 Using The GUI: Parts of the GUI

**The Main Canvas** The main canvas represents the main working area for the user as shown in Figure 4.10. As one would expect from a drawing program the canvas can be scrolled by using the horizontal and vertical scroll bars. In addition the drawing can also be zoomed in and out by using the zoom buttons located at the bottom of the canvas. Object manipulation is accomplished by selecting the appropriate tool from the tools palette then manipulate the object directly in the drawing canvas.

**The Tools Palette** The tools palette is where the user selects his/her tool to use on the canvas shown in Figure 4.11. Only one tool can be used at a time. All tools except for compile and simulation are selected for use on click. Compile and

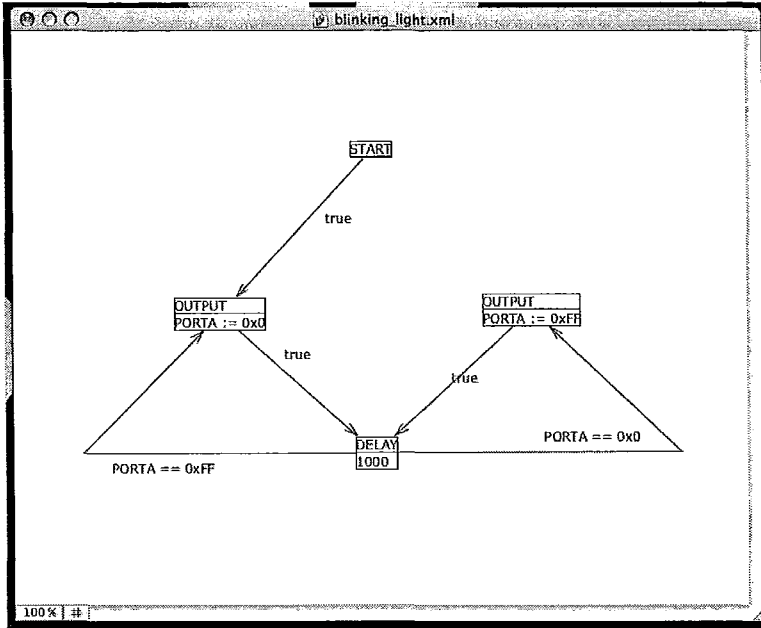


Figure 4.10: The Drawing Canvas

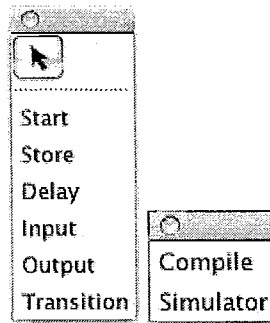


Figure 4.11: The Tools Palette

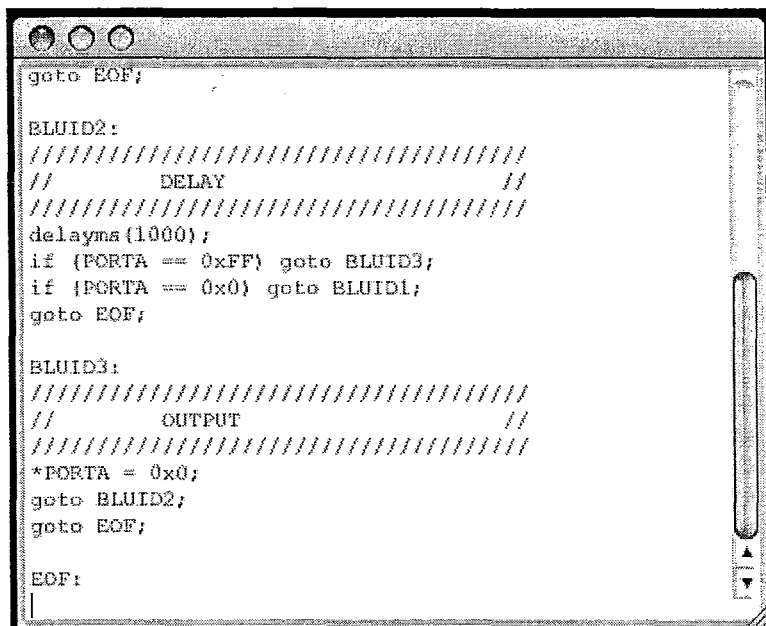
simulation are immediately executed on click. The tools listed on the tools palette are as shown in Figure 4.11 and have the functionality as follows.

- **Selection Tool:** This tool allows you to select one or multiple objects. Or edit properties of objects. You can move the object around the canvas by first selecting the object with a single click, then clicking and dragging the object to the desired location. Editing properties are accomplished by double clicking the property you would like to edit on the object itself.
- **Transition Tool:** This tool draws transitions between one block to another. Before you draw a transition you must have created the two blocks you wish to connect. To draw the transition you start by clicking and holding down the left mouse button over the starting block then dragging until the line snaps over the ending block. On release of the left mouse button a transition is formed and the two blocks are linked. For layout purposes you may also double click on a transition line to add more anchors.
- **The Start Tool:** The code will give a compile error if more than one start block is placed on the diagram. Start blocks have no other data associated with them but they serve as the starting point of your program when the PLC unit is first turned on. Your diagram must contain exactly one start block and there must only be one edge leaving the start block. No variables or guard conditions can be evaluated at the start of the program since the controller is still uninitialized so the result of the guard conditions is entirely dependent on what the chip has at these memory locations.
- **The StoreBlock Tool:** You can use these blocks to perform calculations or update internal variables. To begin drawing a store block you first select this tool. Then you click on the canvas where you would like it to appear. Store blocks are auto-resizing objects thus the size is determined by the content. To edit each field in a store block you double click the parameter you wish to edit. If you wish to delete a line you can simply erase the identifier. If you wish to

add a new line change the last identifier to something other than the default placeholder value. A new placeholder will be created as soon as you perform this action in order to allow you to add more lines.

- **The Delay Tool:** Delay blocks are used to insert a specified delay into your program the length of the delay is specified in milliseconds. When an edge is taken to the delay block the code execution is delayed and none of the departing edges are taken until the delay time specified has elapsed. It is often useful to have your program wait for a specified period, although the same can be achieved by using a store block and a counter update it is far easier and more accurate to use the dedicated delay block.
- **The Output Tool:** Output blocks are used to send outputs from the program to pins on the PLC itself. Depending on the chip type there can be one or many different output ports. Each port is a 8-bit representation of the pins, an expression is also allowed on the right hand side so masking could be done.
- **The Input Tool:** Similar to the format of the output block the input block allows you to read data from one or many ports (depending on hardware) into an internally used variable.
- **Compile Action:** Executing the compile action will start the compilation process into PLC *IL* (intermediate Language) code this code is directly compilable on targets that have implemented the hardware framework which is discussed in detail in Section 5.2.
- **Simulator Tool:** The simulator action brings up a tracing and debugging window. This window allows you to step through each of the code blocks to simulate what might happen as your program runs.

**Compile Preview Window** After the compile button is invoked the compile preview window is presented as shown in Figure 4.12. The preview window serves as a quick overview of the program code that can be loaded onto the device. The user



```
goto EOF;

BLUID2:
////////////////////////////////////
//          DELAY          //
////////////////////////////////////
delayms(1000);
if (PORTA == 0xFF) goto BLUID3;
if (PORTA == 0x0) goto BLUID1;
goto EOF;

BLUID3:
////////////////////////////////////
//          OUTPUT          //
////////////////////////////////////
*PORTA = 0x0;
goto BLUID2;
goto EOF;

EOF:
|
```

Figure 4.12: Compile Preview Window

then has the option of saving the text off to a compilable file for loading onto the device.

**Simulator Window** In order to aid in debugging, a simulator mode has been added to the visual editor. When the simulator window is brought up it helps the programmer visually identify what their current memory layout looks like as well as which block they are currently executing. In Figure 4.13 the following view is presented right after the simulator button has been invoked. Note the contents of all variables and all ports are shown. Also the start block is automatically highlighted and the ports are initialized to their starting values.

The *Step Once* button allows the user to to step through the current block one step at a time. The *Step Next* button finishes all operations in the current block and jumps directly to the next block. Figure 4.14 demonstrates what the program might look like after a it is allowed to run for a few steps.

