

A Master's Plan B Project: Automatic Generation of Source Code from Ladder Diagrams

Grant Edwards

January 11, 2000

Abstract

This paper describes a system for the automatic generation of program source code from a graphical representation of system requirements. These are in the form of ladder diagrams that are used to document the Boolean-logic based programming present in many industrial automation applications. These diagrams are descended from electrical schematics of relay panels and are used as a means of programming microprocessor-based programmable logic controllers. The code generation system described consists of two parts: a graphical ladder diagram editor and a post-processor that generates program source code.

1 Introduction

Programmable logic controllers (PLCs) used to control automated machinery are often programmed using ladder diagrams [Krigman]. These diagrams provide a widely understood visual representation of system requirements. This project demonstrates a set of tools that allow such ladder diagrams to be used to automatically generate program source code in a high level language such as C. These tools consist primarily of a graphical ladder diagram editor running under Unix/X11 and a source-code generation program that runs on any platform that supports ANSI C.

While ladder diagrams may be used to program a PLC there are applications when it is not practical to use a PLC. Instead, custom designed microprocessor-based hardware may be required. The tools discussed in this paper allow source code targeted for such hardware to be generated

from a familiar method of requirements specification.

An introduction to and brief history of ladder diagrams is presented in Section 2 along with examples of how these ladder diagrams have evolved and are used today. We will see examples of situations where a PLC and ladder diagrams might be applied but due to application constraints, a purchased PLC isn't suitable.

After the objectives of the project are discussed in Section 3, Section 4 will describe the architecture of the project. The features and design of the ladder diagram editor will be outlined in Section 5, and the code generator will be described in Section 6.

2 History of Ladder Diagrams

If you have ever watched an automated manufacturing process (metal stamping, plastic molding, etc.) you saw the synchronized and sequenced operation of tens, hundreds, or thousands of separate mechanisms. Electric and hydraulic motors start and stop on cue to move conveyor systems. Hydraulic and pneumatic cylinders actuate in the proper sequence to press mold halves together and then inject molten plastic. Switch contacts built into safety covers automatically shut down dangerous portions of the system when covers are opened. After every 5000 parts, an empty cardboard carton replaces a full one which is sealed and labeled.

Such systems can involve many hundreds of inputs: mechanical limit switches, proximity switches, safety interlock switches, temperature sensors, weigh scales, liquid level sensors, pressure switches, control panel switches, and so on. That system might also have hundreds of outputs: solenoids, hydraulic or pneumatic cylin-

ders, electrical motors (stepper or continuous), heating elements, control panel indicators, audible alarms, etc. For the purposes of discussing ladder diagrams, all of them are Boolean devices – they are either on or off.

In computer science terminology, the system can be described as either simple combinatorial logic or as a finite state machine, depending on where you draw the line between internal state and external inputs/outputs. Indeed, most such systems built within the last decade or two are controlled by a computer of some type – though possibly just a 8-bit processor with a few thousand bytes of code space and a few hundred bytes of RAM. But, such complex machines were running quite happily before the microprocessor, before the transistor, before the computer, and even before the vacuum tube.

One primary technique that was (and occasionally still is) used to control such systems is relay logic. This technique involves wiring relay contacts and switch inputs in series/parallel combinations between a power supply and an output device to implement Boolean expressions. Various types of mechanically latching relays, counters, time-delay relays, and other electro-mechanical devices can be connected for additional functionality.

The schematics showing the series/parallel combination of relay and switch contacts are known as “ladder diagrams” because when large sets of them are drawn side-by-side with a common power line across the top they somewhat resemble a ladder laying on it’s side. (They don’t look that much like ladders, but probably as much as drawings of “tree” data structures resemble real trees.)

Recently, the International Electrotechnical Commission has issued a standard that covers the programming of PLCs [IEC 1131-3]. This standard includes specifications for ladder diagrams – unfortunately the standard only specifies how ladder diagrams are to appear when drawn with ASCII characters and does not address the issue of representing ladder diagrams on bit-mapped displays or printers with graphics capabilities.

