Designing an Interpretive Language and Compiler for ASM Charts

Computer Engineering Departmental Honors Project

Wright State University
Winter 2001

Advisor

Dr. Doom

Researchers

Jason Wright
Mike Peterson
Jason Gilder

Table of Contents

1. Overview

The ASML compiler allows one to transform ASM State Charts into a language that can be used with the Lego Mindstorm Robot. Users can utilize all functionality and components of state charts including mealy-based operations. The programs will utilize all components of the Lego Mindstorm, including its motors and sensors. The compiler transforms the ASML language into an LASM (Lego Assembly) program that can be directly downloaded to the robot using the Lego SDK.

1.1 What is an ASM Chart?

ASM stands for Algorithmic State Machine. The entire behavior of any synchronous state machine can be represented by this type of flow chart. Their main components are state boxes and decision boxes.

State Box – Represented by a square box, it contains all of the asserted outputs that a particular machine should execute in that state.

Decision Box – Represented by a diamond, it contains a set of conditions. If those conditions are not met, the flow of execution will go down the left No path; otherwise, it will travel down the right Yes path. These paths are usually represented by the Boolean values of 0 and 1.

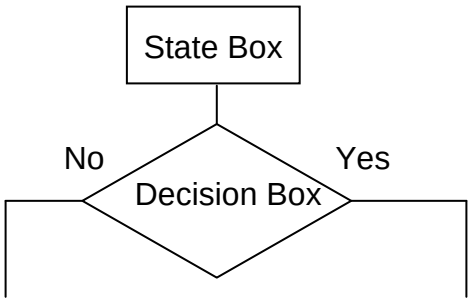


Fig 1.1.1 : Basic Layout of ASM Chart

Execution begins at the first state box and travels downward. The time it takes to execute each state box is called a clock tick. This is a constant value and can be defined in the ASML language. In a Moore-based machine, the decision boxes are considered only at the end of a clock tick. A third structure called a **Mealy Box** can be employed on a branch of a decision box. The Mealy box is a drawn with a circular shape and contains commands (outputs to be asserted) that are

similar to those in a State box. When a Mealy box is present, the machine is said to be of Mealy type. Unlike Moore outputs, Mealy outputs can be asserted at any time during the clock tick, not just at the end. This allows outputs to be asserted instantaneously.

Any state machine can be illustrated with an ASM chart. Here is a simple example of a light. If it has electricity and its switch is on, the light will be on.

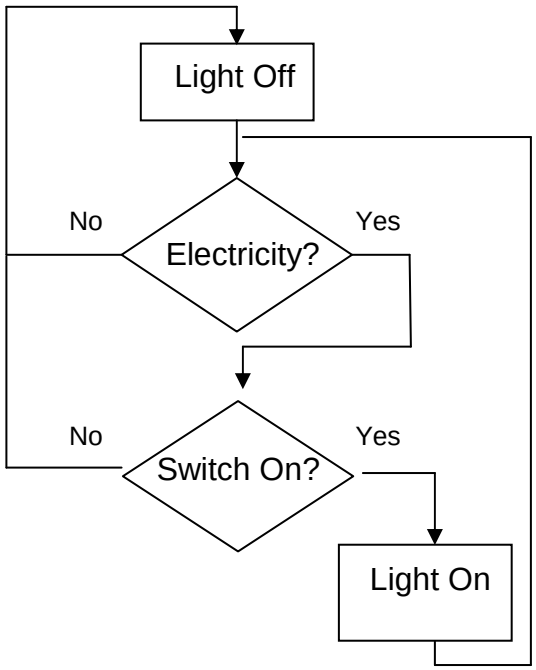


Fig 1.1.2 : ASM Chart of a Light Bulb

The machine starts in the Light Off state. It checks for electricity and for the switch to be on. If either of these conditions fail, the light remains off. If both are true, the machine goes to the Light On state. After the light is on, it returns to the decision boxes to make sure that the conditions are still met. This is a simple example, but it is a good example of how any machine can be expressed as a state machine. ASM charts are useful tools for representing any complicated system because they break it down into its basic parts.

1.2 What is a Lego Mindstorm?

The Mindstorm is a robotic kit developed by MIT for Lego. At its heart is the RCX brick containing a small computer complete with a Hitachi CPU, memory, and an operating system. The RCX brick has three inputs for various sensors including touch, rotation, and light sensors; it also has three outputs for motors. The brick contains firmware in memory to compile Lego Byte Code into its own machine

language. Since it is a Lego product, any number of designs can be created using basic Lego bricks.
The Mindstorm receives its information through an IR port on the RCX brick. One can download programs from a computer to the brick by using a special IR tower. The ASML compiler creates programs that can be downloaded to the robot in this fashion.

For more detailed information as well as links to various Mindstorm sites on the Internet, go to http://www.crynwr.com/lego-robotics/.

1.3 What is ASML?

ASML is the written language equivalent of an ASM chart. The user designs an ASM chart and then types that chart directly into a text file using the ASML language. The format is fairly simple:

State_Box_Name: command1, command2, …, command n, -> Next_State;

Decision_Box_Name: if(decision) then -> 1_Branch else -> 0_Branch;

Mealy_Box_Name~ command1, command2, … , command n, -> Next_State;

By labeling all boxes with a name, the compiler is able to make a working program from an ASM chart.

1.4 ASML Requirements

The ASML compiler was developed using Perl. Perl programs are not compiled executables, so the user must have Perl installed in order to run the compiler. A version of Perl is available on the Internet (ActivePerl at http://www.activestate.com). A specially compiled executable of the ASML compiler may be developed if it is needed.

In order to download an ASML-compiled program to the RCX, the Lego SDK is required. This is also available through Lego on the Internet. While Perl is offered on almost all platforms, the Lego SDK is currently only available for IBM-Compatible systems.

2. The ASML Compiler (User-Level Docs)

2.1 Language Overview (Quick Reference)

The layout of the language is very simple and should follow the ASM state chart exactly. The order of the code does not matter except in the case of the first state. The first state read will always be treated as the **START** state. All other states may be coded in any order.

Overview

State Box	*Name*: *command1,…, ->next state*;
Decision Box	*Name*: **if(***decision***) then** -> *0_state* **else** -> *1_state*;
Decision	*Sensor* *Variable* [*operator*] *value*
Mealy Box	*name~ command1,…, ->next state;*
Variable List (at beginning)	*#var1, var2,…, var n;*
Sensor List (at beginning)	**&***sensor1,…, sensor3;*
Comment (1 per line)	**%** *comment*
Motor Direction and Power	[*Motor Letter*] [*on, off, f , b, (0-7)*]
Play Sound	sound [0-5] 0 : key click sound 1 : beep sound 2 : sweep down 3 : sweep up 4 : error sound 5 : fast sweep up
Assignment	*Variablename* <- *value* *Variablename* <- *Var [operator] value* *Variablename* <- *Var [operator] Var*
Assignment Operators	**+** (addition) **-** (subtraction) */* (division) ***** (multiplication)
Decision Operators	**>** (greater than) **<** (less than) **=** (equal to) **!=** (not equal to)
Line Terminator	**;**
Goto Command	**=>**
Threshold of Light (light sensor;default 50)	**#LIGHT=** *threshold of light*
Threshold of Dark (light sensor; default 30)	**#DARK=** *threshold of dark*
Clock Tick (optional; default is 50)	**#CLOCK=** *clock tick value* (in 1/100 second increments)

Initializations

The top of the program must include a list of sensors and variables used. A clock tick may also be declared here. If no clock tick is defined, the default of 50 ticks (half a second) will be used. In addition, LIGHT and DARK calibration variables can be defined at the top of the program. The default for light is 50(%) and DARK is 40. Declare values for LIGHT, DARK, and CLOCK on separate lines.

Coding State Boxes

A state boxes is represented by its arbitrary name followed by a colon. Any commands inside of the box will follow that colon. A comma must be placed between each command, and a semicolon should be placed at the end of the command. A state box must contain a **goto** statement or else the program will abort. Mealy boxes are slightly different -- their names are ended with a tilde (~) instead of a colon.

Ex. STATE5: Af, Cb, ->STATE7;

This line will make motor A go forward, motor C go backward, and will go to STATE7.

Coding Decision Boxes

Decision boxes also begin with an arbitrary name followed by a squiggly brace set {…}. The expression to be evaluated will be placed between the two braces. The resulting states are written next following a 1 (TRUE) and 0 (FALSE) evaluation. This is done using **goto** statements. As with the state box, commas must separate the goto statements and a semicolon must end the line.

Ex. DECISION9{TIMER<1} 0->TIMERDEC, 1->STATE29;

This line will check the value of TIMER and will go to the TIMERDEC state if it is greater than or equal to 1 and will go to STATE29 if it is less than 1.

Motors and Sensors

Sensors are named by their type and the number or letter of the port they are connected to. Touch sensors return a Boolean 1 or 0. Light sensors return an integer based on reflectance. A larger number means more light is present.

Touch : T[port number] (Ex. T1 would be a touch sensor on port #1)
Light : L[port number]

A **motor** can only be connected to port A, B or C, so it is designated by its port letter followed by its direction. The direction is indicated by a lower-case **f** for

forward or a **b** for backward. A power setting may also follow a motor letter provided it is between 0 and 7.

NOTE: Motors will automatically be turned on when any direction command is issued. This is the same as entering [motor]on.

Ex. Af would be motor A forward
C5 sets motor C to a power level of 5

Calibrating the Light Sensor

Run the Light Calibration program or any program that uses the light sensor. Stop the program and press the VIEW button until the caret mark (^) is below the light sensor. The value currently measured will be displayed. Use this to determine light threshold values.

The LCD Display

The Display will show the integer ID of the current state. These IDs are assigned and displayed during compilation on the PC.

Compiling

The program was written and tested using ActivePerl v5.6 and UNIX Perl. Only these versions can be guaranteed to work. The compiler prompts the user for the names of the input and output files. Most versions of Perl for Windows require one to use a DOS window. Since an error will cause the DOS window to close when double clicking on the ASML icon, it is advisable to use a DOS shell when debugging. Placing the path of Perl in the PATHS= list in AUTOEXEC.BAT will allow you to run Perl from any directory.

Downloading The Program to the Mindstorm Robot

Once an ASML program is compiled correctly, it is ready to be sent to the robot. It uses the ATLClient program which is part of the Lego Mindstorm SDK.

1. Open the ATLClient program by double-clicking its icon.
2. Open your compiled program
3. Select LASM under SOURCE and make sure to uncheck AUTO RUN
4. Place the robot in front of the IR tower and turn it on
5. Click on the OPEN button and then click on the DOWNLOAD button

The program will compile once more and will download directly to the program slot currently active in the RCX robot. Push the RUN button on the RCX to begin the program.

2.2 Initializations

Variables and sensors must be initialized at the beginning of an ASML program. The order of the two lists is not important, but any variable or sensor used in the ASML program must be declared here. The sensor list begins with an ampersand, and commas separate the sensors. As with all lines of ASML code, a semicolon ends the list. The variable list begins with a pound sign(#) and ends with a semicolon; like the sensor list, each variable is separated by a comma. Clock tick and light sensor values must be on their own line.

The format is as follows:

&sensor1,sensor2,…**;**
#variable1,variable2,…**;**
#CLOCK = <*clock tick value*>**;**
#LIGHT = <*light sensor threshold for LIGHT*>**;**
#DARK = <*light sensor threshold for DARK*>**;**

2.2.1 Declaring Variables

#variable1, variable2, … , variable n;

Variables are used to store values and perform mathematical and comparison operations on them. The variable list is entered at the beginning of an ASML program and is preceded by a #. A variable can be any combination of ASCII characters as long as it does not begin with a digit and is not a reserved word.