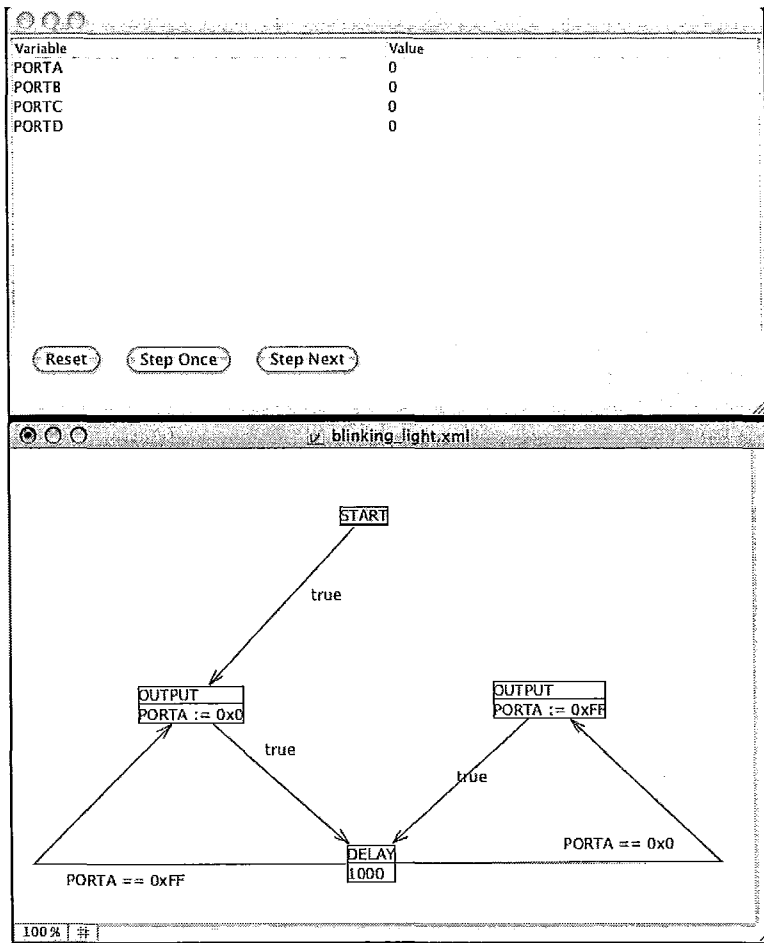


Figure 4.13: Simulator Window Starting Configuration

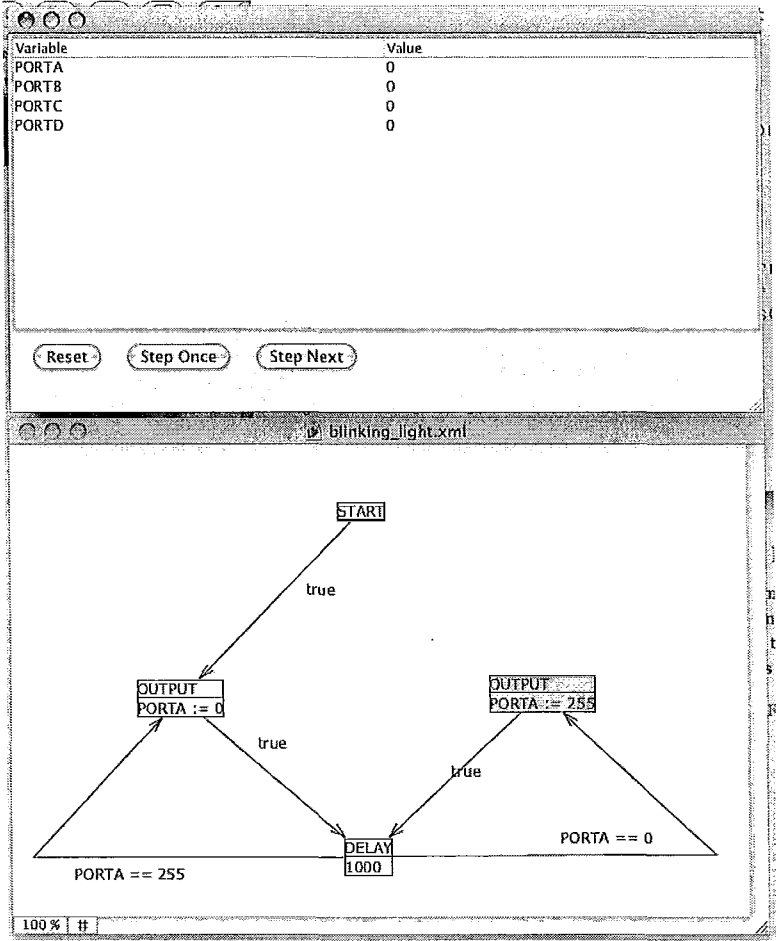


Figure 4.14: Simulator Window Program Running



### 4.5.3 Data Flow

Construction of the final compiled code starts at the GUI level. The designer starts by drawing the layout of the program they wish to construct. This drawing forms the basis of the underlying code structure. The designer then tests the behaviour of their program by utilizing the simulation tools. Once satisfied the designer uses the compile tool to generate the final program code. This program code is then compiled onto the final architecture by the embedded hardware compilation tools. Any hardware that has implemented the support required by the hardware framework (which is discussed in detail in Section 5.2) will run compile and run this program code.

### 4.5.4 Generated Code Structure

The structure of each compiled section was carefully constructed to preserve the structure of the drawn graph as closely as possible. The basic structure of each compiled block can be seen as follows:

**Definition 4.5.1** *Structure of Logic Control Chart generated code*

```

BLUID5:
////////////////////////////////////
//// Block Header                //
////////////////////////////////////
(Program Code)

if ( guard condition ) goto BLUID2;
goto EOF;

```

As you can observe from Definition 4.5.1 each code block generation first starts with a *BLUID* (Block Unique Identifier). The purpose of the *BLUID*'s is to give a reference point for the start of the block having a *BLUID* label for each block enables any block to be jumped to should we want to. The block header which comes right after is an identifier to the type of block being generated.

In the “(Program Code)” section of Definition 4.5.1 sits the generated *IL* code. This code is the same for each target platform. Each *IL* target will have appropriate implementations that will map *IL* calls to hardware specific calls. These calls can be found in Section 5.2. The generated *IL* code resembles C code with specific calls to lower level functions that are supported by the PLC target’s hardware framework.

The final block of if statements implement the edges leaving the block. Each edge is guarded by an if statement, and will have the goto portion point the the correct *BLUID* as specified by the diagram. *goto* was chosen over conventional control structures as it provided several advantages. The first is that translation from graph structure to code is fundamentally easier. *goto*’s are analogous to graph edges where trying to analyze the diagram and fit structures like *for while* are inherently much harder and prone to error. In addition these structures are more at home in textual programming languages and do not improve the structure of the graphical programming language. The next advantage was during code generation when trying to unroll structure from our graphical programming language not using *goto*’s will cause the order of items to be important. This makes it necessary to form a dependency tree of blocks to each other if they started requiring each other. All these problems were easily avoided by staying with a simple goto structure inside the actual generated *Intermediate Language* code itself. The trade off in this design decision is the resulting generated code becomes a machine generated code and is not very user editable.

Finally if at any point no edges are valid the the program will terminate by going to the label “EOF” which will be located at the end of the file.

### 4.5.5 Compiled Structure

Each of the blocks will compile their own snippets of code. It may be helpful when reading this section to refer to Definition 4.5.1 for the generic block structure definition. Each block primitive’s construction is detailed below.

#### 4.5.5.1 Start Block

```

00 BLUID#:
01 //////////////////////////////////////
02 //          PROGRAM START          //
03 //////////////////////////////////////
04
05 if (condition) goto LABEL;
06 goto EOF;

```

The primary function of a start block is to provide a block that can have edges directed at it, and to give a starting point for the program. Line 00 is the “block unique identifier” as stated in Definition 4.5.1 this is used by other blocks when edges are directed at this block. Lines 01-03 is a block description without this it becomes difficult to determine what kind of block is executing if the code needs to be examined by human eyes.

On line 06 the standard “goto EOF” is generated to ensure if no edges exist the program will stay in the block and not execute any other block by going to the end of the program.

#### 4.5.5.2 Delay Block

```

00 BLUID#:
01 //////////////////////////////////////
02 //          DELAY          //
03 //////////////////////////////////////
04
05 delay_ms(integer);
06
07 if (condition) goto LABEL;
...
08 goto EOF;

```

Since the structural components are identical to the code sample provided in the “Start Block” we will skip lines 00-04. Unlike the “Start Block” the “Delay Block” does perform an operation during execution. On line 05 the “Delay Block” calls `delay_ms(integer)` where `integer` refers to the integer specified in the diagram. The routine for delay is done on the hardware side as the manufacturer will choose how to appropriately determine 1ms on hardware. `delay_ms` is a call that will be implemented on the hardware framework show in Section 5.2.

#### 4.5.5.3 Input Block

```

00 BLUID#:
01 //////////////////////////////////////
02 //          INPUT                      //
03 //////////////////////////////////////
04
05 variable = PORTIN;
06
07 if (condition) goto LABEL;
   ...
08 goto EOF;

```

The input routine performs an operation of setting a variable to the value being read off the input port. This value is located on line 05 “`variable = PORTIN;`”. Lines 00-04 are the same as Definition 4.5.1 and line 07-08 are identical to the Delay block. In this case variable can be any variable in our program that can accept a 8 bit integer values.