2.1 Ladder Diagram Examples

The standard schematic symbol for a set of relay contacts is a pair of parallel lines perpendicular to the direction of current flow. If you are familiar with electrical schematics, this is also the symbol for a capacitor. Whether the device is a capacitor or a pair of relay contacts is determined

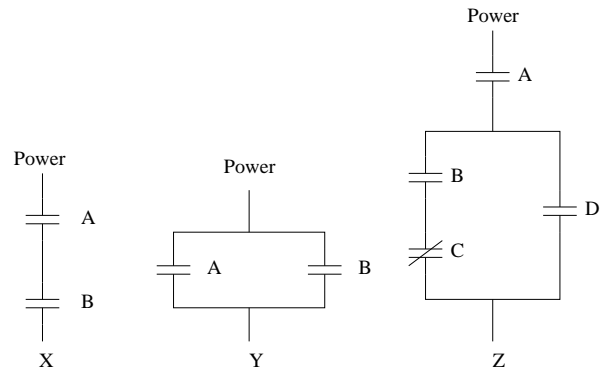


Figure 1: Simple Ladder Diagram Examples

either by context or by the device’s label.¹ Capacitors are not used in ladder diagrams, so in these examples all of the described symbols will represent either actual, physical relay contacts or “logical” contacts used to represent the truth value of a Boolean variable or input.

A symbol without a diagonal line indicates “normally open” contacts that allow current to flow only when the relay is turned “on” or the value represented by the symbol is true. A diagonal line through the symbol indicates “normally closed” contacts that allow current to flow when the relay is “off” or the value represented by the symbol is false – the equivalent of a logical “not” operator.

Figure 1 shows three simple examples of ladder diagrams. The output X will be true (powered) if A and B are both true (switches closed). The output Y will be true (powered) if A or B are true (switches closed). The output Z will be true if A is true and D is true, or if A is true, B is true, and C is false:

$$X = A \cdot B$$

$$Y = A + B$$

$$Z = A \cdot (D + (B \cdot \overline{C}))$$

Where ‘ \cdot ’ is Boolean and, ‘+’ is Boolean or, and \overline{C} is the unary Boolean negation of C.

¹A device labeled C103 on a schematic would likely be a capacitor, while one labeled K103 would be a pair of relay contacts.

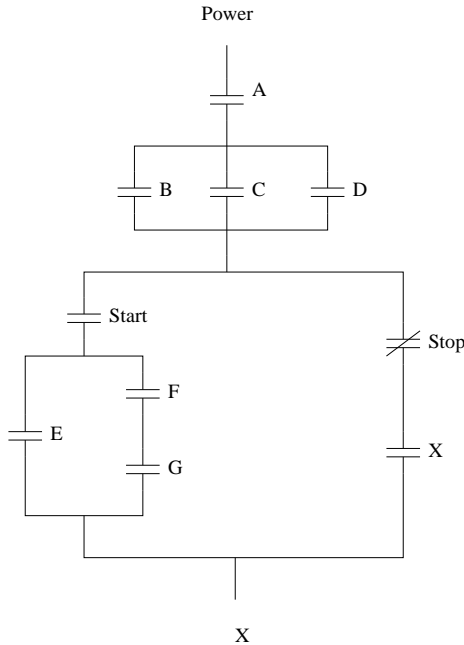


Figure 2: A More Complex Ladder Diagram

In ladder diagrams such as Figure 1, A, B, C, and D might be inputs from proximity or limit switches. Something analogous to temporary or internal state variables are implemented by using the output to control another relay whose contacts show up in other combinatorial circuits. A relay typically had several sets of output contacts – both true (normally open) and inverted (normally closed).

A circuit representing a more complex Boolean expression might have ten, twenty or a hundred elements that might include counters, time delays, and other functional blocks. By using feedback (the contacts controlled by the output signal are used in the circuit that controls the output), it is possible to implement a circuit which has an internal state such as an R-S flip-flop.

The example in Figure 2 shows a circuit with an internal state (which, in this case, happens to be equal to the output signal). In Figure 2, the state of X is given by the Boolean expression:

$$A \cdot (B + C + D) \cdot [(Start \cdot (E + F \cdot G)) + (\overline{Stop} \cdot X)]$$

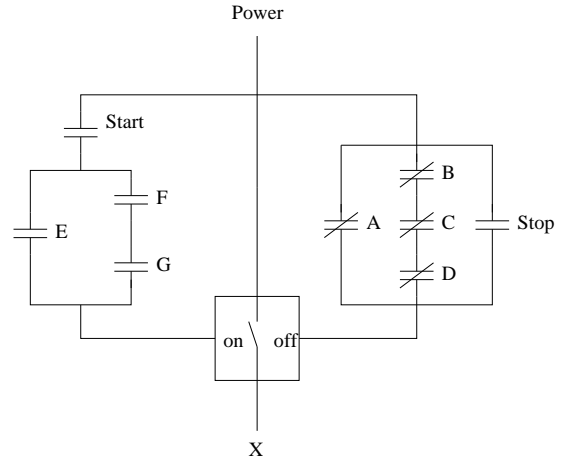


Figure 3: A Ladder Diagram Using a Function Block

Such a diagram might be generated by an electrician or mechanical engineer to satisfy the following requirements:

- X is the control signal for some output device.
- X may not be enabled unless the following interlocks are present: A, and any one of B, C or D.
- X may be turned on by pressing the Start button when either condition E is true or conditions F and G are both true.
- X will remain on (assuming the interlocks stated above are present) until the Stop button is pressed.

Rather than showing feedback as in Figure 2, a functional block containing a “switch” might be defined that has two control inputs; one to turn the switch off, the other to turn the switch on. When neither of the two control inputs are true, the switch’s state remains unchanged. The switch’s behavior when both control inputs are true could be defined a number of ways, but we will define the “off” input has having priority: when both the “on” and “off” signals are true, the switch is off.

Figure 3 shows a ladder diagram using such a switch block that meets the same requirements as did the ladder diagram in Figure 2. This diagram more clearly shows the purpose of this control sub-system: the logic on the

left controls when the output will be turned on, and the logic on the right shows when it will be turned off.

2.2 Ladder Diagrams Today

For several decades before computers were invented, mechanical engineers and plant electricians created complex discrete controls by designing relay-logic systems. Today, Boolean computation using actual series/parallel combinations of physical contacts is rarely used in systems with more than a few inputs and outputs. Even though the use of relays to perform logic is rare, the use of ladder diagrams to document the operation of electro-mechanical systems is still widely used. The primary use of ladder diagrams is to program microprocessor-based PLCs.

These PLCs range in capabilities from 5-10 I/O points and a serial port for programming, to those with hundreds of I/O points, a graphical local operator interface for editing diagrams, and a LAN interface for connection to a supervisory control system. Purchase prices range from a few hundred to several thousand dollars and they are sold by most of the providers of process control equipment: GE, Westinghouse, Gould, L&N, Toshiba, Allen-Bradley, Square D, etc. Specifications for a typical low-end PLC programmed using ladder diagrams, the Toshiba EX14B, is included as Appendix E.

PLCs programmed by ladder diagrams are used in almost all applications involving discrete process control.² These processes can consist of anything that requires a set of discrete operations be performed in set sequences: injection molding, stamping, conveyor operations.

PLCs interpret ladder diagrams anywhere from ten to one thousand times per second. The move from physical implementation to interpretation by a microprocessor has introduced both new capabilities and new restrictions to the process of designing a system. No longer is the designer limited by the number of contacts per relay or the fan-out restrictions that limited how many relays an output could control. However, no longer is the response of the system instantaneous (within electro-mechanical lim-

²Discrete process control refers to systems where a controller switches output devices on and off according to pre-defined requirements – as opposed to continuous process control where an analog output (speed, temperature or pressure for example) is controlled to maintain a system within desired operating limits.

its) and no longer are all calculations performed continuously and in parallel.

The ladder diagram interpreters evaluate diagrams in a specified order, from top to bottom, left to right. To continue the analogy with a physical implementation, current can only flow down the page and not up the page.

The “computerization” of ladder logic has made it easy to add higher-level devices to the tool set. The designer commonly has access to blocks such as a counter that counts input pulses and sets an output to true after N pulses have been accumulated. Such a counter might have several inputs such as increment, decrement, clear, and a “trigger” level that determines when the output goes true. Other functional blocks that are commonly seen include time delays and analog input blocks with binary outputs for multiple high and low alarm conditions.

2.3 The Need for Code Generation

While PLCs are suitable for many discrete control applications there are situations when either custom designed hardware or a more general purpose computer must be used. A more general purpose computer might be required in order to provide access to hardware or software capabilities that aren’t available in a PLC. For example, an application might require some sort of machine vision capabilities. While video capture equipment and pattern recognition facilities are readily available for workstation-class machines, such facilities aren’t available for PLCs.

Unusual physical requirements such as an extended temperature range or hardening to resist radiation or electro-magnetic interference might preclude the use of off-the-shelf PLCs and require that custom hardware be designed. Unusual interface or performance requirements might also require the design of custom hardware or the use of a particular computer not originally intended for discrete control applications.