Reserved Words : INTLOOPCLK, Aon, Aoff, Afloat, Af, Ab, A[0-7], Bon, Boff, Bfloat, Bf, Bb, B[0-7], Con, Coff, Cfloat, Cf, Cb, C[0-7], sound[0-5], LIGHT, DARK

2.2.2 Declaring Sensors

&[type][port], … , sensor3;

Type – T = Touch L = Light

Sensors are also defined at the beginning of an ASML program, and its list is preceded by an **&**. They are defined by a letter denoting type and a number denoting input port. Touch sensors are labeled with a T, and light sensors are labeled with an L. The input port can be 1-3. A touch sensor on input port 2 would be labeled T2 and a light sensor connected to the same port would be

labeled L2. Touch sensors are similar to variables in that they can be referenced later but their value is always a Boolean 1 or 0. Mathematical operations other than comparisons cannot be performed on touch sensor values. Light sensors return an integer based on percentage of reflectance. A larger number means that more light is present. Only one sensor can be connected to each input port.

2.2.3 Light Sensor Thresholds

#LIGHT = *threshold for light;*
#DARK = *threshold for darkness;*

The thresholds for what is considered LIGHT and DARK can be designated at the beginning of the ASML program with the other initializations. These values are added to the variable list, so they can be accessed like any other variable for comparison, assignment, and mathematical operations. If these initializations are not present and a light sensor is present, the compiler will default to 50 for LIGHT and 30 for DARK. Use the light calibration program to determine values.

2.2.4 Clock Tick

#clock = clock tick value;

The clock tick is the amount of time it takes for each state box to execute. It is measured in 1/100 second increments. A clock tick value of 50 would mean that each State box would execute for half of a second before advancing to the next state or decision box. Mealy outputs can be used to make any output assert instantaneously.

Note that this is an emulated clock tick in the LASM language. It is merely a pause at the end of a state. It has no bearing on the actual clock tick of the Lego Mindstorm.

2.3 Coding State Boxes

State**:**command1**,**command2**,** … **,** command n**, ->** next_state**;**

State boxes begin with an arbitrary name followed by a colon. Commands follow and are separated by commas.

A **goto** command (**->**next_state_name) must be present in the command list; it should be the final command in the list.

2.3.1 Motors

[output port][direction, on/off, or power level]

Motors may be connected to the output ports A, B, and C. You can turn motors on or off with [output port]**on** and [output port]**off**. Direction is set with an **f** for forward and a **b** for backward. Depending on the motors' connection and possible gearing, the assumed direction of forward or backward may be switched. Motor speed is set with [output port][power level]. Power levels can range from 0 (slowest) to 7 (fastest).

2.3.2 Sensors

[type][input port]

See 2.2.2 Declaring Sensors and 2.4.1 Checking Sensors

2.3.3 Variables

Variablename <- *value*
Variablename <- *Var [operator] value*
Variablename <- *Var [operator] Var*

Variables are manipulated using the assign **<-** command. A variable can be set to a constant value or another variable's value; in addition, mathematical operation can be performed on it. Operators include addition (**+**), subtraction (**-**), multiplication (*****), and division (***/***). Any combination of variables and constants may be used on the right side of the assign command. All variables listed must have been declared in the initializations.

2.4 Coding Decision Boxes

Decision_name**: if(**decision**) then->** 0-Path_name **else->**1-Path_name**;**

Decision boxes begin with an arbitrary name followed by (decision). A valid decision can be a sensor or a mathematical comparison using variables and/or sensors. Only one decision can be present in each decision box. If more than one decision is needed to determine the next branch, multiple decision boxes can be combined (see 2.4.3 Combining Decisions).

2.4.1 Checking Sensors

(sensor)

To check a sensor, users should simply enter the sensor as the decision. The sensor will be polled and will return a Boolean true (1) or false (0) if it is a touch sensor or an integer if it is a light sensor. This determines which path the program will take.

Touch sensors return a 1 when they are pressed. Light sensors return an integer percentage based on how much light they perceive. They return a larger number when they perceive an increased change in light. Thresholds can be set with **#LIGHT=***value* and **#DARK=***value* at the beginning of the program. These thresholds can later be referenced with comparison operations (eg. if(L2>DARK)then…). Checking a light sensor alone (e.g. if(L1)…) is the same as comparing the sensor to LIGHT (e.g. if(L1>LIGHT)…). Multiple decisions can be used with light sensors to avoid the "gray area" between LIGHT and DARK.

2.4.2 Checking Variables (Comparisons)

{[variable, sensor, or constant] [comparator] [variable, constant, or sensor]}

Any combination of variables, sensors, and constants can be compared. Comparisons include greater than (**>**), less than (**<**), equal to (**=**),
not equal to (**!=**), greater than or equal to (**>=**), and less than or equal to (**<=**). Comparisons return true (1) or false (0) to determine program branches.

2.4.3 Combining Decisions

Since only one decision is allowed per decision box, it is possible to combine multiple decisions by connecting the decision boxes. The first decision box will have the second decision box listed as one of its paths, and there is no limit to the number of consecutive decisions.

NOTE: It is advisable not to have a decision box loop back to itself or to an earlier decision box in the same state. Such a design does not guarantee that the next state will be reached, and the compiler will terminate if it detects such a path.

2.5 Mealy Outputs

MealyName~command1, command2, … , command n, -> Next_State;

Sometimes the user will need to have a sensor react instantly. Mealy outputs are used to accomplish this. Under normal Moore conditions, the outputs are asserted at the end of a clock tick. This means that a sensor could be activated for an entire clock tick before the program goes to the next state. Mealy outputs check the sensors constantly and assert Mealy outputs the moment they are triggered.

Creating a Mealy output is achieved by adding a Mealy box to the ASM chart. Mealy boxes must follow decision boxes. The compiler's design will not allow a Mealy box to go directly to another Mealy box. If you have two Mealy boxes connected, simply add the commands together to form one box.

Unlike Moore outputs that simulate a clock tick by pausing, Mealy outputs must loop through the code a certain number of times to achieve the time associated with the clock tick. This causes a problem with the compiler as an extra delay can occur when looping through a large quantity of code. The more decision boxes you have in a Mealy state, the longer the clock tick will be for that particular state. The program will make adjustments to your clock tick to minimize the delay for the state, but a small delay may be present.

See Appendix 7.1 for examples

3. Main Research Paper

3.1 Initial Research

Obviously, before deciding on how to implement the compiler, we had to choose a programming language. Prior to beginning the project, we were experienced only in the C++ language, but we were concerned with the clumsy string manipulation functions that C++ offers. Luckily, our advisor informally suggested that we look into using the Perl (Programmers' Extraction and Report Language) language. He gave us a brief illustration of the ease at which strings could be compared and manipulated with Perl; as a result, we were faced with a tough decision. Despite having no knowledge of Perl, we decided that it wouldn't take too long to learn most of the syntax of this language. We also welcomed the opportunity to pick up a new and potentially useful programming language during the course of the project. Our decision has worked out extremely well as we have implemented the entire compiler in Perl without very much difficulty. Had we decided to use C++, we would have likely spent many more hours implementing even the most basic tasks.

One must also realize that we had no experience in compiler design prior to this project. Because we wanted to implement and test the basic functions of the project before the more difficult ones, we added new functionality every week. By the time the compiler was finished, we had written a great deal of code without worrying too much about optimization techniques. However, we certainly did appreciate the effort that went into writing the compiler.

3.2 The Text Parser

3.2.1 Introduction

The compiler is logically broken down into two parts: the text parser and the code converter. The first part, the text parser, is responsible for inputting and manipulating the user's program. It breaks up each of the commands and formats them in a way that is convenient for the conversion portion of the compiler.

3.2.2 Input the User's Program

Obviously, we knew from the beginning of the project that the creation of the compiler would be a very daunting task, especially since we had taken no courses on compiler design. The task of converting the user's code into Lego Assembly code seemed simple enough, but there were a variety of minor details that had to be considered. The first major task of the text parser is to open the input file and read all of the lines into an array. Because the user's input file is reasonably small, it is possible to read in all of the lines without worrying about memory limitations of the compiler. As each line is read into the program, the user's comments are instantly removed. A comment is defined as any character between a % character and the end of the line, and Perl's string manipulation operations make it quite easy to remove the comments.

After reading in the user's program lines, the text parser uses the simple string manipulation commands to reformat the information. First, it splits each of the lines on the white space character, \s. It then rejoins them, effectively removing all blank spaces from the program. Because other compilers do not care whether a user puts one space or multiple spaces between words or commands, we decided to design ours in the same way. In addition, by eliminating all white space from the user's program, the parser is able to more effectively perform string comparisons and error checking. After removing the spaces, the parser has to break the user's input lines by semicolons. This does not seem like a daunting task until one considers that a user could spread one line of information over several lines of text. To handle this possibility, the parser uses the Perl *join* command to join all of the user's lines into a single line. Then, the Perl *split* command is used to divide the lengthy line into individual commands that each end with a semicolon. By this stage of the compilation, we are guaranteed to have an array of strings that have no internal white space and that are each ended by a semicolon.

3.2.3 Creating Structures

To complete its major task, the text-parsing portion of the compiler breaks each of the reformatted input lines into its components. Obviously, the user language that we defined utilizes a variety of special characters to designate different types of ASM boxes (see the Quick Reference for the ASML language). For instance, a colon is used to designate a state box, while braces are used to designate the decision box. Because we made these identifier characters unique, it is relatively easy to group together each type of ASM chart input line. As a result, the string parser uses several functions, each of which handles a certain type of user input line. For example, the getStateLines function goes through the input lines and finds each line with a colon; these lines are all grouped into a separate array. This array is used by the createState function to break each of the lines into its components.

Here is an example to illustrate the division that is performed on each line. Let's say that the user inputs the following line to designate a state box:

S1:Aon, Con, Af, Cf, timer<-3, ->S2;

In this case, the text parser creates a Perl structure that is equivalent to a C struct, as shown by the following illustration:

C language

```
 struct state
{
   string name = S1;
   string destination = S2;
   string value[ ] =
        {Aon, Con, Af, Cf, timer<-3}
};
```

Perl language

```
$state
{
   $name = S1;
   $destination = S2;
    @value =
            {Aon, Con, Af, Cf, timer<-3}
 }
```

A similar structure is created for each of the types of boxes, including state, mealy, and decision boxes. In addition, the parser builds the necessary sensor and variable lists. The code conversion portion of the compiler uses these structures extensively to produce the correct Lego Assembly code that is fed to the RCX.

3.2.4 Handling Mealy Boxes

For state and decision boxes, the structures defined above provide adequate means for the later parts of the compiler. However, mealy box code is much more difficult to handle. Mealy outputs can be asserted at any time during the current clock tick. Because of this, the compiler has to be aware of mealy boxes that could be encountered at any time during a given clock tick. In order to ease the code conversion portion of the compiler, the text parser creates structures that list all of the possible mealy boxes that could be encountered in each state. To do this, the parser includes a recursive function called recurse. This function lists all of the possible paths through the ASM chart from the current state to any of the others. The code for this function is very similar to the common recursive binary tree spanning algorithms since it traverses down each ‘zero’ branch of the decision boxes as well as down each ‘one’ branch. Recurse is called each time a new state is encountered, and its output is a global array that collects the traversed names in a meaningful way. The parser then takes the elements of this array to create the TMEALY and TMEALYI arrays that are mentioned below. Basically, they present a relatively clean way for the compiler to determine which mealy boxes need to be monitored during each clock tick.