#### 4.5.5.4 Output Block

```

00 BLUID#:
01 //////////////////////////////////////
02 //          OUTPUT                      //

```

```

03 //////////////////////////////////////
04
05 PORTOUT = variable;
06
07 if (condition) goto LABEL;
    ...
08 goto EOF;

```

The “Output Block” is nearly identical to the “Input Block” the only difference is on line 05 instead of reading a value from the input port a value is written to the output port. This is accomplished by the line “PORTOUT = variable;”.

#### 4.5.5.5 Store Block

```

00 BLUID#:
01 //////////////////////////////////////
02 //          STORE          //
03 //////////////////////////////////////
04
05 variable0 = expression0;
06 variable1 = expression1;
07 variable2 = expression2;
    ...
08 if (condition) goto LABEL;
    ...
09 goto EOF;

```

Again we start the store block similar to the other blocks with identical structure until line 04. Line 05-07 are the code conversions for the store block. The store block converts each line in the diagram of the form “type variable\_name := expression” into “variable\_name = expression” as seen on line 05. In addition if there is more

---

than one line they are then placed one after the other in sequence. The expressions used can be any valid mathematical "C" style expression. Functions calls used in the expressions are not supported.

## 4.6 Correctness

In this section we look at the correctness of the code generation process. Correctness can be described as the mapping shown in Figure 4.15. Expressions in our system are compiled into the program code. Expressions can be evaluated by hand ( $\delta$ ) to produce a value. Likewise the program can be executed to produce a final value. Correctness holds in the system if the value of the evaluation ( $\delta$ ) is the same as a value achieved by execution.

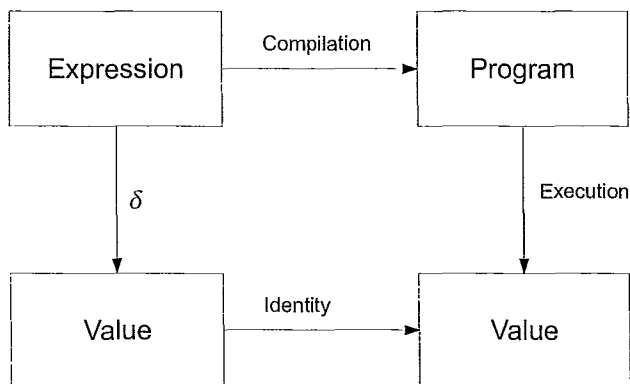


Figure 4.15: Code Transformations

Our tool constructs executable code from the drawn diagram the constructs are defined in Sections 4.4 and 4.3. To justify the correctness we need to demonstrate the program structure generated from a graph structure is correct. To do so we extend Figure 4.15 to obtain Figure 4.16 to incorporate the transition construct. If each Figure 4.15 represents an individual block than the extended Figure 4.16 represents the several blocks with the ability to transition between them.

In this section we begin by showing that each of the transformations to code are correct by examining the basic atoms. We will start by looking at atoms that have the simplest assignments first.

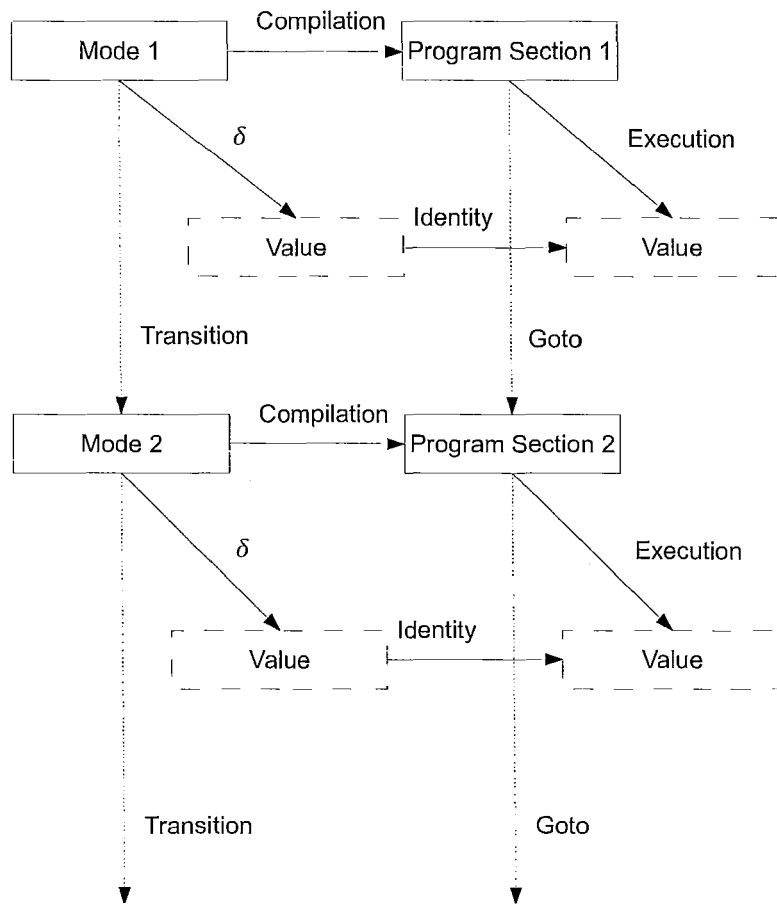


Figure 4.16: Code Transition Structure



### 4.6.1 Atoms

Each block in *Logic Control Chart* compiles to their own atomic assign statements. In the case where variables are used the type information is not present in the atomic assignment portion of the code generation. Variables are collected at compile time declared and initialized as part of the program construction. By collecting variables in this manner a variable with the same name but given two different types can be identified and will cause a compile error. Also the syntax of *Logic Control Chart* uses “:=” as an assign symbol, the final generated code will use the “=” symbol.

#### 4.6.1.1 Start Block

START
-------

Figure 4.17: Start Atom

(No atomic assignments or operations in code)
---

The start atom is the only atom in our system that does not generate any actual code for the atom itself. Instead the start atom is used as a place holder so that transitions can be constructed. It also structurally indicates where the program should start. Since the start block contains no atomic code it is trivially correct as it does nothing itself.

#### 4.6.1.2 Delay Block

DELAY
10 ms

Figure 4.18: Delay Atom

delayms(10);
--------------

The delay atom generates “`delayms(<Expression>)`”. The expression is mirrored in the diagram. The units shown in Figure 4.18 as “ms” are omitted in the code. The units in the visual representation serves as a reminder to the user that the delay is always measured in milliseconds. Expressions follow the syntax given in Section 4.4. Since there are no assignments to values, the mapping from evaluated values (none) to executed values (none) is trivially correct.

#### 4.6.1.3 Output Block

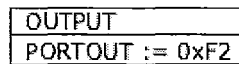


Figure 4.19: Output Atom

PORTOUT = 0xF2;
-----------------

The output atom generated refers to `PORTOUT`, which is mapped on a hardware level to the PLC hardware’s output port. This is done to allow the hardware manufacturer a bit of flexibility. In our implementation `PORTOUT` can be assigned any 8-bit value. Any values larger than 8-bit will be truncated. In the example shown in Figure 4.19 we can see that the assignment in the diagram is “`PORTOUT := 0xF2`”. We can say that the meaning of this line is  $\delta(PORTOUT) = (F2)_{16}$ . The final compiled output code is “`PORTOUT = 0xF2`”. We can see that  $Execution(PORTOUT) = (F2)_{16}$  since we have a direct mapping from the value in diagram to value in execution ( $\delta(PORTOUT) = Execution(PORTOUT)$ ), we can conclude that the assignment in the output block is done correctly with respect to the diagram.

## 4.6.1.4 Input Block

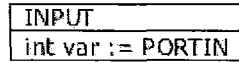


Figure 4.20: Input Atom

```
var = PORTIN;
```

We can see in Figure 4.20 that the diagram clearly shows the assignment “int var := PORTIN”. We can say the meaning of the assignment in the diagram is  $\delta(var) = PORTIN$ . The final compiled code for the atom is “var = PORTIN”. The left hand side must be a valid variable name as specified in the syntax of *Logic Control Chart* (see section 4.4). The variable can be any type however PORTIN will always be read as an 8-bit integer. Any variable on the left hand side that is not integer would have PORTIN automatically casted to the appropriate type. Thus we can see that  $Execution(var) = PORTIN$ . We note that there is a direct mapping from diagram *value* to execution *value*, that is:  $\delta(var) = Execution(var) = PORTIN$ . Thus by our definition of correctness as defined in the beginning of this section and Figure 4.15, we can conclude that the input block is correct.