If either a general purpose computer or custom designed hardware is used, it is likely to be the exception within the facility. If the majority of the discrete control within a plant is performed by PLCs programmed using ladder diagrams, it would be advantageous if our “exception” application could also be programmed using ladder diagrams. This would allow personnel already trained in the use of ladder diagrams to create and maintain the discrete control portion of the application without hav-

ing to learn a programming language such as C. Since ladder diagrams are easily generated and understood by non-programmers, they are sometimes used to document control even when the actual program is written in a compiled language. The manual translation of ladder diagrams into a procedural language suitable for use either on a general purpose computer or custom hardware is a time-consuming and error-prone task.

Automatic generation of source code from ladder diagrams will allow users familiar with ladder diagrams to continue to use familiar design methodologies even though the target machine does not provide a ladder diagram interpreter. To change the discrete control functionality, the user merely needs to edit the pertinent ladder diagram and then do a “make” which will re-generate the source code from the altered diagram and then recompile the application. Once such a system is set up, maintenance can be performed with less risk of introducing bugs, and there is less required programming experience on the part of the person maintaining the system.

3 Project Objectives

In the past, the programmer would be given ladder diagrams showing the system requirements and would have to translate these into source code.

This project will provide tools to automatically generate source code from ladder diagrams. The generated source code can then be incorporated in a compiled program. This would allow the control logic and operation sequence to be designed by the industrial or mechanical engineer using a familiar methodology and notation (ladder diagrams) even though an off-the-shelf ladder-logic controller is not suitable for the task. The ladder diagrams would be designed by the engineer designing the process or the machine, and then automatically translated into C for incorporation with the remainder of the software.

3.1 Related Work

A literature search revealed nothing in the area of generating source code from ladder diagrams. There has been work done [Devanathan et al] on programs to do the inverse: generate ladder diagrams from a definition written in a state-transition language. The ladder diagrams

are then downloaded into the ladder controllers mentioned previously.

4 Project Architecture

The project consists of two main components: an X11-based graphical ladder diagram editor and a code generator. The code generator may be run from within the diagram editor or it may be run as a stand-alone utility.

The decision to separate the functionality into two programs was driven by the additional utility of being able to perform the source code generation as part of an automated software build facility such as is provided by the Unix “make” utility. If the source code generation was available only from within the ladder diagram editor, it might be possible for the user to edit the diagram but forget to run the code generation function. This would result in the ladder diagram being out of sync with the corresponding source code.

If the source code generator is available as a stand-alone utility, then it is quite easy to use a utility like “make” to insure that when a ladder diagram has been changed the corresponding source code is regenerated and compiled only as needed.

The following sections will describe the design and operation of the two portions of the project.

5 LEd: A Diagram Editor

The portion of the project most visible to the user is the ladder diagram editor. This section will describe the program’s features and usage and then address the design of the editor.

5.1 Program Features

From the user’s perspective the ladder diagram editor resembles a simple drawing or schematic capture program. LEd has facilities for:

- Placing objects of various types (relay contacts, function blocks, etc.) on the diagram.
- Editing the names of these objects.

- Making interconnections between the terminals of objects.
- Moving and deleting objects.
- Saving, loading, and printing ladder diagrams.
- Running the code generator program and viewing its output.
- Viewing hypertext help documentation.

Figure 4 shows the application window of the ladder diagram editor. The editor is invoked by typing the following at the shell prompt.

```
$ led [filename]
```

If a file name is provided on the command line, the editor will open that file and display the contained diagram in the main sub-window. Once the editor has started, the user is presented with a menubar across the top of the application window and a toolbar down the left side of the application window.

5.1.1 Menubar

The menubar provides for operations ancillary to the actual manipulation of the drawing. It contains four drop-down menus:

File contains entries that allow the user to open and save diagram files, print the diagram, and exit the program.

View allows the user to view a list of objects on the current diagram, the corresponding net-list, or the code produced by the code generator program.

Options allows the user to choose the settings for various program options that control line width, drawing size, etc.

Help allows the user to start an HTML browser that is used to view the program's help documentation.