3.2.5 Error Checking

Obviously, the text parser as described so far works very well in manipulating user's programs that contain no errors. However, as with any programming languages, the users are bound to make syntactical and logical errors as they write their programs in ASML. Therefore, the text parser also handles the majority of the error checking for the compiler.

The Perl language has a very convenient function to gracefully terminate a program. The built-in *die* command allows one to print out an error message and terminate the program. Whenever the string parser encounters an error, it uses the *die* command, and the program ends. Unlike some other sophisticated compilers, the text parser does not create a list of errors; it simply ends the compilation process as soon as the first error is encountered. This approach was certainly easiest to implement, and it provides a very clear way for the user to debug his or her programs.

Because so many different types of errors have to be considered by the text parser, error-checking code is spread throughout the first half of the compiler. Although we had originally planned to try to capture all of the errors in one portion of the code, we soon realized that not all errors could be checked at the same time. The most common types of errors that user will make are syntactical errors (missing semicolons, misuse of a key character, or missing destinations, for example). Therefore, the text parser carefully checks each state line before creating the state structure, each mealy line before creating the mealy structure, etc. The program will terminate if any syntactical error is detected along the way.

Logical errors are more difficult to detect since they are syntactically correct. They include the use of an undefined destination, the duplication of a sensor type, the failure to place all variable and sensor inputs at the top of the program, etc. In addition, the user is further restricted by the rules of ASM charts. For example, all mealy boxes must be descended from decision boxes, rather than state or other mealy boxes. The last few functions of the string parser portion of the code specifically check for these types of errors. Of course, because error checking was not the primary focus of the problem, the string parser does not take some of the most obscure errors into consideration, but it is reasonably thorough. In addition, the parser is very liberal in some areas. For instance, it does not enforce which types of characters are permissible in user-defined names as long as none of the defined key characters are used.

3.3 The Compiler

The compiler works by going through the various structures created in the parse process and translating them to LASM code. Each state, mealy, and decision box has its own block of LASM code beginning with its label, which is its state name, and ending with a jump command which corresponds to its goto command.

For a detailed explanation of the LASM commands mentioned in this section, please reference section 4.

3.3.1 Initializations

The compiler begins with the initialization process. In LASM, all variables are identified by an integer. The compiler is able to bypass this by using a series of **#define** commands at the beginning of the LASM program. This is done by going through the variable array and defining the variable name by its position in the array. This allows the use of the actual variable name rather than an integer in the rest of the program, which makes the program much more readable.

Next, the sensors are initialized. The sensorlist array is used to determine which sensors are connected to each input port. The appropriate type and mode commands are created using the **senm** and **sent** commands. Touch sensors are set up as type Boolean, so when they are checked in a decision box, they return a 1 or 0. Light sensors return a raw value, so the user must select thresholds for what is considered to be light and dark.

A clock tick may also be defined in the initialization phase. It is identified similar to a variable with **#clocktick=value**. The value corresponds to a clock tick in the system (the time it takes to execute each state box). Each number is 1/100 second. If the command is not used, the system will default to the value of 50, which is half of a second.

3.3.2 State Boxes

state_name: command1, command2, … , command n, -> next_state_name;

The state boxes are the first to be compiled into LASM. This is done by going through the statenames structure. The name is first printed with a colon to denote a label so that it may be accessed at any time with a simple jump command.

Next, commands are accessed through the $state{@statenames[$i]}{value}[$j] structure. Commands are split by key letters and symbols to denote an action. Motors are split by A, B, and C, making it is easy to see if an f, b, on, or off follows.

3.3.2.1 Assignments

Assignments are a bit more difficult since there can be many different combinations in their structures:

Variable <- *Variable [operator] constant*
Variable <- *Variable [operator] Variable2*
Variable <- *Variable2*
Variable <- *constant*

The command is split by the **<-** character, and the commands along with the variable name are sent to a subroutine. If there is only one variable or constant on the right side of the assignment, a simple **setv** command is written. The compiler looks at the first character of the variable on the right side of the assign. If it begins with an integer, it is a constant and the appropriate setv arguments are written. Otherwise, the variable is checked against the variable list for error checking purposes before the variable setv arguments are written.

If there is an operation to the right of the assignment, that string must be split by the operator to determine what operation is to be performed. The variable on the left side of the operator must also be checked. If it differs from the variable on the left side of the assign, a setv command must first be written. Then the appropriate mathematical operation is written based on the operator that the expression split on.

Error checking is accomplished by checking the variable against the variable list that is created during the parse process. Constants are determined by looking at their first character since variables cannot begin with an integer. All variables and constants are checked as they are encountered in the assign process.

3.3.2.2 Goto Command

State boxes end with a check of their mealy status with a subroutine call. The subroutine determines whether any of the paths from the current state box go through a Mealy box before reaching the next state box. If there are no Mealy outputs, the compiler simply waits for the clock tick to expire with a **wait** command and then jumps to the next state or decision box with a **jmp** command. Otherwise, the mealy subroutine must be called.

3.3.2.3 The LCD Display

The Mindstorm LCD display can be very useful in reading current information about the system, but it is limited to integers. During compilation of State Boxes, an integer ID is assigned for each state. This ID is displayed on the PC screen along with its state name for reference. The number will be displayed on the Mindstorm LCD screen during execution of the program. This is very beneficial in identifying the current state of the machine and any problems that may occur with the ASM chart. Displaying information on the LCD screen is accomplished with the **disp** command.

3.3.2.4 Decision Boxes

decision_name: if(*decision expression*) then-> zero_path_name
else-> one_path_name;

The decision boxes are the last structures to be created. Their code is considerably smaller than the state box since they act as a simple if statement.
The compiler first looks at the decision expression to determine whether it is a sensor or variable comparison. Determining exactly what components are inside of a decision expression is necessary to create the proper arguments for the **chk** command. Sensors are determined with a quick compare to T1, L1, etc. Variables are determined by comparing the name to the variable list that is created in the text-parsing phase. Constants are determined by looking at the first character in a variable name. Since variables cannot begin with a digit, names that start with an integer are assumed to be constants.

If variable expression is found, several comparison operators split the expression, and the appropriate chk command is written. If a comparison is written, both values are included in the chk command. If a sensor is used, it is compared to a Boolean 1. This type of command is unique in that it not only jumps on a fail but it also can only jump forward in the code. This makes things very difficult since all state box code precedes the decision box code. The solution is to have the chk command jump down to a label inside of the same decision box and then jump to the next state or decision box.

Decision_name:
chkl source1, value 1, relational operator, source 2, value 2, Decision_name-1
jmpl 1->path_name

Decision_name-1:
jmpl 0->path_name

3.3.2.5 Mealy Support

After the normal state box commands have been written, the compiler must determine whether any of the branches from the current State Box contain Mealy outputs. This is done with the subroutine that was discussed above. The subroutine not only returns a 1 or zero but also creates two new structures if Mealy outputs exist. @TMEALY contains the current state name and all of its paths while @TMEALYI contains the index of those structures in their respective arrays. The format is as follows:

@TMEALY = [state box name, decision 1, 0->path, 1->path, decision2, 0->path, 1->path, …,
decisionN, 0->path, 1->path]

@TMEALYI = [empty, decision1 index, 0->path index, 1->path index, decision2 index, …]

If decision 1 is “CheckSensor” then the decision1 position in TMEALYI will contain the index of CheckSensor in the decision structure array. The positions for 0->path and 1->path are only filled if a Mealy output is the destination of the decision box.

Mealy outputs are asserted the instant the decision box input is asserted; therefore, the decision boxes must be checked constantly. This is accomplished by looping repeatedly through the code. The subroutine that creates the Mealy states first initializes a loop counter variable to the value of the clock tick. So if the clock tick is 50, the mealy code will loop 50 times waiting 1/100 second before each loop.

Decision Box Labels

The compiler must determine what code needs to be executed in the loop. This involves writing each decision box inside of the state box code. The labels cannot be the same as their decision name because they will be written later in the normal decision compiling process. Instead, an index is assigned to the current state box name to associate it with a particular decision box. After the Mealy array structures are created, a new array, @mealydec, is created, and all decision boxes in the mealy array structure are placed in the @mealydec array. Only the names are copied. Now the decision boxes are identified by their position in the @mealydec array. For instance, the decision box is array position 5 is now identified as state_box_name-5 inside of the Mealy code.

Now the Mealy code is written. The compiler goes through the @TMEALY structure creating the appropriate decision commands indexed by their position in the @mealydec array. Any goto command in the decision box and Mealy box code must be checked with the @mealydec array. If the destination of any goto command does not exist inside of that array, it will jump to state_box_name-[#of

decision boxes +1]. This last index points to the last block of code in the mealy state that decrements the loop counter and jumps to the next state with the **decvjnl** command. If the loop counter is greater than zero after decrementing, it will jump to the beginning of the Mealy code which begins at label state_name-0.

Mealy Code

Mealy boxes inside of the Mealy code are labeled as follows:

State_name-M-[decision number]-[1 or 0]

A Mealy box coming off of the 1 branch of the decision box in position 2 in state HELLO will be HELLO-M-2-1 and a Mealy box on the 0 branch would be HELLO-M-2-0.

The Mealy code is written in the same way as the State Box code since they contain the same types of commands. The goto command is checked with the @mealydec array to see if the destination is part of the current mealy code. If it is found, the proper jump command is produced. If it is not found, the program will jump to the last index in the mealy code where the loop decrement takes place.

Often, Mealy boxes exist on only one branch of a decision box. If the Mealy box is on the 0 branch, there isn't a problem. The 1 branch jump command will follow the chk command in the current decision. If the single Mealy box is on a 1 branch, another block of code must be produced to handle the problem of chk commands only being able to jump forward. In this case, a label of state_name_M_[decision number] is created with one line of code: this allows for a jump to the next state or decision box.

After all Mealy code has been executed and the loop counter has expired, the code will jump to the first decision box to be executed in a Moore-type fashion. This may seem redundant, but other non-Mealy states may jump to the same decision box at a different time. All decision and Mealy boxes are written again in separate code blocks to be executed at the end of the clock tick. This also allows the program to continue its natural flow through the states.

See Appendix 7.1 for examples.

Mealy Example

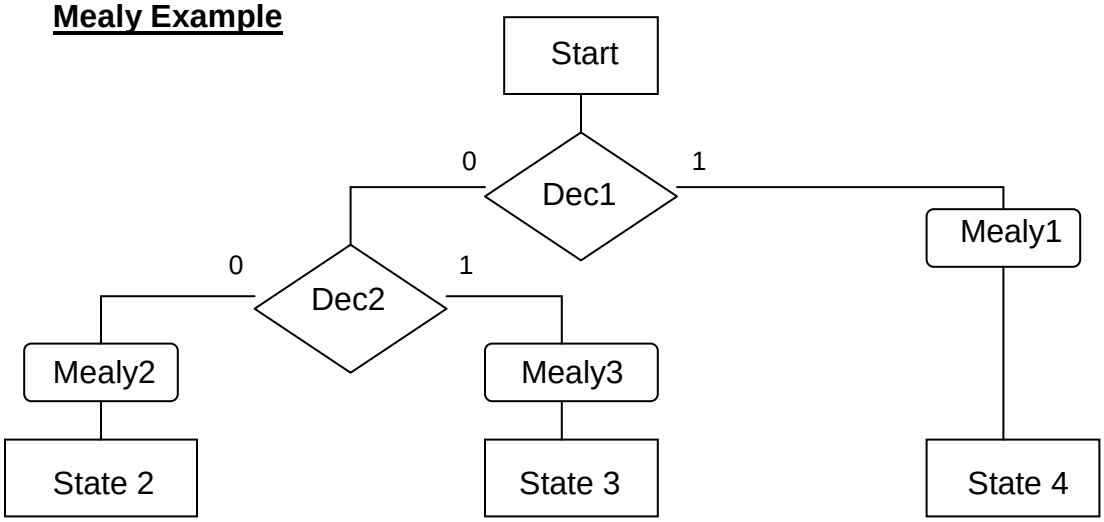


Fig 3.1.1 : Example of a Mealy ASM Chart

```
@TMEALY =  [Start, Dec1, - , Mealy1, Dec2, Mealy2, Mealy3]
@TMEALYI = [  -  ,  2  , - ,   5   ,  3  ,   1   ,   9   ]
@mealydec = [Dec1, Dec2]
```

Note : Index values are arbitrary here. They are totally dependent on the order in which they were entered in the ASML code.