## 4.6.1.5 Store Block

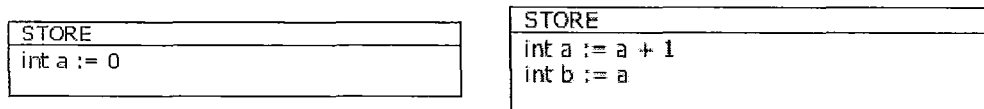


Figure 4.21: Store Atom

Single Assign (Left Diagram)

```
a = 0;
```

Multiple Assign (Right Diagram)

```
a = a + 1;  
b = a;
```

The *Store Block* is used for assignments. In the left diagram of Figure 4.21 we see a *Store Block* with one assign statement and in the right diagram of Figure 4.21 we see that store blocks can also contain multiple atomic assign statements.

Looking at the left diagram first, we note that the assign statement is “int a := 0” that is to mean  $\delta(a) = 0$ . We see the final generated code is “a = 0” meaning  $Execution(a) = 0$ . We see that  $\delta(a) = Execution(a)$  that is to say there exists a mapping from the value in the diagram to the value from the execution. We can say that the code generated is correct with respect to the diagram. Looking at the right diagram we see that each line of the *Store Block* becomes own atomic assign statement in the code. We see that the diagram has “int a := a + 1” and “int b := a” with evaluations  $\delta(a) = a + 1, \delta(b) = a$ . The generated code is then “a = a + 1” and “b = a” with the executed values  $Execution(a) = a + 1, Execution(b) = a$ . We can see at this point that  $\delta(a) = Execution(a)$  and  $\delta(b) = Execution(b)$ . Thus we can conclude that the *Store Block* generated code is correct with respect to our diagram. We note that we can create equivalent diagrams by having two single line store atoms. Thus, it is not difficult to see that the right diagram is just a visual simplification one dual line *Store Block*, corresponds to several atomic assigns. Being able to group assigns together into one atomic diagram removes the need to have several atomic assign blocks. This is ultimately removes visual clutter and makes the overall diagram simpler.

### 4.6.2 Constructors and Transitions

Constructors takes the atoms and places structural elements around them. In the example code below we demonstrate the constructed sections for any arbitrary compiled code. On lines 00-07 we have the variable initialization section, any variables used are initialized and the type is declared here. Variables in our program are collected at

compile time by scanning all the diagram objects and collecting any variables used. Any conflicts where a variable is given inconsistent types are caught and will cause a code generation error. The initialization section will set all variables to a default value after defining the type.

The generated constructs also exist to provide a way for each of the states to terminate the program if no departing transitions exist. This is accomplished by lines 32-33 an “end of file” label is generated to mark the end of the program followed by a return statement to end the program and return control to the programmable logic controller chip. On receiving a return the *Programmable Logic Controller* will hold all last known values on every port and halt, this simulates a stop.

```
00 // BEGIN VARIABLE INITIALIZATIONS //
01 int a = 0;
02 int b = 0;
03 float c = 0;
05 double d = 0;
06 byte e = 0;
07 // END VARIABLE INITIALIZATIONS //
...
32 EOF:
33 return;
```

Transitions are used to string together a sequence of atoms in order to perform the computations necessary in our program. In order to show correctness we must show that the transitions are structurally correct. We must show transitions will correctly map modes in the right sequence. Figure 4.16 shows how transitions factor into our correctness justification. Figure 4.16 states that in addition to our original correctness justification we must also demonstrate that the transitions take us to the same modes and program sections.



```
19 goto EOF;
20
21 BLUID2:
22 //////////////////////////////////////
23 //          BLOCK2          //
24 //////////////////////////////////////
25 ...
26
27 if (a >= 3) goto BLUID1;
28 if (a < 3) goto BLUID2;
29 goto EOF;
30
31 EOF:
32 return;
```

Starting by looking at the code above we can see that our first entry point into the program is the “PROGRAM START” block. This refers to the *Start Block* in our diagram. During compilation the entire diagram is scanned to ensure there is one and only one start block. The generated code for the start block is then placed at the top of the program to ensure it is the first entry point of the program after initializers.

Transitions in our program are compiled to “goto” statements. Each block is given a unique “Block Unique Identifier” which is a line label starting with “BLUID”. BLUID’s are generated as each block is scanned at compile time and a monotonically increasing number is appended to the end of the label. The “goto” transitions are inserted after the atomic block assignments have been made.

Looking at the diagram given in Figure 4.22 we note that the *Start Block* transitions to “BLOCK1” with an edge guarded by “true”. In the code “BLOCK1” has label “BLUID1” associated with its section of code. We see that the generated code for the *Start Block* has a “goto BLUID1” on line 9. This corresponds to transition

leaving the *Start Block*. Next we have a transition from “BLOCK1” to “BLOCK2” in Figure 4.22. We can see the code corresponding to the transition on line 18 “goto BUILD2” where “BLUID2” refers to “BLOCK2”. Finally we note the two guarded transitions leaving “BLOCK2”. For the guarded edge “ $a \geq 3$ ” we can see the generated code “if (a  $\geq$  3) goto BLUID1” on line 27. We note that “BLUID1” refers to “BLOCK1” in the diagram so the transition goes from “BLOCK2” to “BLOCK1” which is correct. For the transition guarded by “ $a < 3$ ” we note the generated code on line 28 “if (a  $<$  3) goto BLUID2”. We see that “BLUID2” refers to “BLOCK2” so the transition is a self loop, which is correct with respect to the diagram in Figure 4.22. Since all the transitions are generated correctly for any arbitrary block we can conclude that the transitions are structurally correct in our system.





Table 4.1: Start Diagram shown in Figure 4.23

Mode	Variables	Transitions	Next Mode
start	(none)	(none)	(none meaning stop)
(none)	(none)	(none)	(none)

To show an execution trace we replace mode with line number and we get the following Table 4.2.

Table 4.2: Start code execution for compiled code from Figure 4.23

Line	Variables	Code	Next Executed Line
00	(none)	(comment)	07
07	(none)	goto EOF	09
09	(none)	(line label)	10
10	(none)	return	(stop)

It is not to difficult to see that from lines 00 to 07 we are in the “start” mode so we can append a mode marker to the end of the table. We can also identify that line 09 “EOF” represents the end of the file and thus has no mode associated with it.

Table 4.3: Start code execution for compiled code from Figure 4.23 extended

Line	Variables	Code	Next Executed Line	Mode
00	(none)	(comment)	07	<b>start</b>
07	(none)	goto EOF	09	<b>start</b>
09	(none)	(line label)	10	<b>none</b>
10	(none)	return	(stop)	<b>none</b>

We can already see that the trace and execution will produce the same outcome if we compare the two tables. For ease of comparison we can merge the two tables side by side so we can directly compare each executed line to its associated graph.

In Table 4.4 we can easily compare the execution vs the diagram trace. We can see that line numbers can be associated with a mode despite not having one themselves. In order to verify correct execution it is necessary to show that the sequence of modes and values are the same. In the above example in which we run our first trivially simple start code snippet it is easy to see that this holds. Therefore we can conclude

Table 4.4: Start code execution combined table. For Figure 4.23

Mode	Var (Diag)	Transitions	Next	Line	Var (Exec)	Code	Next LN
start	(none)	(none)	(none meaning stop)	00	(none)	(comment)	07
				07	(none)	goto EOF	10
				10	(none)	return	(stop)

that the execution is correct for the start diagram shown in Figure 4.23.

#### 4.6.3.2 Start Diagram Execution

The start diagram analysis was already done as a previous example to construct our comparison tables please see Table 4.4. We may note that the only mode is “start” and that the mode is the same through the executed line by line trace. We can also note that the code stops after “start” mode is finished which also is correct behaviour. Finally there are no variables listed in our system so variables are trivially correct.

#### 4.6.3.3 Delay Diagram Execution

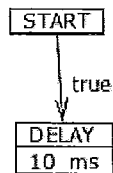


Figure 4.24: Delay Block

Generated *IL* code for diagram in Figure 4.24.

```

00 // BEGIN VARIABLE INITIALIZATIONS //
01 // END VARIABLE INITIALIZATIONS //
02
03 BLUID0:
  
```

```

04 //////////////////////////////////////
05 //          PROGRAM START          //
06 //////////////////////////////////////
07 goto BLUID1;
08 goto EOF;
09
10 BLUID1:
11 //////////////////////////////////////
12 //          DELAY          //
13 //////////////////////////////////////
14 delays(10);
15 goto EOF;
16
17 EOF:
18 return;

```

Table 4.5: Delay code execution combined table. For Figure 4.24

Mode	Var (Diag)	Transitions	Next	Line	Var (Exec)	Code	Next LN
start	(none)	if (true) delay	delay	00	(none)	(comment)	07
				07	(none)	goto BLUID1	10
delay	(none)	(none)	(stop)	10	(none)	(line label)	14
				14	(none)	delays(10)	15
				15	(none)	goto EOF	17
				17	(none)	(line label)	18
				17	(none)	return	(stop)

Once again in this example we don't have any variables so they are easily verified by checking that in both cases there are no variables. All that's left to justify correctness is ensuring the sequence of modes is executed correctly. It should be easy to see that the sequence: start, delay, stop. Is clearly implemented by the executed code from the Table 4.5. We can therefore conclude that the executed code trace is correct with respect to the original diagram.