5.1.2 Toolbar

The toolbar along the left side of the application window is used for manipulation of the drawing itself. The functions provided by the toolbar can be divided into four categories:

1. The top seven buttons are used to place new objects on the drawing. First the user left-clicks on the desired object (the function block button will pop-up a menu that allows the type of block to be chosen). The user then moves the pointer into the drawing window – a new object of the selected type will be shown in red, and it will follow the pointer as it is moved about the drawing window. When the new object is in the desired position, the user left-clicks and the object is placed on the drawing. If the user right-clicks the new object is discarded.
2. The wire button is used to interconnect devices on the drawing. The user left-clicks on the wire button, then moves the pointer into the drawing window. In the drawing window, a left-click will start a new wire at the pointer position (terminating a pending wire if one is being drawn). Center-clicking will terminate the wire currently being placed. Right-clicking will discard the wire currently being placed.
3. The text button is used to edit the name associated with each non-wire object in the drawing. To edit an object's name, the user left-clicks on the text button then left-clicks on the object. A text cursor will appear and the user may then enter or edit the object's name.
4. The move and delete buttons are used to move and delete either single objects or groups of objects. To move or delete a single object the user left-clicks on the appropriate button and then clicks on the object. If the object is to be moved, it will change color to red and follow the pointer as it is moved. A left-click will place the object in its current location and a right-click will abort the move operation; returning the object to its original location.

Deleting or moving multiple objects is similar except the user selects a rectangular region by center-clicking at opposite corners of the desired region.