Compiled Code :

Start:
commands
Start_0:
if !(decision box 1) jump to Start_M_0
  else jump to Start_M_0_1
Start_1:
if !(decision box 2) jump to Start_M_1_0
  else jump to Start_M_1_1
Start_2:
decrement loop counter
if (loop counter <= 0) jump to Dec1
   else jump to Start_0

Start_M_0:
jump to Start_1
Start_M_0_1:
Mealy1 commands
jump to Start_2

Start_M_1_0:
Mealy2 commands
jump to Start_2
Start_M_1_1:
Mealy3 commands
jump to Start_2

3.4 Lego Assembly Language

3.4.1 Programming the LEGO RCX

When designing our compiler, we needed to choose a language which our programming language could be compiled into. Because the programming language packaged with the LEGO Mindstorms Robotics Invention System is visually oriented and designed for very young people, it is not very versatile for advanced users. Because of this, many robotics enthusiasts have developed a number of complicated programming languages for use with the LEGO Mindstorms system. We looked at several of these languages as possible candidates to be the target language for our compiler. After examining several possible languages, we settle on NQC and LASM as our top two choices.

Perhaps the most commonly used programming language for the LEGO Mindstorms robots is NQC (Not Quite C), written by David Baum. Because NQC is syntactically similar to the C programming language, it holds a wide appeal among engineers and computer enthusiasts. NQC is very versatile, and it is very well documented. Mr. Baum maintains a web page for NQC at http://www.enteract.com/~dbaum/nqc/index.html. Because of our familiarity with C and the ample, readable documentation available, NQC was an attractive target language for our compiler. However, NQC is a high level language. We intended to have our compiler produce code at a low level in order to maintain as much control over the RCX as possible. Ideally, we were looking for an assembly-level language capable of interfacing with the RCX.

We soon discovered that the LEGO Group provides tools that allow the user to program the RCX directly with byte code commands, or with LEGO assembly (LASM) commands based upon the byte code commands. Additionally, LEGO provides a program called the ATLClient which compiles programs written in LASM and downloads them into the RCX. Because the LASM language is a low-level language, we felt that it would give us the most control over the RCX. Although the documentation for LASM is not very clear, we soon determined that simple programs that involved accessing touch sensors and controlling motors were easy to write, and we were confidant that, with time and effort, we could figure out how to develop more complicated programs with LASM. Additionally, we felt that it would be easier to have our compiler produce LASM programs than NQC programs. With that in mind, we chose to use the LASM language as our target language for our compiler. The ATLClient program and the LASM command documentation can be downloaded from LEGO's Mindstorms web page at http://www.LEGOmindstorms.com/sdk2/.

In LASM, the basic commands to access motors and touch sensors are simple. The **SetPower**[1] command is used to set the power of the motors to a level that can range from 0 to 7. Once the power has been set, the **SetFwdSetRwdRevDir** command is used to set the direction in which the motors turn. The **OnOffFloat** command is used to turn motors on, to turn motors off (so they cannot move), or to cut the power to motors (so that they can coast to a stop). The **SetSensorType** command is used to associate a certain type of sensor with a given sensor input. For touch sensors, the sensor type should be set to switch (boolean mode). For light sensors, the sensor type should be set to reflection. The **SetSensorMode** command is used to set the operating mode and the measurement slop for a given senor. For touch sensors, the sensor mode should be set to Boolean mode. For light sensors, the sensor mode should be set to PctFullScale mode. In both cases, the slope should be set to 0, for absolute measurement. The LEGO documentation is not clear on how these parameters affect the operation of sensors. We determined the appropriate settings through trial and error, and by examining sample LASM code we obtained from LEGO.

The LASM commands that we used to structure our programs are also fairly simple. Tasks are the main components of an LASM program. An LASM program can run multiple tasks concurrently. The main part of the program must be placed in task 0. There can be up to sixteen tasks in an LASM program. Within the LASM program, the start of a task is indicated by placing the text "task #" within the code, where "#" is the number of the given task. The string "endt" indicates the end of that task. When an LASM program runs, task 0 automatically runs first. Other tasks can be started using the **StartTask** command. Tasks can be stopped using the **StopTask** command. It is important to note that while tasks can run concurrently, if multiple tasks try to access the same resource at the same time, such as a variable or a register containing sensor data, the running program will crash. LEGO does not provide protection against this behavior; it is up to the programmer to ensure that multiple tasks do not try to access the same resources.

Within tasks, LASM provides commands that allow us to jump to various points of a program, to pause the program for varying amounts of time, to manipulate integer variables, and to implement if statements that control program flow. The **Ljump** and the **Sjump** commands allow the program to jump to various labels, either forward or backward in direction. Ljump provides longer jumps than **Sjump** at the expense of an extra byte in the program. Our compiler only uses the **Ljump** command, because several compiled programs that we have produced provided jumps that were too long for the **Sjump** command to handle. The **Wait** command allows a program to be paused anywhere from 10 ms to

[1] In the discussion of LASM programming, the names of LASM commands appear in bold. These should not be confused with the syntax of the actual LASM language. The given names are the names of the commands as they appear in LEGO's documentation. For more detailed information on these commands, see 3.4.4 and 3.4.5 for an LASM command summary and parameters.

several minutes. This command is useful for implementing uniform pauses throughout a program, to model a clock tick between state changes. The **SetVar** command allows us to set one of the sixteen variables with an integer value. There are many other commands available to manipulate variables. These commands are explained in the appendix. Perhaps the most useful program control command is **LCheckDo**. This command performs a logical comparison on two values, and if the comparison fails, the program jumps to a given label, otherwise, the program continues execution immediately after the **LCheckDo** command. This command allows the program to poll the current state of a sensor, or to compare the values of two variables, and then take various actions based upon the result. It thus provides the functionality of an if statement within the program. The limitation of the **LCheckDo** command is that it can only jump forward in direction when a comparison fails. This shortcoming can be overcome by jumping to an **Ljump** command, which can then jump backwards. Another useful program control command is **SdecVarJumpTZero**. It decrements a given variable. If the variable reaches zero after being decremented, the program jumps to a given label. Like **LCheckDo**, this jump can only occur in the forward direction. Within an LASM program, **SdecVarJumpTZero** is best used to implement loops that only loop a set number of times.

There are several other commands that are useful in modeling ASM charts in LASM programs. The **PlaySystemSound** command will play one of six preset sounds on the RCX's speaker. This can be used to output an audible notice that a certain point in the program has been reached. For instance, this command, in conjunction with **LCheckDo**, could be used to play a simple beep every time a touch sensor is pressed. Along the same lines, the **PlayTone** command can be used to play a sound, but the user can set the frequency and duration of the sound with this command. The **ViewSourceValue** command can be used to display the value of a constant or a variable on the RCX's display. This command is used to display the number of the current state within an ASM chart based program. This is very useful for debugging improper robot behavior caused by programming errors.

3.4.2 Implementing Mealy Functionality

The above commands provide all of the necessary structure needed to implement a program that models an ASM chart with Moore outputs. However, finding an easy method to implement an ASM chart with Mealy outputs in the LASM language was more difficult. Because a Mealy output can be asserted in the middle of a clock tick, it was necessary to find a way to detect conditions that assert a Mealy output as soon as those conditions occur. Our first approach to implement Mealy functionality in LASM programs was to use the **SEnterEventCheck** command. This command waits for a given event to occur, and then, when the event occurs, the program jumps to a given label. An example of an event occurrence could be the touching of a touch sensor. The LASM syntax for this command is: "**mone** event source, event value, label or

jump distance." However, there is no explanation on how to properly set an event using the event source and value in LEGO's LASM documentation. For an event based upon a touch sensor, we tried setting the event source to source 9, the parameter source for a sensor value, and then setting the event value to either 0 or 1. This approach did not work, nor did several other approaches that we tried. With inadequate documentation on the use of this command, we tried to find examples or instructions on the use of the **SEnterEventCheck** command on the internet. I found an excellent web page[2] that documents the LASM commands and their byte codes, but the **SEnterEventCheck** command, and all other commands relating to events, was omitted. After further searching, we was unable to find any examples or documentation on the use of events within LASM programs. Our next approach was to write a program in Not Quite C that we hoped would contain a monitor command. The RCX Command Center, a program that compiles NQC programs and transmits them to the RCX, contains a feature that allows the user to view the bytecode of NQC programs. The bytecode commands translate directly into LASM commands. After reviewing the bytecode of the NQC programs we wrote, it became apparent that David Baum avoided using event monitoring in his programming language. With no clear documentation and no adequate examples of the use of the **SEnterEventCheck** command, it became necessary to find another way to implement Mealy functionality into LASM programs.

Our next approach to implementing Mealy functionality was to run a background task that continuously checked the state of the sensors. If a sensor changed states, the background task would assert the Mealy outputs. Often, asserting Mealy outputs means changing the states of the motors. It was quickly discovered that if two tasks accessed the motors, the program would crash. Thus, this approach did not work.

Our final approach was to use very short pauses within a loop in each state, so that conditions that lead to a Mealy output could be frequently monitored. As discussed previously, the **Wait** command can be used to implement set-length pauses into an LASM program. If the transitions between states within an ASM chart based program occurred once every second, the length of the state can be implemented by using "wait 2,100[3]" within an LASM program. Using "wait 2,1" results in a pause of one one-hundredth of a second. For a clock tick that lasts 1 second, we reasoned that running a loop that pauses for one-hundredth of a second one hundred times, and checking for Mealy assertion conditions in each iteration of the loop, we could implement Mealy-like functionality into LASM programs. This approach worked, but all of the overhead caused by checking sensor states in every iteration of the loop slowed down the programs. We determined that a Mealy state took approximately twice as long to run as a state

[2] This Op-code reference can be accessed at:
http://graphics.stanford.edu/~kekoa/rcx/opcodes.html#Numerical .

[3] The 2 within this command indicates that 100 is a constant value. See the LASM reference appendices for more detail.

that does not implement Mealy functionality. Thus, to correct timing delays, we reduce the number of loop iterations in a Mealy state by half. For instance, if the clock tick for the program is one second, a Mealy state will loop 50 times in stead of 100. Depending on the number of possible Mealy outputs and the complexity of the conditions that lead to their assertion, it is impossible to determine a formula that exactly corrects the timing to match a non-Mealy state. Because this project is intended to be an educational tool, and not a commercial application where timing must be exact, this minor inaccuracy can be tolerated. It should be noted, however, that the inaccuracy will become more noticeable the longer the clock tick is. The above method is the simplest way to implement Mealy functionality into an LASM program, so it is the method that we adopted into our compiler.