#### 4.6.3.4 Output Diagram Execution

Generated *IL* code for diagram in Figure 4.25.

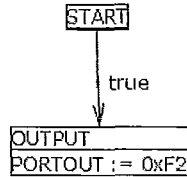


Figure 4.25: Output Block

```

00 // BEGIN VARIABLE INITIALIZATIONS //
01 // END VARIABLE INITIALIZATIONS //
02
03 BLUID0:
04 //////////////////////////////////////
05 //          PROGRAM START          //
06 //////////////////////////////////////
07 goto BLUID1;
08 goto EOF;
09
10 BLUID1:
11 //////////////////////////////////////
12 //          OUTPUT          //
13 //////////////////////////////////////
14 PORTOUT = 0xF2;
15 goto EOF;
16
17 EOF:
18 return;
  
```

Table 4.6: Output code execution combined table. For Figure 4.25

Mode	Var (Diag)	Transitions	Next	Line	Var (Exec)	Code	Next LN
start	PORTOUT = 0	if (true) output	output	00	PORTOUT = 0	(comment)	07
				07	PORTOUT = (no change)	goto BLUID	10
output	PORTOUT = 0xF2	(none)	(stop)	10	PORTOUT = (no change)	(line label)	14
				14	PORTOUT = 0xF2	PORTOUT = 0xF2	15
				15	PORTOUT = (no change)	goto EOF	17
				17	PORTOUT = (no change)	(line label)	18
				18	PORTOUT = (no change)	return	(stop)

First we make a note that PORTOUT is a special variable that is used to send an output to the ports on the device. According to the hardware specification section PORTOUT is initialized to 0 when the device first starts up. Likewise PORTOUT is zero until changed in our diagram. Understanding this the rest of the code trace is as follows:  $\{(start, PORTOUT = 0), (output, PORTOUT = 0xF2)\}$  where we observe that our tuple comprises of (mode, variables). It is not difficult to see that our two code traces produce this sequence and by our definition we can conclude that our output execution is correct with respect to the diagram.

#### 4.6.3.5 Input Diagram Execution

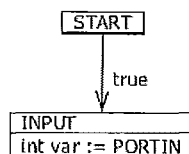


Figure 4.26: Input Block

Generated *IL* code for the diagram in Figure 4.26.

```

00 // BEGIN VARIABLE INITIALIZATIONS //
01 int var = 0;
02 // END VARIABLE INITIALIZATIONS //
  
```

```

03
04 BLUID0:
05 //////////////////////////////////////
06 //          PROGRAM START          //
07 //////////////////////////////////////
08 goto BLUID1;
09 goto EOF;
10
11 BLUID1:
12 //////////////////////////////////////
13 //          Input                    //
14 //////////////////////////////////////
15 var = PORTIN;
16 goto EOF;
17
18 EOF:
19 return;

```

Table 4.7: Input code execution combined table. For Figure 4.26

Mode	Var (Diag)	Transitions	Next	Line	Var (Exec)	Code	Next LN
				00	var = UNDEFINED	(comment)	01
				01	var = 0	int var = 0	04
start	var = 0	if (true) → input	input	04	var = (no change)	(comment)	08
				08	var = (no change)	goto BLUID1	11
input	var = PORTIN	(none)	(stop)	11	var = (no change)	(line label)	15
				15	var = PORTIN	var = PORTIN	16
				16	var = (no change)	goto EOF	18
				18	var = (no change)	(line label)	19
				19	var = (no change)	return	(stop)

We must first note in this code trace that inputs require a variable in order to store their data. These inputs are initialized at the beginning of the execution. Initialization is not required for our diagram it is understood that an initial value of 0 is

used. Our table starts with line 00, and 01 which initialize our variable, we consider this “house keeping” and on the left side of our table we do not associate this process with one of our diagram modes. This time the start mode occurs 3 rows down with the corresponding line 04. The execution of the start block and diagram is identical to previous examples with the exception of different line numbers. The sequence of modes and variables is observed as (start, var = 0), (input, var = PORTIN). We can observe that “var” is set to “PORTIN” on row 5 in the diagram. In the execution the corresponding set occurs on row 6. We note the sequence of modes and variables are identical. Thus according to our definition of correctness we have shown that both diagram and execution follow the same modes and variable values in the same sequence. We can conclude from this that the execution of the input code is correct with respect to the diagram.

#### 4.6.3.6 Store Diagram Execution

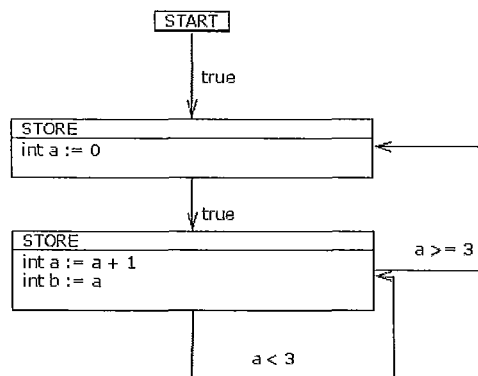


Figure 4.27: Store Block Example

Generated *IL* code for diagram in Figure 4.27.

```

00 // BEGIN VARIABLE INITIALIZATIONS //
01 int a = 0;
02 int b = 0;
  
```



```
03 // END VARIABLE INITIALIZATIONS //
04
05 BLUID0:
06 //////////////////////////////////////
07 //          PROGRAM START          //
08 //////////////////////////////////////
09 goto BLUID1;
10 goto EOF;
11
12 BLUID1:
13 //////////////////////////////////////
14 //          STORE          //
15 //////////////////////////////////////
16 a = 0;
17
18 goto BLUID2;
19 goto EOF;
20
21 BLUID2:
22 //////////////////////////////////////
23 //          STORE          //
24 //////////////////////////////////////
25 a = a + 1;
26 b = a;
27
28 if (a >= 3) goto BLUID1;
29 if (a < 3) goto BLUID2;
30 goto EOF;
31
32 EOF:
```

```
33 return;
```

Table 4.8: Store code execution combined table. For Figure 4.27

#	Mode	Var (Diag)	Transitions	Next	Line	Var (Exec)	Code	Next LN
1					00	a = UNDE- FINED b = UNDE- FINED	(comment)	01
2					01	a = 0 b = UNDE- FINED	int a = 0	02
3					02	a = 0 b = 0	int b = 0	09
4	start	a = 0 b = 0	if (true) → store <sub>1</sub>	store <sub>1</sub>	09	a = NC b = NC	goto BLUID1	12
5	store <sub>1</sub>	a = 0 b = NC	if (true) → store <sub>2</sub>	store <sub>2</sub>	12	a = NC b = NC	(line label)	16
6					16	a = 0 b = NC	a = 0	18
7					18	a = NC b = NC	goto BLUID2	21
8	store <sub>2</sub>	a = 1 b = 1	if (a ≥ 3) → store <sub>1</sub> if (a < 3) → store <sub>2</sub>	store <sub>2</sub>	21	a = NC b = NC	(line label)	25
9					25	a = 1 b = NC	a = a + 1	26
10					26	a = NC b = 1	b = a	28
11					28	a = NC b = NC	if (a ≥ 3) goto BLUID1	29
12					29	a = NC b = NC	if (a < 3) goto BLUID2	18
13	store <sub>2</sub>	a = 2 b = 2	if (a ≥ 3) → store <sub>1</sub> if (a < 3) → store <sub>2</sub>	store <sub>2</sub>	21	a = NC b = NC	(line label)	25
14					25	a = 2 b = NC	a = a + 1	26
15					26	a = NC b = 2	b = a	28
16					28	a = NC b = NC	if (a ≥ 3) goto BLUID1	29
17					29	a = NC b = NC	if (a < 3) goto BLUID2	21
18	store <sub>2</sub>	a = 3 b = 3	if (a ≥ 3) → store <sub>1</sub> if (a < 3) → store <sub>2</sub>	store <sub>1</sub>	21	a = NC b = NC	(line label)	25
19					25	a = 3 b = NC	a = a + 1	26
20					26	a = NC b = 3	b = a	28
21					28	a = NC b = NC	if (a ≥ 3) goto BLUID1	12
22	store <sub>1</sub>	a = 0 b = NC	if (true) → store <sub>2</sub>	store <sub>2</sub>	12	a = NC b = NC	(line label)	16
23					...			

The reader should note that we have shortened “no change” in the previous tables to “NC” in order to fit it in the tables. Similar to the input example the store example also has variables that must be initialized before it can enter it’s first state. Lines 00-02 in the Table 4.8 represent the “house keeping” steps required in order to define, setup and initialize the variables. The entry of the start mode represents the start

of our diagram. The reader can see from Table 4.8 the two variables are considered initialized to 0 in the diagram, that is “ $a = 0, b = 0$ ”. We may also note that this is taken care in the execution by lines 01 and 02.