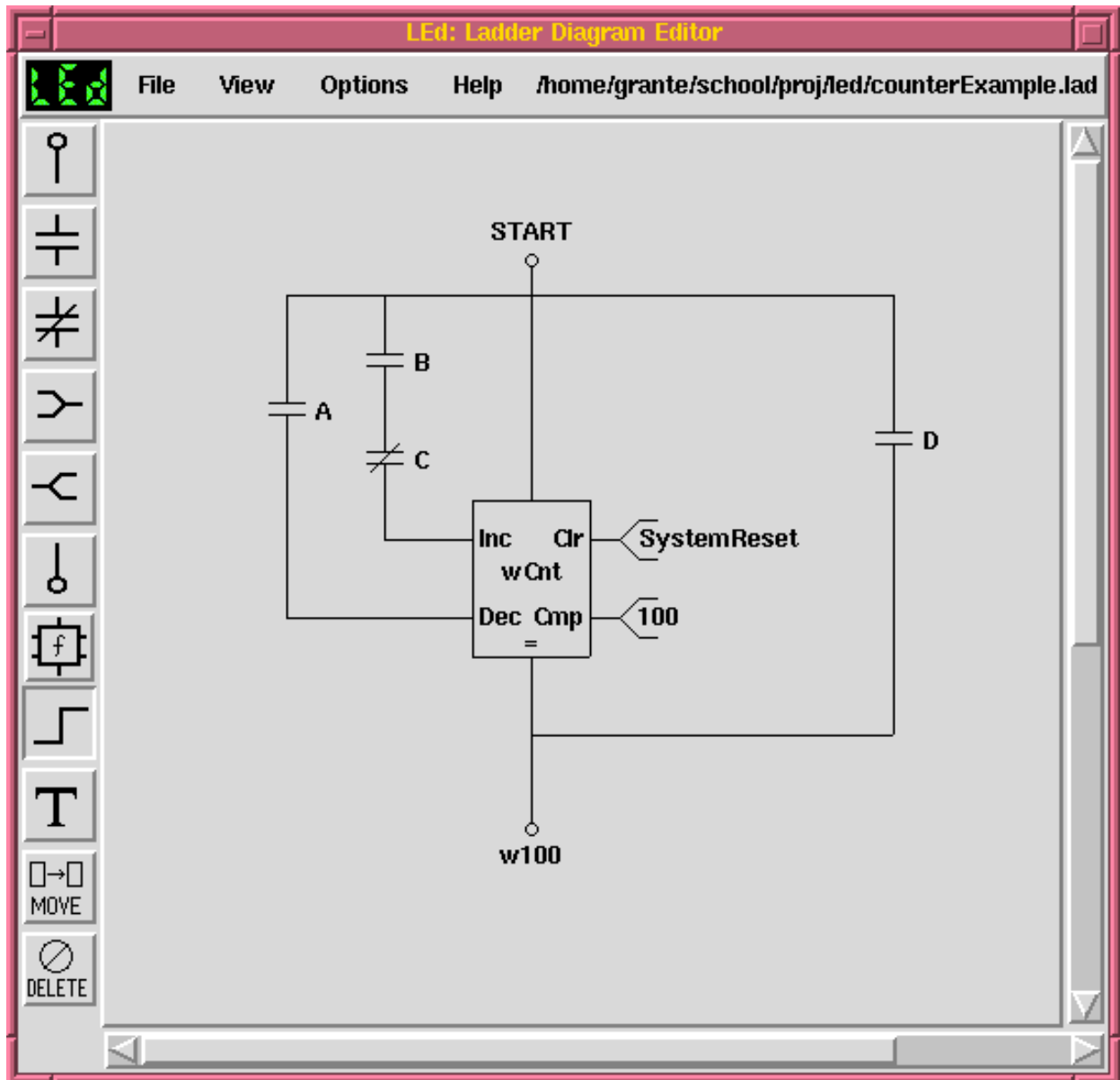


Figure 4: Ladder Diagram Editor

The Unix man page and hypertext help documents for led are included as Appendixes A and B.

5.2 Development Platform

The language chosen for the project was the STk package [Gallesio]: a Scheme interpreter linked with the Tk X11 toolkit and widget set [Outsterhout].

Several language options were investigated for the implementation of a ladder diagram editor:

- Smalltalk.
 - Digitalk Smalltalk with a third party user interface kit.
 - Gnu Smalltalk with X11 interface library.
 - Parc Place Smalltalk.
- The Tcl/Tk Tcl interpreter and X11 toolkit.
- C and various X11 widget sets.
- The STk Scheme interpreter and X11 toolkit.

Smalltalk. A freely distributable and easily extended package was one of the goals of the project. For this to be practical, the development tools must also be freely distributable.

Of the Smalltalk systems, only Gnu Smalltalk met this requirement. Unfortunately, the X11 interface available for Gnu Smalltalk was rather primitive and didn't provide a set of widgets sufficient for the task.

Tcl/Tk. This language and X11 toolkit was designed as part of a set of tools to do integrated circuit design and includes a very sophisticated "canvas" widget that provides the fundamental functions required for an object-oriented drawing application like editing ladder diagrams. The important features of the Tk canvas widget are:

- Two dimensional scrolling within a window.
- Direct support for the manipulation of graphical "objects".
- Built in capability to generate a postscript representation of the "drawing" contained in the widget.

While the Tk toolkit and widget set was ideally suited for the ladder diagram editor portion of the project, the Tcl language didn't seem so well suited. The Tcl language is basically a string-oriented command language akin to a Unix shell. It lacks any data type other than the string, and the syntax and semantics of the language itself are a bit odd.

C. Gnu C provided an excellent, freely distributable compiler. A freely distributable, high-level widget set that provided the required features wasn't found.

STk. The STk package is a Scheme interpreter mated with the Tk toolkit. Scheme is a small, modern, statically scoped dialect of LISP. It is a well defined, standardized³ language with a large library of available source code for functionality, ranging from hash tables and queues to a complete object oriented programming system based on CLOS, the Common Lisp Object System. The Advantages of STk are:

- STk is available for most Unix/X11 platforms.
- STk source code is available and freely distributable.
- STk is easily extended in C.

The disadvantages of STk are:

- Inability to generate stand-alone, binary applications. An STk application is like a Unix shell script – if you don't have the interpreter installed on your system, you can't run the program.
- STk has not yet been implemented for non-Unix/X11 platforms.
- The Tk toolkit is designed to be used by an interpreted language rather than a compiled language, so it's not likely that it will be possible to produce a stand-alone binary application in the future.

³Like the C language, there are two "standards." One is IEEE P1178-1990 "IEEE Standard for the Scheme Programming Language." The second standard is the "K&R" of Scheme: "Revised⁴ Report on the Algorithmic Language Scheme," Clinger and Reese (Editors)

5.3 The Design of LEd

The design of the editor must address three major areas:

- Manipulation of graphics – the actual mechanics of editing the drawing.
- Maintenance of a data structure that is sufficient to recreate the drawing and an associate net-list.
- Generation of a net-list suitable for post-processing.

In order to explain how these problems (especially the first two) are solved, it is important to understand the functionality available in the Tk toolkit's canvas widget, which is seen as the main drawing area in the application window. The canvas widget provides for the display and management of several types of graphical primitives: lines, ovals, text, rectangles, etc.

These primitive objects are managed and organized mostly by the assignment of "tags" to objects. Each primitive has a unique, automatically generated numerical tag. The user is allowed to assign additional tags to primitives as desired. A primitive object may have any number of tags, and tags are not required to be unique to an individual primitive. A new tag may be assigned to primitives within a given region, to the primitive closest to a specified coordinate, to the primitive(s) that have a specified tag, or to the primitive selected by the pointer. Additionally, it is possible to retrieve all of the tags assigned to a particular primitive, as well as other information about the primitive such as coordinates and type (line, oval, text). In the case of text objects, the string value may be also be set, modified, or queried.