3.4.3 Implementing Variable-length Pausing

It is desired to allow a user to be able to pause a running program, and then unpause the program so it can continue running. Though this seemed like an easy task, it immediately became apparent that pausing an unpausing a running program is difficult to implement in LASM. The first attempt to implement pausing involved sending a command directly to the RCX from a computer to set a given variable to 1. The LASM program would continuously monitor this variable in a loop, similar in manner to the way Mealy functionality is implemented. Then the program noticed that the variable was set, it would jump to a section of code that turned the motors off. It would then enter an infinite loop that would wait for the variable to be reset to 0. When the variable was reset, the program would jump back to the original code. We used the ATLClient program to send the SetVar command that sets the variable. Unfortunately, sending this command crashes the running program for some unknown reason.

LASM provides a method to send infrared messages between RCX bricks using the **SendPBMessage** command. We hoped that we could modify the above approach to use IR messages instead of setting a variable. Unfortunately, the **SendPBMessage** command can only be called from inside a running program; it can't be called remotely by the ATLClient and sent to the RCX. A similar command, **InternMessage**, sets the IR message buffer within the RCX to a given value, and it can be called from the ATLClient. However, sending this command also crashes any running program. We determined that sending any LASM command to the RCX while a program is running causes a program to crash. Because of this, there is no clear way to implement pausing into an LASM program.

The next approach was to write a program in NQC that implements pausing. If successful, the underlying program could be decompiled and then examined in the hopes that a similar program could be written in LASM. Mike e-mailed David Baum asking for advice on how to implement a pause in NQC. His advice to Mike was to write a program that watches the IR buffer waiting for a message.

When the message arrives, a subroutine is called that turns the motors off and waits for another message. When the message arrives, the subroutine will restore the motors to their original states, and then return to the place within the program where the subroutine was called. Mike successfully wrote this program in NQC and, using the RCX Command Center program, he could send messages to the RCX, and the program paused and unpaused appropriately. Unfortunately, he was still unable to send the messages from the ATLClient without crashing the running program. Since it is not feasible to use the RCX Command Center every time a pause is desired, we have not yet implemented variable-pausing into the LASM programs. However, LEGO sells a remote control that is capable of sending messages to the RCX. We have ordered the remote in the hopes that we will be able to implement pausing using it and the commands matching the bytecodes from the disassembled NQC program that Mike wrote. Based upon this approach, we hope to implement pausing functionality in the future.

3.4.4 LASM Command Summary

Motor manipulation commands

SetPower (0x14)
-Set Motor Status Registers with new power levels
syntax: **pwr** motor list, power source, power value
ex: pwr 7,2,7 => sets motors A,B,C to power 7 (highest)

SetFwdSetRwdRevDir (0xE1)
-Set Motor Status Registers with direction settings
syntax: **dir** direction, motor list
ex: dir 1,5 => set motors A,C to reverse

OnOffFloat (0x21)
-Set motor status register with motor values to on, off, or float
syntax: **out** onofffloat, motor list
ex: out 2,6 => turn on motors A,B

Sensor manipulation commands

SetSensorType (0x32)
-Set given sensor to given type
syntax: **sent** sensor number, sensor type
ex: sent 0,1 => set sensor 0 to Boolean type

SetSensorMode (0x42)
-Sets given mode & measurement slope for the given sensor, also resets the processed sensor value
syntax: **senm** sensor number, sensor mode, slope
ex: senm 1,1,0 => set sensor 1 to boolean mode, absolute measurement

Task Manipulation commands

StartTask (0x71)
-this (re)starts a given task if the task slot is not empty
syntax: **start** task number
ex: start 3 => starts fourth task (0 is the first)

StopTask (0x81)
-Stops the given task, releases its access resources
syntax: **stop** task number
ex: stop 3 => stops fourth task

Program Control Manipulation Commands

-**Ljump** (0x72)
-Jump a certain distance, either forward or backward
Syntax: jmpl label or offset from start of command (use labels)
ex: **jmpl** monitor => jumps to label called monitor
(Note: we can just use the command jmp, jmpl is for long jumps and takes more space to code)

Sjump (0x27) p40
-Same as Ljump, except use **jmp** instead of jmpl, shorter jumps but takes less space

SdecVarJumpTZero (0xF2) p62
-Decrements the variable & jumps if the variable is negative afterward. It does not rely on internal state information, and since it uses ordinary variables, it also allows direct access to the loop counter for conditional termination of loops.
syntax: **decvjn** variable #, label or jump distance (use label)
ex: decvjn 3, gohere => decrement var 3, if it becomes negative, jump to label gohere

SCheckDo (0x85) p93
-Does a comparison, and if it fails (evalutates to FALSE), execution jumps to given label. This can facilitate efficient implementation of switch/case statements because a failed match can jump forward to the next check. Note: the label must be a forward label.
syntax: **chk** src1, val1, relop, src2, forward label or jump distance
ex: chk 2,0,2,9,0,gohere => check if sensor 0 is equal to value 0, if not, go to label gohere.

LCheckDo(0x95) p94
-This is the same as SCheckDo, except it can do longer jumps but takes more space. The command is **chkl**

Wait (0x43) p67
-Suspends task execution for the given number of 10 ms
syntax: **wait** wait source, wait value
ex: wait 2,100 =>wait 1 second

Variable Manipulation Commands

SetVar (0x14) p75
-Set given variable with given number
Syntax: **setv** variable #, source, value
ex: setv 5,2,3 => initalize variable 5 to value 3

SumVar (0x24) p76
-adds the given # to the given variable
syntax: **sumv** variable #, source, value
ex: sumv 0,2,4 => add 4 to var 0

SubVar (0x34) p77
-subtract given # from given value
syntax: **subv** variable #, source, value
ex: subv 1,2,6 => subtract 6 from var 1

DivVar(0x44) p78
-Divides given Variable with given number; if zero, the command is ignored
syntax: **divv** variable #, source, value
ex: divv 13,2,4 => divide var 13 by 4

MulVar (0x54) p79
-multiply given var with given num
syntax: **mulv** variable #, source, value
ex: mulv 10,2,3 => multiply var 10 by 3

SgnVar (0x64) p80
-Sets given variable with the sign of the given number (-1,0,or+1)
syntax: **sgnv** variable #, source, value
ex: sgnv 2,2,-14 => set var 2 to –1, the sign of –14

AbsVar (0x74) p81
-sets given var with absolute value of given #
syntax: **absv** variable #, source, value
ex: absv 23,2,-13 => set var 23 with 13

AndVar(0x84) p82
-bit-wise AND on given var with given #
syntax: **andv** variable #, source, value
ex: andv 12,2,4 => bit-wise AND var 12 with 4

OrVar (0x94) p83
-bit-wise OR on given var with given #
syntax: **orv** variable #, source, value
ex: orv 16,2,3 => bit-wise OR var 16 with 3

Additional Useful Commands

PlayTone (0x23)
-Queues tone into RCX sound buffer, if enabled; buffer will play it
syntax: **playt** frequency, duration (in tenths of a second)
ex: playt 440, 10 => play frequency 440 for 1 second

PlaySystemSound (0x51)
-Play given system sound, if sounds are globally enabled
syntax: **plays** sound (sounds: 0-key click sound; 1-beep sound; 2-sweep down; 3-sweep up; 4-error sound; 5-fast sweep up)
ex: plays 4 => play error sound

ViewSourceValue (0xE5) p97
-display the given source with the given value on the RCX display. The display can show values ranging form –9999 to +9999
syntax: **disp** precision, display source, display value (precision is the decimal precision and ranges from 0 to 3
ex: disp 0,2,1 => display 1 as an integer

3.4.5 LASM Parameters for Lego RCX

motor list: The 3 least significant bits in a byte are for motors A,B,C in order of C, B, A
ex: 5 = 101 => use motors A,C

power source: use 2 => constant source
*NOTE: using 2 in a source parameter always indicates a constant source

power value: ranges from 0 to 7, 7 is the highest

direction: backwards(0), reverse (flip current direction) (1), or forwards(2)

sensor number: can be 0, 1, or 2 (left to right on the RCX)
sensor type : (recommendation: use 1 for touch sensors, 3 for light sensors)

- 0 = no sensor (raw mode)
- 1 = switch (boolean mode)
- 2 = temperature
- 3 = reflection (% full scale mode)
- 4 = angle (angle steps mode)

slope: 0 = absolute measurement; 1-31 = dynamic measurement(slope)

sensor mode: (recommendation: use 1 for touch sensors, 4 for light sensors)

- 0 = raw mode
- 1 = boolean mode
- 2 = transition count mode
- 3 = period counter mode
- 4 = % full scale mode
- 5 = Celsius mode
- 6 = Fahrenheit mode
- 7 = Angle steps mode

onofffloat: 0 = float; 1 = off; 2 = on

task number: ranges from 0 to 9

variable number: there are 32 global variables available, use 0-31 (variable source is source #0)

relop:

- 0 = greater than
- 1 = less than
- 2 = equal to
- 3 = different from

4.0 Things We Learned

We began with virtually no knowledge of the Lego Mindstorm, it's programming language, and Perl. We came a long way in a relatively short period of time. There were many obstacles and we still haven't surpassed them all, but we have accomplished quite a bit.

We had to first find a language to compile our new language into. We wanted a language that would be as low-level as possible. Byte code is the machine language of the Mindstorm so we started with that. There is very little documentation on byte code so we were forced to rely on the Lego byte code manual found in the Lego SDK (www.mindstorm.com). This is where we learned that for each byte code command, there is a LASM counterpart. Programming in LASM allows us to use labels and variables to make a translation more easily. Plus, we do not have to worry about memory addresses.

Deciding on a language to write our compiler was also a concern. We have used C/C++ our entire college career so that was the obvious choice. However, we learned about Perl from Dr. Doom and found it to be much easier.

Our biggest obstacle was understanding the LASM language. Lego's manual was spotty at best offering only a brief explanation of each command. Samples could only be found with the Scout SDK, a smaller version of the Mindstorm, and there were only five of them. There was a lot of trial and error before we could get a command to work. We resorted to some reverse engineering using another high-level language, Not Quite C. Not Quite C is a C-based compiler for the Mindstorm that has a "show code" button. This shows a rough quasi-LASM representation of the program with byte code. We used this byte code to determine the parameters for the LASM commands.

Designing a clock tick for a system that had none became a challenge. Execution of LASM code is almost instantaneous so we had to figure out a way to have the system work in a sequential manner with an appropriate timing method. We decided to pause at the end of each state which would keep the outputs asserted but wouldn't perform any decision box operations. Mealy operation was the most difficult because all decision boxes must be checked during the clock tick. We solved this by placing all state box and decision box code together in one large indexed block. The program would then pause for 1/100 second before looping through the code again. This gave a pause while also checking all decision boxes. It is a bit of a hack, but it is about the best we could do in an assembly language environment.

Not everything we did was successful. We tried using monitors, which are interrupts, to handle sensor detection. This would allow the program to monitor a sensor and jump to location at the instant the sensor was pressed, regardless of where it was in the execution of the program. This led us to our problem of

handling a pause function using a remote. We wanted to be able to step through an ASM chart program using the remote. The LASM program would monitor the remote via the IR port and would go to a pause state when a message was received. The challenge of understanding IR messages and how to handle them is also present. With some work, we may be able to implement this in the future.

Working in a group worked out well. The project was much too large for one person to tackle, especially at the beginning. As the project went on, we separated into our own niche. Jason Wright worked on the text parser, Jason Gilder took on the compiler, and Mike Peterson discovered how all of the LASM commands and parameters worked. We still were in contact often because the all of the parts relied on each other. There was great interaction in the group. We were able to work out a lot of the larger problems by breaking the problem into pieces that each person could work on.