On line 12 we enter our first store block which we have denoted  $store_1$ . Observe that only variable “a” is modified in this store block thus “b” takes on a value of “NC”. The corresponding operation for “ $a = 0$ ” in the diagram occurs in the execution on line 16. Finally on line 18 we can see that mode  $store_1$  is exited on the execution side by executing “goto BLUID2”.

On first entry to  $store_2$  our diagram updates “a” and “b” to 1 and 1 respectively. The corresponding line 21 of the execution is just a line label and the update to variable “a” does not occur until line 25. In the generated code the order of variable assignments is preserved so variable “a” is assigned before variable “b”. Thus, so far our execution produces the sequence (start,  $a=0, b=0$ ), ( $store_1, a=0, b=0$ ), ( $store_2, a=1, b=1$ ). We note that both the diagram and execution produce the same values at this point.

In the diagram transitions are understood to be evaluated and taken right after the work is done inside the mode. from Table 4.8 row 8 we can see that the transitions are  $\text{if } (a \geq 3) \rightarrow store_1, \text{ if } (a < 3) \rightarrow store_2$ . By evaluation  $store_2$  is the result where  $a=1$  and  $b=1$ . In lines 25-26 we can see the corresponding execution take place to produced the results for the diagram in row 8. First on line 25  $a = a + 1$  is executed so  $a = 0$  that was last set on row 6 now becomes  $a = 1$  after the execution. The following line 26 then sets  $b = a$  which is to say  $b = 1$  at this point.

On row 11 we can see that the transitions are now being evaluated. Unlike row 8 on the diagram side where we treat evaluating transitions as a parallel operation during execution we see that the transitions are evaluated in sequence. Our definition for *Logic Control Chart* states that conditions for transitions must be mutually exclusive. If mutual exclusion was not the case sequence would be important. However, if the conditions are mutually exclusive it is not difficult to see that regardless of which order each condition is evaluated only a maximum of one will be true at any point in time. In our case on row 12  $a < 3$  is evaluated to true at this point and we continue

to execute on the self-loop back into  $store_2$ .

It is not difficult for the reader to see that row 13 to 17 plays out the same way as 8 to 12 with updates to variables  $a = 2, b = 2$ . We will continue our justification on row 18 where we see that on the diagram side  $a = 3, b = 3$ . We can see that with this condition in place the edge that should be taken according to the diagram is now changed to  $store_1$ . On line 25 of the execution we see that  $a = a + 1$  updates the variable to  $a = 3$ , and line 26 updates  $b = a$  making  $b = 3$ . With the variable updated we can now take a look at our conditions. We can see that  $a \geq 3$  is now true on line 28 therefore we execute the goto that takes us to *BLUID1* which is on line 12. This occurs on row 22 of our execution trace.

It is not difficult to see at this point that row 22 is identical to row 5 in both modes and values so they are the same in our system. This means that our sequence repeats and line 23,24,25... would be identical to 6,7,8... A full execution trace is now (start,  $a=0, b=0$ ), ( $store_1, a=0, b=0$ ), ( $store_2, a=1, b=1$ ), ( $store_2, a=2, b=2$ ), ( $store_2, a=3, b=3$ ), ( $store_1, a=0, b=0$ ), ( $store_2, a=1, b=1$ ), ( $store_2, a=2, b=2$ ), ( $store_2, a=3, b=3$ ), ...) repeating forever. We note that both execution and the diagram were shown to produce this sequence of modes and values, and by our definition we have shown that this corresponding code execution is correct with respect to the diagram.

By showing all executions for all basic atoms in our system we have shown that our system is correct in execution. All more complex systems can be demonstrated correct in a similar fashion. Because all systems regardless of how complex are just a combination of atoms any system constructed from these same atoms, will also inherently be correct. Thus we can conclude that the execution of this system is then correct by construction with respect to its base atoms.

# Chapter 5

## Hardware

### 5.1 Platform

In Ladder Logic the compiled code runs on an embedded microcontroller in the PLC. Similarly for *Logic Control Chart* we also require such a hardware platform to run our generated code (see Section 4.5.4). The initial hardware platform chosen for this project was the PIC18F452 however this project was designed from the ground up to allow for other hardware to be utilized with some effort from a hardware support team. The PIC18F452 was chosen for this project due to its popularity in the industry and low cost.

In addition the PIC18F452 has the following features that are essential for PLC construction:

- 8 Digital inputs.
- 8 Digital outputs.
- 2 Ports.
- Has support for C syntax.
- Accessible hardware clock.

- On board program memory.

These minimum specification have become the baseline for our hardware requirements. In order to utilize a different chip with our programming software the hardware support team will be required to implement a framework layer as described in Section 5.2.

For real time operation hardware requirements on clock speed will become more critical execution of “goto” should occur as close as possible to constant time or  $O(1)$ . Without the ability to execute “goto” in constant time delays between transitions become increasingly harder to predict and account for.

The specifications for the PIC18F452 are as follows:

- CPU Frequency: 40Mhz
- Program Memory: 32K
- Data Memory: 1536 Bytes
- I/O Ports: 5
- Timers: 4
- 10-Bit A/D: 8 inputs channels

As the reader may observe the PIC18F452 exceeds our minimum requirements for hardware that will run our OS and software. In particular we currently do not include provisions to take inputs from our A/D channels. This was done because the original design had separate I/O modules to handle the inputs and outputs. Given more time on this project it would have been possible to more fully utilize the on board hardware.

## 5.2 Hardware Framework

### 5.2.1 Purpose

The hardware framework allows different chips to be utilized with the same *Intermediate Language* code generated from our software package “PLCEdit”. The hardware must meet the minimum requirements defined by Section 5.1. The framework consists of a bunch of definitions which map the symbolic software references to hardware specific calls. In addition the framework is also responsible for properly initializing the chip and performing any clean up operations once the chip completes execution.

### 5.2.2 Hardware Framework Overview

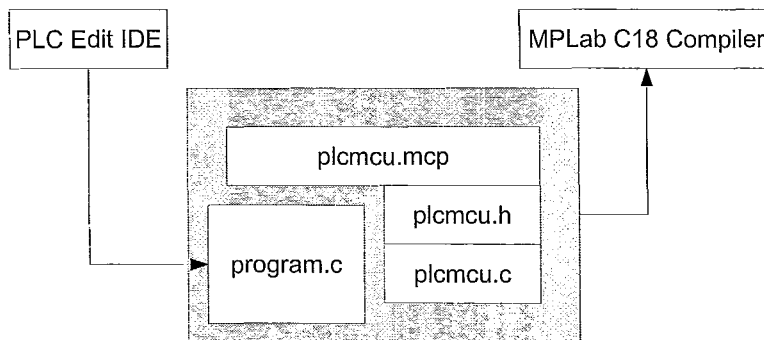


Figure 5.1: Hardware Framework Components

The hardware framework was designed to allow multiple targets. Our IDE generates a `program.c` file that refers to hardware functions and definitions implemented in “`plcmcu.h`” and “`plcmcu.c`” to be combined with an optional “`plcmcu.mcp`” project file. “`plcmcu.h`” contains definitions of the hardware. These are usually designer choices such as oscillator selection, and which ports on the chip they will be using for input and output. “`plcmcu.c`” will implement any function calls that are required and

take care of any chip initializations. In addition it is “plcmcu.c” job to call the user generated “program.c” once the chip is fully initialized and setup. The entire framework is then sent to the MPLab C18 compiler, or a compiler of the implementer’s choosing. In the next section we will show and comment on some of the sample code for our prototype system.