These primitive objects are manipulated by operations such as move, delete, change color. The primitives upon which the operations are to be performed are specified by the previously discussed tags. For example, all of the primitives with a specified tag may be moved 100 pixels to the left and 75 pixels down.

5.4 Internal Data Representation

The most important design decision is how the ladder diagram is represented internally within the editor. Conceptually, the diagram consists of a set of high-level objects (relays, function blocks, labels, wires) each with a name (except wires) and a position on the drawing.

A certain minimum amount of data is maintained internal to the canvas widget: The coordinates of all of the graphical primitives and the string values of those primitives which are text objects.

This raises a number of design questions revolving around the central issue of how to partition the editor's representation of the diagram between the canvas widget's internal state (using the tags described previously) and a data structure within the application program:

- If the complete state of the diagram is to be resident in a data structure external to the canvas widget, how much effort is required to make sure that the information duplicated between the canvas widget and the external data structure is maintained in a consistent state?
- If the state of the diagram is to be split between the canvas widget's internal state and an external data structure without duplicating information, how will changes to the diagram state be coordinated?
- Is the tag mechanism powerful enough to allow the state of the diagram to be contained entirely within the widget?

In order to decide on one of the three approaches, the requirements that must be met have to be defined:

Grouping of Primitives. Groups of graphical primitives need to be defined so that the application program can perform the desired operations on higher level objects such as relays and function blocks. This can be accomplished external to the canvas widget by maintaining lists of primitives corresponding to each high level object. It may also be accomplished within the canvas widget by assigning a unique tag to the set of primitives that make up each high-level object.

Attribute Assignment. Each high level object has several attributes that need to be manipulated: a user-specified name, the location on the drawing, the object type (relay, wire, label, etc.). These attributes are required for generation of the netlist – the locations are used to determine the connections between objects and the other attributes (name, type) are required by the code generation algorithm.

The maintenance of these high level object attributes can be performed external to the canvas widget with a data structure that is instantiated for each high level object. This may also be done using the canvas widgets internal state. The canvas widget certainly must keep track of the location of each primitive, and if we define one of the primitives within each high-level object as the “origin” of that object, then we can use the location of that primitive to define the location of the high level object.

To maintain other attributes tags with a defined format may be assigned to the primitives that compose a high level object. For example we can decide that a tag with a value of “type-XYZ” will be assigned to an object to define the type of that object as “XYZ”.

Likewise, we can assign a predefined tag to the text primitive within a high level object that contains the user edit-able name for that high level object. This will allow us to edit or retrieve the name for a given high level object.

The Road Chosen. The decision was to utilize the canvas widget’s tag facilities and its internal state to represent the entire state of the drawing. This approach has several advantages:

- No duplicated data. Since all state information is contained in the canvas widget, there can’t be any discrepancy between the application’s representation of the drawing and what is displayed to the user.
- No effort is required to coordinate changes to two data structures. When an object is deleted, it is not necessary to delete it both from an external data structure and from the canvas widget.
- The routines required to manipulate object attributes using the canvas widget’s tag mechanism require little or no additional work compared to using an external data structure.

5.4.1 Example of Tag Usage

Primitive objects (lines, text, ovals) need to be grouped into high-level objects such as relays and function blocks. This is done by assigning a unique high-level object tag to all of the primitive objects that compose a particular high-level object. These tags always take the form “obj-NN” and are generated automatically by the application

code as new high-level objects are created. These high-level object tags can be seen in the net list generated by the editor.

High-level objects can then be operated on as single logical objects. For example, when the user wishes to move a function block, the following takes place:

- When the mouse is clicked on one of the primitive objects that make up the function block, the editor reads the tag list from that primitive object and selects the “obj-NN” tag. For our example, we will assume it is “obj-23”.
- A temporary “move-tag” is added to all primitive objects with the tag “obj-N23”.
- The canvas widget is then told to change the color of all primitive objects with the tag “move-tag” to red.
- As the pointer is moved, the widget is told to move all objects with the tag “move-tag” the same relative distance as is moved by the pointer.
- When the move is completed, the widget is told to change the color back to black for everything with the tag “move-tag,” and then the tag “move-tag” is deleted from all primitive objects.