5.0 Future Work

Remote control functionality to send messages to the RCX. This will allow the user to step through their ASM chart program.

Communication between two RCX bricks; sending and receiving messages in the form of asserted outputs and IR sensor input.

Support for more sensors (temperature, rotation, etc.)

GUI Interface with built-in text editor

Automated compilation and download routine

6.0 History

6.1 Project Proposal

The main goal of our project is to create a language and corresponding compiler for ASM charts that model the behavior of Lego Mindstorm Robots. As an early estimation, this project will be two quarters in length and will count toward the departmental honors project requirements in Computer Engineering. We will register the first quarter of the project as a CEG 499-04 independent study course and as a CEG 498 design clinic the second quarter.

Timeline & Responsibility

The following is a tentative schedule and division of responsibility for our first quarter of this project as well as long-term goals for the entire length of the project. A schedule for the second quarter of the project will be presented by the end of the first quarter. We may add additional goals to our project based on the progress we make.

Weeks 1-2
Individually: we will each research the background of the Lego Mindstorm system in order to understand the inputs, outputs, sensors, limitations and other basic operations. Additionally, we will review ASM charts and state machine concepts.
As a group: we will meet to discuss our research and teach each other what we have learned.

Weeks 3-4
Individually: we will each create five ASM charts of various levels of complexity that model possible functions of an individual Lego robot based upon capabilities involving three sensor inputs and three motor outputs.
As a group: we will go over each other's ASM charts.

Weeks 5-10?
As a group: from the ASM charts that we create, we will collectively decide on a syntax for a programming language that will model these charts for Lego robots.
Individually: For each of the ASM charts that we created earlier, we will write a program in the language that we create. This will allow us to test the robustness of the language and decide how we will proceed in the design of the language.

First quarter (time permitting) & winter break
Individually: we will study compiler design concepts and source code of current Lego compilers for various languages.

Second quarter goals:

1. Develop a compiler that translates programs written in our language into Botcode understood by Lego robots. These programs will be for the function of an individual robot.

2. Add functionality to the language and compiler to allow infrared communication between two robots so that they may function concurrently, time permitting.

6.2 First Quarter Report

Introduction

This report summarizes our progress made during this quarter on our senior honors research project involving coding Asynchronous State Machine diagrams into assembly code that runs on Lego's Mindstorm robot system. Included in this report is a partial description of the Lego assembly language and a manual for our ASM programming language. It also includes our goals for the concluding quarter of this project.

Background

The Lego MindStorm system is a robotic kit designed to introduce children of all ages to basic programming and robotic design. The core of the MindStorm system is a device called the RCX. The RCX contains an 8-bit Hitachi processor complete with its own memory and operating system. It is basically a small-scale computer. Although designed for children, the architecture of the RCX has encouraged many advanced engineers and enthusiasts to develop their own methods and languages for programming the MindStorm system. The RCX contains firmware that acts as a bridge between the software designed for the MindStorm system on the PC and the machine code specific to the Hitachi processor in the RCX. Thus, we are able to write programs in an assembly-level language defined by The Lego Group that is much simpler to understand and implement than the Hitachi machine code.

The CEG-360 Digital System Design class taught at Wright State University covers a topic known as Asynchronous State Machine diagrams. These diagrams are used to logically map the functionality of a digital system in which all outputs can be mapped from all combinations of inputs. They are drawn as simple flow charts; an example of an ASM chart is attached to the back of this report. For our project, we are creating a user-friendly programming language that converts an ASM chart into a program that can run on the Lego MindStorm System. This will allow students in Dr. Doom's CEG-360 classes to see the digital systems that they design run in a Lego robot.

In order to meet this goal, we are designing a new programming language and writing a program that compiles this language into the Lego Assembly Language. The major difficulties in this project involve understanding the Lego

Assembly Language and writing a robust compiler. This compiler must be capable of not only generating files in Lego Assembly but also in detecting errors within the programs that are being compiled.

First Quarter Progress

We spent the first couple of weeks learning about the capabilities of the RCX as well as reviewing the concepts from our previous CEG 360 course. We consulted a variety of books, web sites, and experts and we learned several different languages that people have created to communicate with the RCX. After this research, we had a general idea of the resources available to us and we had a better understanding of the scope of the project. We understood how the RCX motors and the sensors worked and we used other available languages to create simple programs for the robot. By the end of the third week, we knew what the Lego Mindstorm was capable of and at this point, we were able to begin designing our programming language.

After creating several sample ASM charts for various possible robot designs, we developed the syntax for our programming language. The most recent manual for this language is attached to this report. We converted some of our ASM charts into our language to make sure that it could handle a variety of possibilities, and once we were satisfied, we began the two major tasks for the project: designing the compiler and finding a low-level method to communicate with and program the robot.

Our major obstacle of the first quarter was learning the Lego Assembly Language (LASM) that we intend to use with our compiler. Because most Lego enthusiasts program the RCX with a higher level language, we found virtually no documentation or examples of the LASM language. We were able to find one vague manual as well as some examples of a similar language implemented on a different Lego product. Because of the way that parameters are handled, this language was very difficult to learn and we used trial-and-error techniques for several weeks. Near the end of the quarter, we were able to make some significant progress after performing some reverse engineering. Luckily, we found another programming language called Not Quite C that could be compiled into the visible hexadecimal byte code that the RCX recognizes. We were able to translate these byte codes into recognizable LASM commands. By the end of the first quarter, we deciphered most of the necessary commands that we will need to write a basic program from one of our ASM charts (See the attached LASM description). We still need to master some of the more difficult LASM commands, but we feel that we are on the right track.

The other major task was writing the compiler. We originally planned to write the compiler in C since we were all familiar with it. Since we knew that this would heavily involve string parsing, Dr. Doom suggested using the Perl programming language. After some introductory research, we realized the benefits of using Perl and decided that we would learn it for this project. We also thought that it would be a useful language to know for the future. We have completed preliminary functions for the file reading, string parsing, and state box creation.

Second Quarter Plan

Once the second quarter begins, each of us will take on a separate responsibility. We will start with Mike testing the LASM code to determine the best way for us to implement the commands in our project. He has already deciphered a good portion of the language and will now focus on the decision box commands. He will be working closely with Jason Gilder to determine the best way to use the compiler commands. Jason Wright will continue working with Perl. He will first complete the error checking functions of the text parser. By the time he is finished, he will be able to deliver three structures that are ready for compilation: a variable list, a state box hash, and a decision box hash. The error checking function will display the argument that caused the error, an error message, and the name of the line where the error was found. Time permitting, Mike and Jason W. will also begin work on the RCX display and a possible remote control feature. Jason Gilder will continue to work on the state box compiler. He will add more error checking functions and robust functionality to the state box compilation. Once this has been completed, he will begin working on the decision box compiler. This will be one of the most difficult parts of the project so it should require the help of Jason W. and Mike.

There are several enhancements that we would like to add if we have the time. A routine to display the current state on RCX's LCD display would help students see what their program is doing. We would also like to add remote control functionality to be able to step through the program using the remote to start and stop the program. When we have finished the compiler, we would like to create a nice front-end to write and compile the code. Perl works well for creating HTML code so we may use that feature to create our GUI and even to publish it on the web. We will have to look into finding or writing a program that will send our compiled code directly to the RCX. There are programs available that we have been using but it will add another unnecessary step for the students to complete.

COMPILER DESIGN

Our compiler is still in its infant stages, but we have developed a general plan for its overall design. The main program will contain the code that actually translates the users' code into valid LASM commands. We will divide this code into several major functions, including one for the state box manipulation and one for the decision box manipulation. Most of the code for the background tasks will be included in a separate header file. This file will contain several small functions for the string parsing and the error checking. By breaking the code up this way, we will make sure that our code does not become too long and complicated. Also, it will allow each of us to be responsible for one section of code without having to worry too much about what the others are doing.

DELIVERABLES

At the end of next quarter, we will deliver copies of all of the work we have done, including sample ASM charts, programs written in our own language, the

final manuals for our language and the LASM commands, the Perl code for our compiler, all of our weekly progress reports, as well as any other materials we feel are necessary. Our project will conclude with a demonstration of our compiler and of programs written in our language running on the Lego robots.

7.0 Appendix

7.1 Examples from ASM Chart to LASM Code

7.1.1 The Wall-Hugger : A simple robot that stays close to the first wall it encounters. It could be used for some rudimentary maze navigating. This robot has motors on A and C and touch sensors on 1 and 2 on the front-left and front-right respectively.