### 5.2.3 Sub-Component Implementations

**File:** plcmcu.h

**Section:** Chip configuration

**Description:** Sets configuration bits for the chip that cannot be done at run time.

**Code:**

```
...
#pragma config OSC = HSPLL //set occilator to HS-PLL
#pragma config OSCS = OFF //disable occilator switch
#pragma config PWRT = OFF //enable power on timer
#pragma config WDT = OFF //disable watchdog timer
#pragma config LVP = OFF //disable low power programming
...
```

**File:** plcmcu.c

**Routine:** init\_chip(void)

**Description:** Initializes any chip specific configuration bits that must be done at run time.

**Code:**

```
...
void init_chip(void)
{
    TRISA = 0xFF; //set all portA to input
    TRISB = 0x00; //set all portB to output
}
```



```
...
```

### 5.2.4 Hardware Specific Definitions

Hardware specific definitions are reserved for mapping specific characteristics of the hardware to match up with the references in the *Intermediate Language*. For example it may be necessary to define the input and output ports to a specific port on the chip itself.

```
...
/* PORT specification */
#define PORTOUT PORTB
#define PORTIN PORTA
...
```

### 5.2.5 Hardware Specific Implementations

Function calls from our software that access more complex operations may be required to be implemented directly into hardware. Our current software only requires that the routine for delaying the execution by a number of milliseconds is implemented. It is done as follows:

**File:**plcmcu.h

```
/* CLOCK specification */
...

#define OCCILATOR 10000000
//our occilator in seconds
#define TCYTIME 1
//now many cycles / instruction 4 for non PLL 1 for PLL
```

```
#define TCYTICK OCCILATOR / TCYTIME
//how long per instruction tick
#define MSTMIME 1000
//milliseconds to seconds

#define MSTICY TCYTICK / MSTMIME
//how many clocks in a ms (10000 for pll)

//crystal select block
#if (MSTICY >= 10000)
    #define DEFDELAY(timevalms) \
        Delay10KTCYx(timevalms * (char) (MSTICY/10000))
#endif
#if (MSTICY >= 1000 && MSTICY void delaysms(int time)
{
    DEFDELAY(time);
}< 10000)
    #define DEFDELAY(timevalms) \
        Delay1KTCYx(timevalms * (char) (MSTICY/1000))
#endif
#if (MSTICY < 1000)
    #error “Unsupported OCCILATOR defined by” \
        “hardware manufacturer please ensure crystal” \
        “is faster than 1Mhz”
#endif

/* End CLOCK specification */

...
```

**File:** plcmcu.c

```
...  
void delayms(int time)  
{  
    DEFDELAY(time);  
}  
...
```

It is highly recommended to avoid macros where possible however the nature of the PIC18F452 required macros in order to prevent inaccurate delays caused by repeatedly calculating the oscillator conversions at run time.

# Chapter 6

## Summary

### 6.1 Conclusions

#### 6.1.1 Conclusions

Throughout this thesis we seek to answer the question on whether the current implementations of ladder logic have been keeping pace with the technologies and training received by today's students. We have discovered that ladder logic although adequate before, have proven to be difficult to use and error prone to a modern trained software engineer. Typically ladder logic programs incorporate a great deal of sequential logic and that can only be faked in ladder logic. This not only adds to unintuitiveness but also adds a strain on hardware since modern microcontroller hardware is inherently sequential in nature.

In constructing *Logic Control Chart* we have shown that a graphical programming language is still possible, and that state machines serves as an excellent metaphor. We have shown that with a properly constructed framework *Logic Control Chart* can be an platform independent solution where multiple hardware can utilize the same *Intermediate Language* code generated by our tool.

## 6.1.2 Summary of Contributions

### 6.1.2.1 Ladder Logic Analysis

Ladder logic has been around for nearly 30 years and it remained the unchallenged defacto standard language in the industry. As such no one has ever asked the question as to whether this rather old system has kept up with modern times. Unsurprisingly this thesis has shown with analysis of ladder logic that there are several shortcomings when compared to a system where we take advantage of modern training programmers received.

### 6.1.2.2 Development of *Logic Control Chart* Language

Part of the thesis looked at current different variants for traditional state machine diagrams. This analysis led to the conclusion that UML2 was best suited to be modified into a language that was precise enough to allow for definitions of behaviours of a program so that it may be directly translated into code without additional programmer intervention.

A significant contribution of this thesis comes in the form of the development of *Logic Control Chart* a new visual programming language based on state machine diagrams. In this thesis we define the syntax, and semantics of the language. The goal achieved here is that we have successfully created a language that is very similar to UML2 with only minor extensions and is able to define precise executable code without the need of a programmer to fill in the gaps.

In the development of *Logic Control Chart* this thesis also demonstrates how correctness may be shown by analysis of structure and demonstrating that execution paths produce the same traces. This work may be useful in the future for any work involving construction of an entirely new visual programming language.

### 6.1.2.3 Development of A Prototype Tool and Hardware Environment

In order to justify the effectiveness of the *Logic Control Chart* language it was necessary to develop a working tool. The tool contains the programmer's IDE, the

Compiler, and the Simulator which are described in detail below.

The main IDE described in section 4.5.2 presented a significant challenge to us. The goal was to use *Logic Control Chart* to describe the operation of our system but also to require no program code to be written after the tool compiled our diagrams. In addition the design goal was to make our tool simple to use and to perform as if it was a vector drawing program. This thesis achieves this design goal and the results are shown in Section 4.5.2 the IDE usage is similar to any drawing program that you are use to and is extendable to incorporate new drawing elements.

The goal of the compiler was to take our raw diagram and to generate executable code without the need of a “programmer in the loop”. This compiler which works in conjunction to the models used in the IDE is a significant contribution to anyone intending on making a language based on state machine diagrams. In construction of the compiler which is detailed in Section 4.5.5 challenges were overcome as to how to take a visual metaphor and preserve all the structure of the diagram into the final output of the compiled code. We believe this presents a contribution to a practical view of how visual programming languages may be possible and practical.

Finally the simulator borrows much from other state diagram based trace systems. It can highlight and animate the state diagram to help the diagram constructor visualize how their states will play out. In addition to aid in understanding how the system will operate the simulator will also list all variables in a watch list so the outcomes of each state can be carefully monitored against their design.

#### 6.1.2.4 Development of A Hardware Platform

Since Programmable Logic Controllers are embedded devices it was also necessary to construct an embedded platform in order to ensure this thesis is not purely theoretical. We have developed a hardware platform based on the PIC18F452 chip and the MPLAB C18 C compiler tool chain.

The details of the construction of the hardware platform and framework are listed in Sections 5.1 and 5.2. It is important to point out at this point that any hardware the implements the hardware framework will be able to utilize PLCEdit and all the

other developed tools. We have designed this hardware platform from the ground up to allow usage of any microcontroller that meets the minimum requirements outlined in Section 5.1. The cross platform nature of this tool-chain is unique to our tool chain as conventional Programmable Logic Controllers stay on a proprietary tool chain and programs must be reconstructed to work on different chips.

## 6.2 Future Research

There was not adequate time during the project to go deeper in to the hardware implementation side of Programmable Logic Controllers. In particular the following features are missing in our design.

- In line programming: at present programming requires that a the microcontroller be taken offline it would be ideal that the controller stay operational in order to decrease downtime. In addition taking the microcontroller off of the board is not ideal and therefore a system for in line programming would be a much improved design.
- Input and output modules: at present I/O are directly connected to the microcontroller's IO channels. This means that I/O is restricted to the chip's design. Buffers should be set up between the chip when higher current loads are required. A more ideal design is to utilize dedicated I/O boards that will plug into the main unit much like how commercial Programmable Logic Controllers operate. The added benefit would be in the ability to replace modules as well as have specialized hardware on the I/O boards to deal with special cases.
- Plug in architecture for PLCEdit: At present adding a new block to PLCEdit requires edits to the source code. Although every effort has been made to make this an easy affair, a more ideal implementation involves a plug-in architecture blocks could become modular and not require any modifications to the main source at all.

- Sub-diagrams: The ability to break one massive diagram up into sub diagrams would be highly beneficial to a more modularized approach to designing programs.

In addition while constructing *Logic Control Chart* and our tool PLCEdit we realized that the methods used here to program microcontrollers are also applicable to full scale applications with the addition of “sub-diagrams”. It is quite possible to create a version of the IDE that is capable of creating programs on standard desktop computers and may be worth exploring. We believe that this thesis serves as a good starting point for future exploration in these areas.

For desktop interactive programs state explosion might become an issue especially in user interfaces. Future research might also look into if incorporating “sub-diagrams” is actually sufficient for creating a large scale program.



# Bibliography

- [1] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.
- [2] A. B. R. Automation, *Rockwell Automation: I/O Adapter Specifications*. Rockwell Automation, 1201 South Second St., Milwaukee, WI 53204-2496, USA.
- [3] A. B. R. Automation, *PLC-5 Selection Guide*. Rockwell Automation, June 2006. Publication 1785-SG001B-EN-P.
- [4] W. Bolton, *Programmable Logic Controllers, Fourth Edition*. Newnes, 2006.
- [5] T. L. Booth, *Sequential Machines and Automata Theory*. John Wiley and Sons, Inc., 1967.
- [6] M. Corp., "Mitsubishi electric automation: Catalog." Used image from product catalog only, 01 2009.
- [7] A. J. Crispin, *Programmable Logic Controllers and Their Engineering Applications*. McGraw-Hill Companies, 1996.
- [8] R. Filer, *Programmable Controllers Using Allen-Bradley Slc 500 and Controllogix*. Englewood Cliffs: Prentice Hall, 2002.
- [9] PlcDev, "How plc's work." Image was used with permission, 01 2009.
- [10] PlcDev, "How plc's work." Image was used with permission, 01 2009.

[11] PlcDev, "How plc's work." Image was used with permission, 01 2009.

[12] PlcDev, "How plc's work." Image was used with permission, 01 2009.