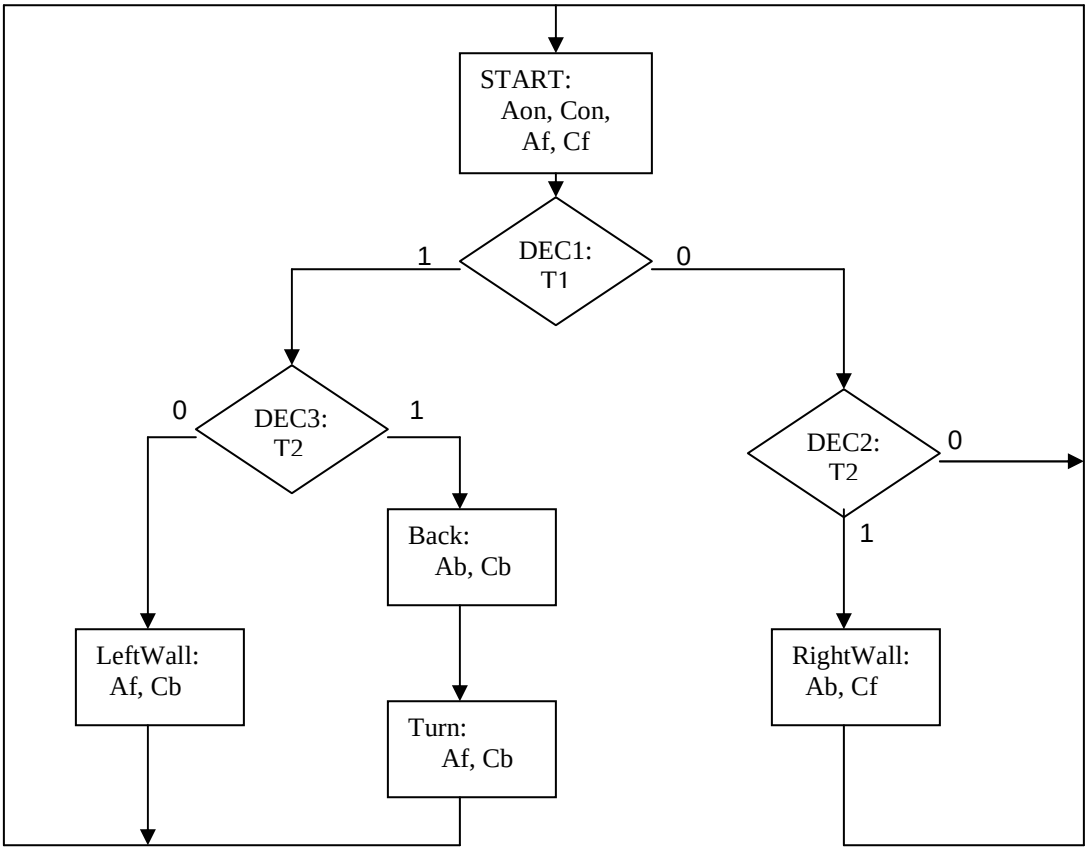


Fig 7.1.1 : Wall-Hugger ASM Chart

Wall Hugger ASML Code

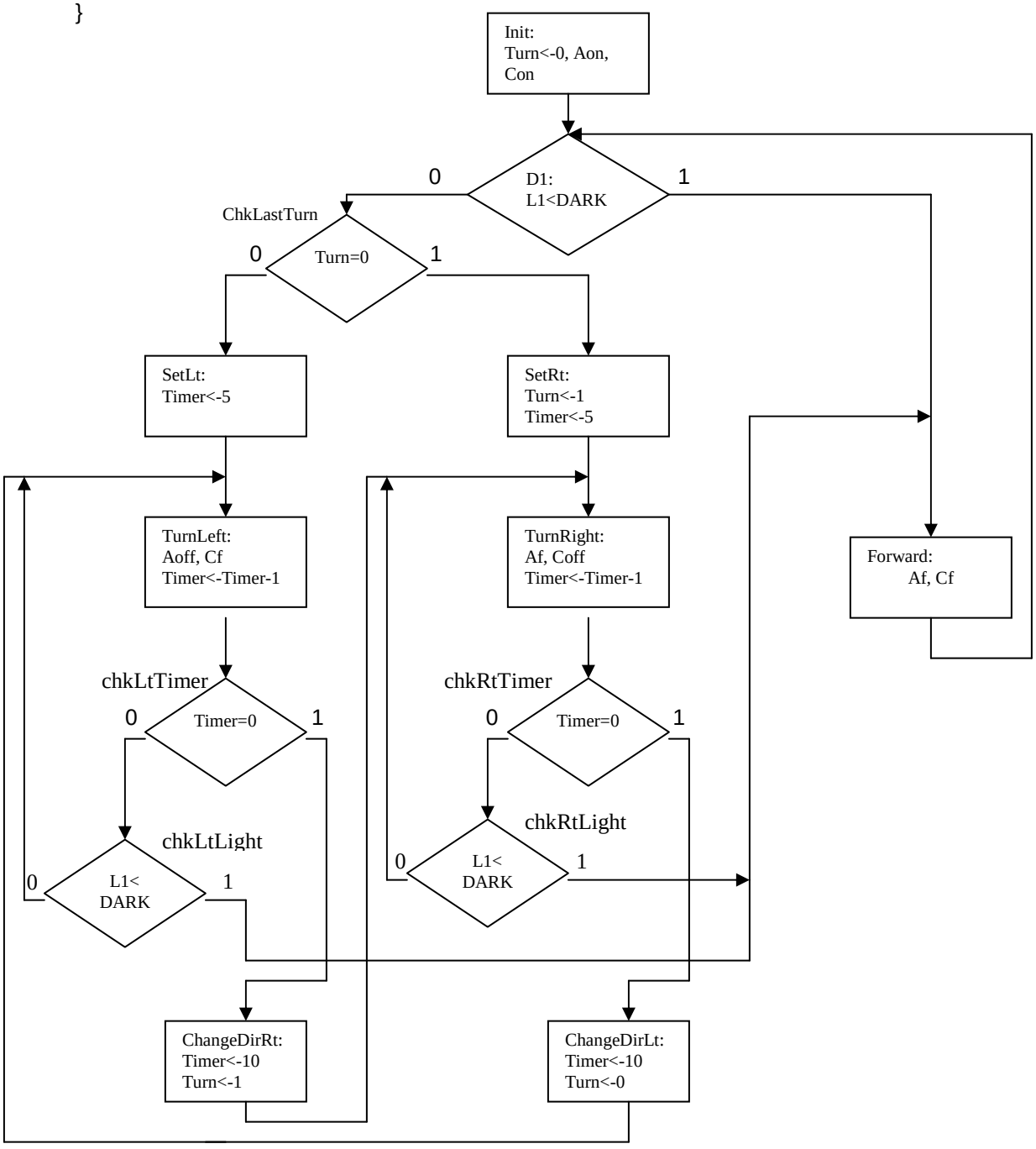
```
% set up touch sensors on 1 and 2 and a clock tick of 100 (1 second)
&T1,T2;
#clock=100;
START: Aon, Con, Af, Cf, -> DEC1;
DEC1: if(T1) then -> DEC3 else -> DEC2;
DEC2: if(T2) then -> RightWall else -> START;
DEC3: if(T2) then -> Back else -> LeftWall;
Back: Ab, Cb, -> Turn;
Turn: Af, Cb, -> START;
LeftWall: Af, Cb, -> START;
RightWall: Ab, Cf, -> START;
```

Wall Hugger Compiled Code

```
// c:\perl\lego\LASM-P1.txt - Created using ASML for the Lego Mindstorm.
//          Developed at WSU (c) 2001 MJJ Inc.
task 0
#define  INTLOOPCLK  0

// -- Sensor Definitions --
sent 0,1
senm 0,1,0
sent 1,1
senm 1,1,0

// -- State Boxes --
START:
disp 0,2,1
out 2,4
out 2,1
dir 2,4
dir 2,1
wait 2,100
jmpl DEC1

Back:
disp 0,2,2
dir 0,4
dir 0,1
wait 2,100
jmpl Turn

Turn:
disp 0,2,3
dir 2,4
dir 0,1
wait 2,100
jmpl START

LeftWall:
disp 0,2,4
dir 2,4
dir 0,1
wait 2,100
jmpl START

RightWall:
disp 0,2,5
dir 0,4
dir 2,1
wait 2,100
jmpl START

// -- Decision Boxes --
DEC1:
chkl 2,1,2,9,0,DEC1_1
jmpl DEC3
DEC1_1:
jmpl DEC2

DEC2:
chkl 2,1,2,9,1,DEC2_1
jmpl RightWall
DEC2_1:
jmpl START

DEC3:
chkl 2,1,2,9,1,DEC3_1
jmpl Back
DEC3_1:
jmpl LeftWall
endt
```

7.1.2 <u>The Line Follower</u> – When you realize that the Mindstorm has a light sensor, the line follower is the first invention that comes to mind. Here is a design that uses 1 light sensor. The starting point can be a problem. This works by starting the robot on the line or to the left of the line. Timer values may also need to be tweaked depending on the clock tick value.

If I'm on the line, go straight forward
If I'm off the line {
  Turn back the opposite direction from the way I turned last time
  If I still don't find the line, turn farther back in the other direction
}



Line Follower ASML Code

Here we used a clock tick of 5 for more precise movement. Therefore we had to change the Timer value to 20 for a small turn and 40 for a large turn. We use the variable Turn to remember the direction we turned last. Zero means that we turned left last and 1 means we turned right last. The program begins with the value of Turn being 0 so it is advisable to start the robot on the line or on the left side of the line.

```
% Line Follower Robot with 1 Light Sensor

% Turn = direction last turned (0=Left 1=Right)
% Timer = time to turn

&L2;
#Turn,Timer;
#LIGHT=50;
#DARK=40;
#CLOCK=5;

Init: Turn<-0, ->D1;
D1: if(L2<DARK) then->Forward else->chkLastTurn;
Forward: Af, Cf, ->D1;
chkLastTurn: if(Turn=0) then->setLt else->setRt;

setLt: Timer<-20, ->TurnLeft;
TurnLeft: Aoff, Cf, Timer<-Timer-1,->chkLtTmr;
chkLtTmr: if(Timer=0) then->changeDirRt else->chkLtLight;
chkLtLight: if(L2<DARK) then->Forward else->TurnLeft;
changeDirRt: Timer<-40, Turn<-1, ->TurnRight;

setRt: Turn<-1, Timer<-20, ->TurnRight;
TurnRight:  Af, Coff, Timer<-Timer-1, -> chkRtTimer;
chkRtTimer: if(Timer=0) then->changeDirLt else -> chkRtLight;
chkRtLight: if(L2<DARK) then->Forward else-> TurnRight;
changeDirLt: Timer<-40, Turn <-0, -> TurnLeft;
```

Compiled Line Follower Code

```
// c:\perl\lego\LASM-P1.txt - Created using ASML for the Lego Mindstorm.
//          Developed at WSU (c) 2001 MJJ Inc.

task 0

// -- Variable Definitions --
#define  Turn  0
#define  Timer  1
#define  LIGHT  2
#define  DARK  3
#define  INTLOOPCLK  4

// -- Sensor Definitions --
sent 1,3
senm 1,4,0

// -- Initial Values for LIGHT and DARK --
setv LIGHT,2,50
setv DARK,2,40

// -- State Boxes --
Init:
disp 0,2,1
setv Turn,2,0
wait 2,5
jmpl D1

Forward:
disp 0,2,2
out 2,1
dir 2,1
out 2,4
dir 2,4
wait 2,5
jmpl D1

setLt:
disp 0,2,3
setv Timer,2,20
wait 2,5
jmpl TurnLeft

TurnLeft:
disp 0,2,4
out 1,1
out 2,4
dir 2,4
subv Timer,2,1
wait 2,5
jmpl chkLtTmr

changeDirRt:
disp 0,2,5
setv Timer,2,40
setv Turn,2,1
wait 2,5
jmpl TurnRight

setRt:
disp 0,2,6
setv Turn,2,1
setv Timer,2,20
wait 2,5
jmpl TurnRight

TurnRight:
disp 0,2,7
out 2,1
dir 2,1
out 1,4
subv Timer,2,1
wait 2,5
jmpl chkRtTimer

changeDirLt:
disp 0,2,8
setv Timer,2,40
setv Turn,2,0
wait 2,5
jmpl TurnLeft

// -- Decision Boxes --
D1:
chkl 9,1,1,0,DARK,D1_1
jmpl Forward
D1_1:
jmpl chkLastTurn

chkLastTurn:
chkl 0,Turn,2,2,0,chkLastTurn_1
jmpl setLt
chkLastTurn_1:
jmpl setRt

chkLtTmr:
chkl 0,Timer,2,2,0,chkLtTmr_1
jmpl changeDirRt
chkLtTmr_1:
jmpl chkLtLight

chkLtLight:
chkl 9,1,1,0,DARK,chkLtLight_1
jmpl Forward
chkLtLight_1:
jmpl TurnLeft

chkRtTimer:
chkl 0,Timer,2,2,0,chkRtTimer_1
jmpl changeDirLt
chkRtTimer_1:
jmpl chkRtLight

chkRtLight:
chkl 9,1,1,0,DARK,chkRtLight_1
jmpl Forward
chkRtLight_1:
jmpl TurnRight

endt
```

Line Follower with 2 Light Sensors – Adding a second light sensor will make the line follower much more accurate. Simply place the second light sensor next to the first.

Left Light Sensor-L1	Right Light Sensor-L2	Left Motor-A	Right Motor-C	Result
Dark	Dark	Forward	Forward	Go forward
Dark	Light	Stop	Forward	Turn left
Light	Dark	Forward	Stop	Turn right
Light	Light	-	-	No change

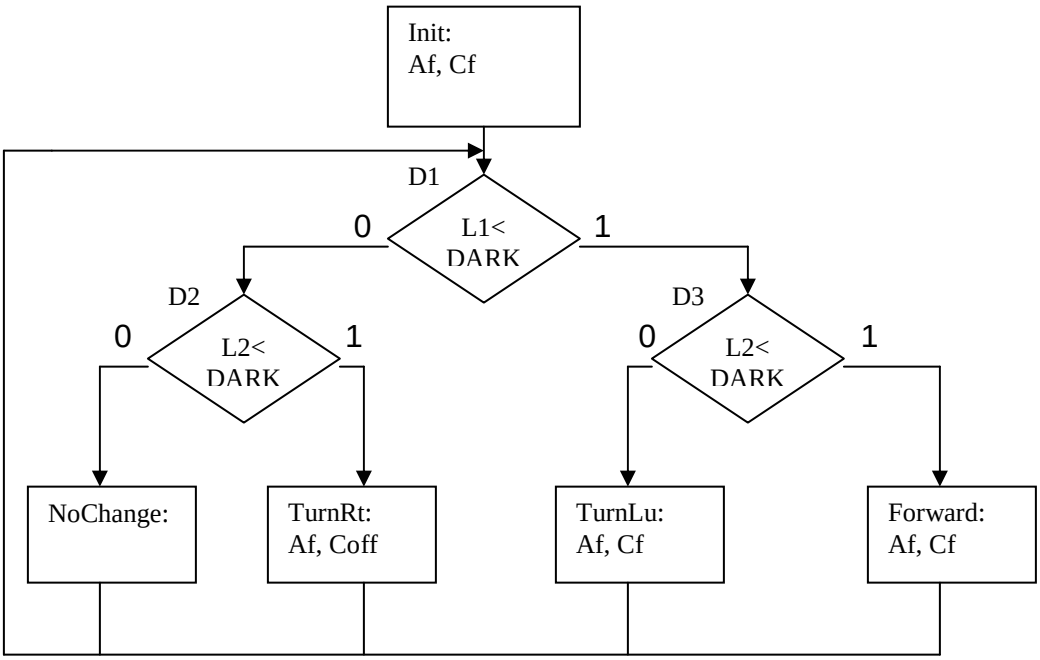


Fig 7.1.3 : ASM Chart for Line Follower with 2 Sensors

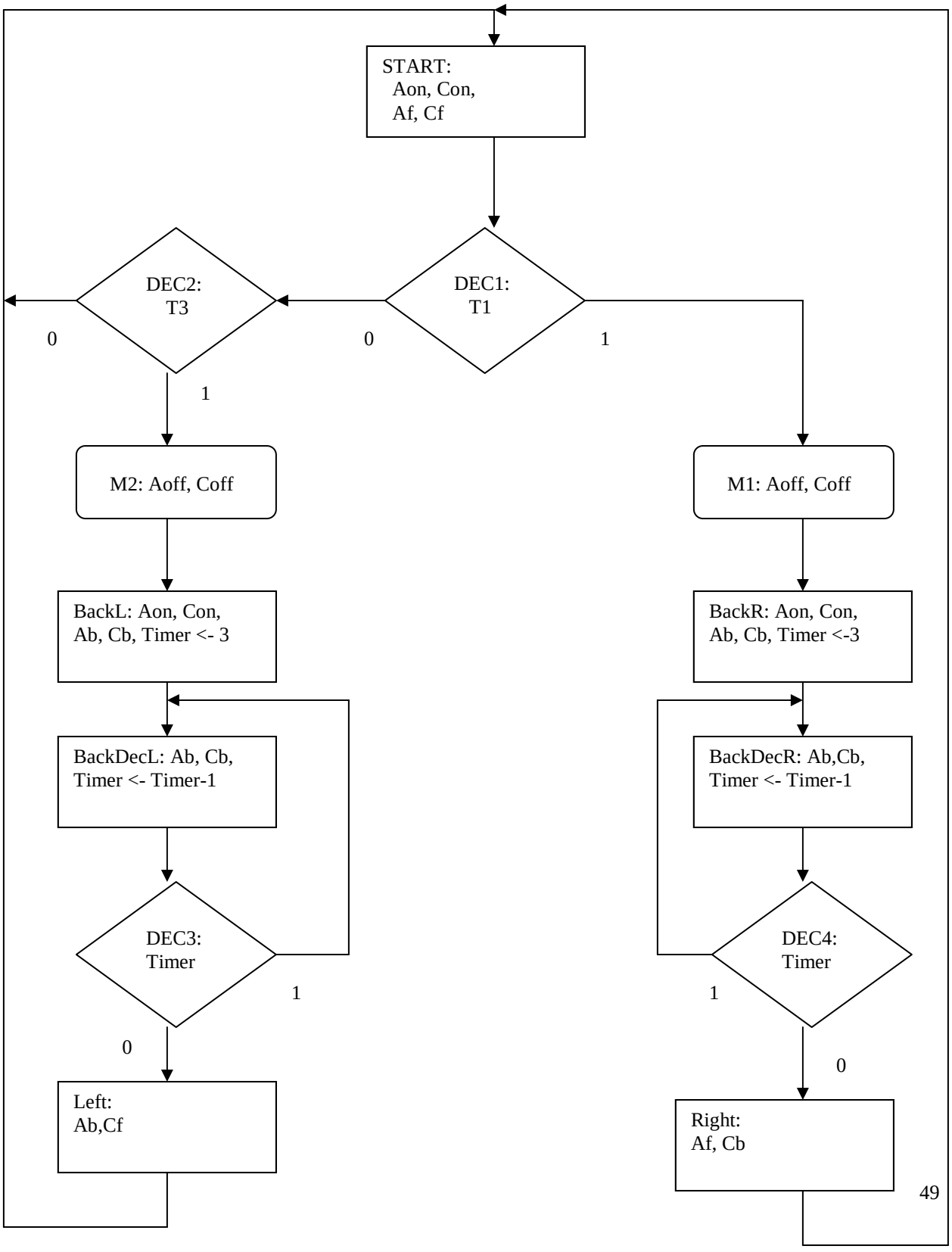
ASML Code for Line Follower with 2 Light Sensors

```
% Line Follower - 2 Light Sensors
&L1,L2;
#LIGHT=50;
#DARK=40;
#clock 5;
Init: Aon, Con, -> D1;
D1: if(L1<DARK) then->D2 else-> D3;
D2: if(L2<DARK) then->Forward else->TurnLt;
D3: if(L2<DARK) then->TurnRt else->NoChange;
Forward: Af,Cf, ->D1;
TurnLt: Aoff,Cf, ->D1;
TurnRt: Af,Coff, ->D1;
NoChange:->D1;
```

Line Follower with 2 Sensors Compiled Code

```
task 0
// -- Variable Definitions --
#define  clock5  0
#define  LIGHT  1
#define  DARK  2

// -- Sensor Definitions --
sent 0,3
senm 0,4,0
sent 1,3
senm 1,4,0
// -- Initial Values for LIGHT and DARK --
setv LIGHT,2,50
setv DARK,2,40
// -- State Boxes --
Init:
disp 0,2,1
out 2,1
dir 2,1
out 2,4
dir 2,4
wait 2,50
jmpl D1

Forward:
disp 0,2,2
out 2,1
dir 2,1
out 2,4
dir 2,4
wait 2,50
jmpl D1

TurnLt:
disp 0,2,3
out 1,1
out 2,4
dir 2,4
wait 2,50
jmpl D1

TurnRt:
disp 0,2,4
out 2,1
dir 2,1
out 1,4
wait 2,50
jmpl D1
// -- Decision Boxes --
D1:
chkl 9,0,1,0,DARK,D1_1
jmpl D2
D1_1:
jmpl D3

D2:
chkl 9,1,1,0,DARK,D2_1
jmpl Forward
D2_1:
jmpl TurnLt

D3:
chkl 9,1,1,0,DARK,D3_1
jmpl TurnRt
D3_1:
jmpl D1
endt
```

7.1.3 <u>The Two-Sensor Mealy BumpBot</u> – This robot is built with two tank treads. There is a bumper with two touch sensors on the front of the robot. The algorithm that the robot follows is simple: If there is nothing in front of me, go forward. If there is an obstacle to my left, stop, go backward, and then turn right. If there is an obstacle to my left, stop, go backward, and then turn left.



ASML Code for BumpBot

```
% set up touch sensors on 1 & 2 and a clock tick of 50 (.5 seconds)
&T1,T3;
#CLOCK=50;
#Timer;
START: Aon, Con, Af, Cf, ->DEC1;
DEC1: if (T1) then -> M1 else -> DEC2;
DEC2: if (T3) then -> M2 else -> START;
M1~ Aoff,Coff, ->BackR;
BackR: Aon, Con, Ab, Cb, Timer<-3, -> BackDecR;
BackDecR: Ab, Cb, Timer<-Timer-1, -> DEC4;
DEC4: if(Timer>0) then -> BackDecR else -> Right;
Right: Af,Cb,->START;
M2~ Aoff,Coff, ->BackL;
BackL: Aon, Con, Ab, Cb, Timer<-3, -> BackDecL;
BackDecL: Ab, Cb, Timer<-Timer-1, -> DEC3;
DEC3: if(Timer>0) then -> BackDecL else -> Left;
Left: Ab, Cf, -> START;
```

Compiled Code for BumpBot

```
// bumpbot.mjj - Created using ASML for the Lego Mindstorm.
//          Developed at WSU (c) 2001 MJJ Inc.

task 0

// -- Variable Definitions --
#define  Timer  0
#define  INTLOOPCLK  1

// -- Sensor Definitions --
sent 0,1
senm 0,1,0
sent 2,1
senm 2,1,0

// -- State Boxes --
START:
disp 0,2,1
out 2,1
out 2,4
out 2,1
dir 2,1
out 2,4
dir 2,4
//  -- Mealy Portion of START --
setv INTLOOPCLK,2,20
START_0:
chkl 2,1,2,9,0,START_M_0
jmpl START_M_0_1
START_1:
chkl 2,1,2,9,2,START_M_1
jmpl START_M_1_1
START_2:
wait 2,1
decvjnl INTLOOPCLK,DEC1
jmpl START_0

START_M_0:
jmpl START_1

START_M_0_1:
out 1,1
out 1,4
jmpl START_2

START_M_1:
jmpl START_2

START_M_1_1:
out 1,1
out 1,4
jmpl START_2

// -- End Mealy Box START --

BackR:
disp 0,2,2
out 2,1
out 2,4
out 2,1
dir 0,1
out 2,4
dir 0,4
setv Timer,2,3
wait 2,50
jmpl BackDecR

BackDecR:
disp 0,2,3
out 2,1
dir 0,1
```

```
out 2,4
dir 0,4
subv Timer,2,1
wait 2,50
jmpl DEC4

Right:
disp 0,2,4
out 2,1
dir 2,1
out 2,4
dir 0,4
wait 2,50
jmpl START

BackL:
disp 0,2,5
out 2,1
out 2,4
out 2,1
dir 0,1
out 2,4
dir 0,4
setv Timer,2,3
wait 2,50
jmpl BackDecL

BackDecL:
disp 0,2,6
out 2,1
dir 0,1
out 2,4
dir 0,4
subv Timer,2,1
wait 2,50
jmpl DEC3

Left:
disp 0,2,7
out 2,1
dir 0,1
out 2,4
dir 2,4
wait 2,50
jmpl START

// -- Mealy Boxes (Moore Execution) --
M1:
out 1,1
out 1,4
jmpl BackR

M2:
out 1,1
out 1,4
jmpl BackL

// -- Decision Boxes --
DEC1:
chkl 2,1,2,9,0,DEC1_1
jmpl M1
DEC1_1:
jmpl DEC2

DEC2:
chkl 2,1,2,9,2,DEC2_1
jmpl M2
DEC2_1:
jmpl START

DEC4:
chkl 0,Timer,0,2,0,DEC4_1
jmpl BackDecR
DEC4_1:
jmpl Right

DEC3:
chkl 0,Timer,0,2,0,DEC3_1
jmpl BackDecL
DEC3_1:
jmpl Left

endt
```