# Appendix A

## Appendix

### Stepper Motor Examples

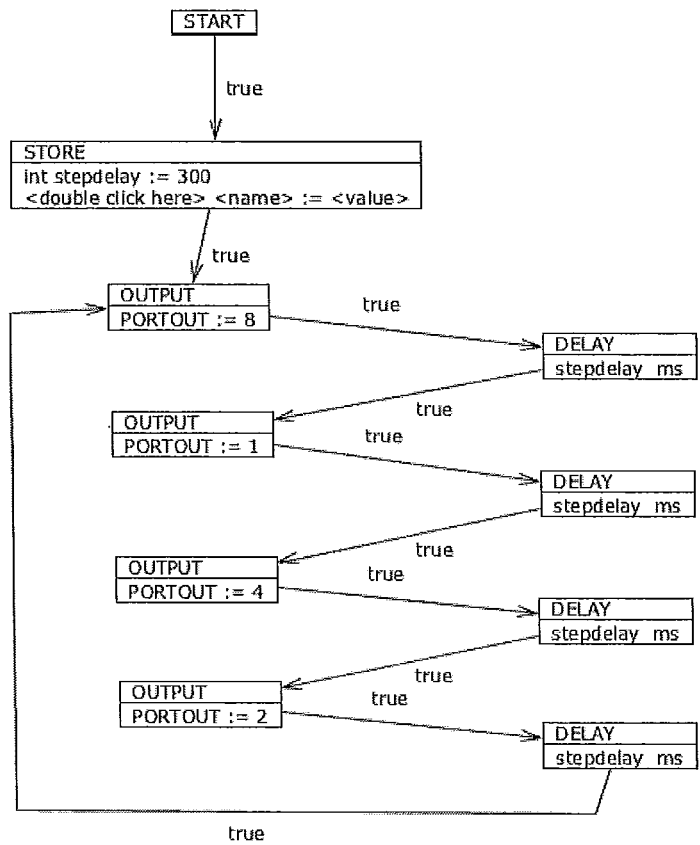


Figure A.1: Running a Stepper Motor with *Logic Control Chart*

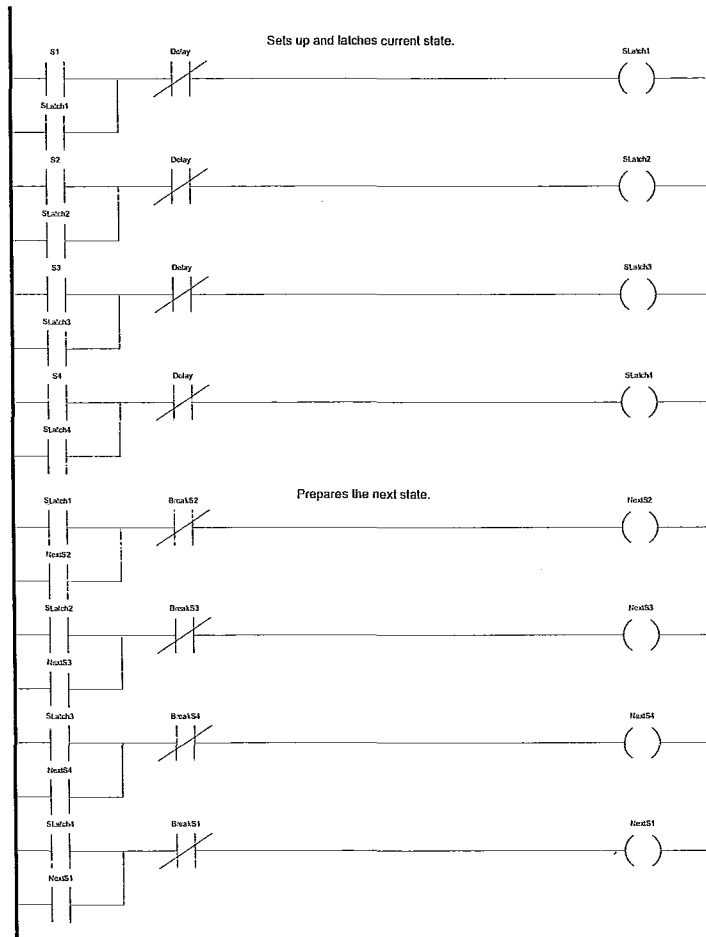


Figure A.2: Running a Stepper Motor with *Ladder Logic* part 1 of 3

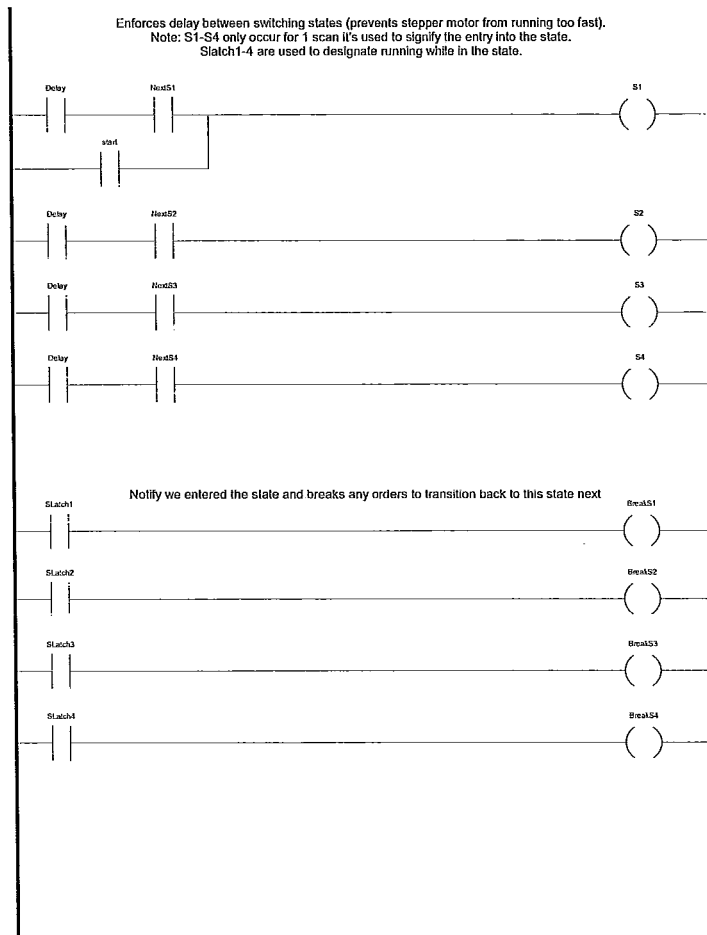


Figure A.3: Running a Stepper Motor with *Ladder Logic* part 2 of 3

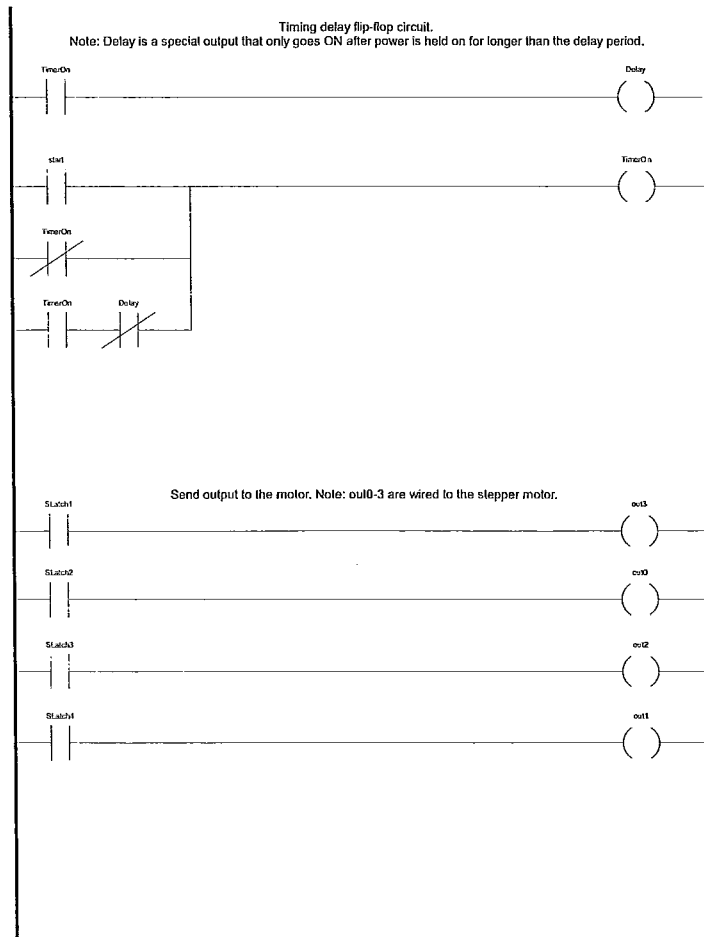


Figure A.4: Running a Stepper Motor with *Ladder Logic* part 3 of 3