Once it was decided to maintain the drawing state in the canvas widget, the implementation of the various editor functionality followed in a straightforward manner.

5.4.2 Net-list Generation

When a ladder diagram is saved by the editor, two data structures are written to the file:

- A list of the objects in the drawing including their attributes (type, location, etc.). This information is used to re-create the drawing the next time the file is opened by the editor.
- The netlist corresponding to the ladder diagram. This net-list is what is used by the post processor to generate source code.

The netlist consists of two parts: an object table that lists the various devices along with their names and types, and the actual node-list that contains the connectivity information.

The generation of the object table is a trivial task that requires nothing more than retrieving a list of objects other than wires and retrieving the relevant attribute information from the canvas widget tags.

The generation of the node-list is slightly more complex, but still consists of three steps.

- First, all of the wires in the diagram are sorted into sets called nodes. All of the wires in a particular node are connected to each other through zero or more other wires, and none of the wires in a node are connected to any wire in a different node.
- All of the device terminals connected to each node are listed as belonging to that node.
- The device terminals not connected to any wire are searched to determine if they are directly connected to any other terminals without using a wire. If so, a new node is defined for each set of inter-connected terminals and assigned those inter-connected terminals.

While the algorithm is not overly complicated, the current implementation is $O(n^2)$ and can take several seconds to process a moderately complex ladder diagram on a slow workstation.

6 lad: A Code Generator

Having described the design and operation of the ladder diagram editor, the code generation portion of the project is next to be addressed. This component of the system is a post-processor that generates source code from a net-list composed of a device table and a node list. The code generator may be run from inside the ladder diagram editor or as a stand-alone utility.

If the code generator is to be run outside the ladder diagram editor, the ladder diagram file must be run through a formatting utility that understands the file format written by the ladder diagram editor. This formatter is a small Scheme program called *ladformat*. This program does not use the X11 toolkit used by the editor. Therefore, it may be run from a makefile or as part of a shell script.

The Unix man pages for *lad* and *ladformat* are included as Appendixes C and D.

6.1 Overview

The ladder diagram paradigm is that of an electrical circuit: if there is a complete path from the power source to the output device, then that device is on. If not, then that device is off. The design of the code generator reflects this view of a ladder diagram. The ladder diagram is represented as a graph which is subjected to a series-parallel reduction that produces and-or trees that represent series-parallel combinations of graph nodes. Then a depth-first search is used to find possible paths from the “source” node to the outputs. The nodes (and corresponding and-or trees) encountered along that path are used to construct a Boolean expression that represents the conditions under which the output is to be true.

The content and format of the generated code is controlled by a set of template files that the user may edit and extend.

6.2 Device representation

A device (and therefore, a graph node) has any number of terminals which are connected to other devices. Two of the terminals are treated specially. These two terminals (the “input” and “output” terminals) represent the relay contacts. Each object in a diagram is required to have these two terminals by IEC Standard 1131-3 (II) “Uniform PLC Programming” in order to preserve the binary data flow view. The model used by the code generator also allows for additional input terminals that can control the internal state of a device – however, there are never additional output terminals.

The output terminal is true if the input terminal and the state of the device are true. The state of the device may, in the case of a normally-open relay, be simply the value of a Boolean variable. It may, for a normally-closed relay, be the inverted value of a Boolean variable. For complex function blocks, the device’s state is controlled by control inputs and block-specific logic which may include internal state information.

The definition of a “function block” is any device that has terminals other than required single input and output terminals.

The “output” device is actually a function block whose input and output terminals can’t be connected to anything (this is *not* in accordance to IEC 1131-3). What would

appear to be an input terminal on the diagram is actually a control input. The reason for this will become evident later.

6.3 Graph generation and reduction algorithm

After the device table has been read and the graph has been constructed, the next step in the code generation process is that of combining nodes that are in series or parallel. This processing consists of 3 steps illustrated in figures 5 through 7:

1. Construct a directed graph from the net list. Each node in the graph represents a single device from the ladder diagram. Arcs between nodes represent connections from the output terminal of one device to the input terminal of another.
2. Add an and/or tree to each node of the graph. This and/or tree has a single terminal node whose value is the name of the device represented in the graph node.
3. Reduce the graph of and/or trees:
 - (a) Combine nodes that are connected in parallel into a single “or” node whose children are the nodes that were connected in parallel.
Two nodes are defined to be in parallel if their input terminals are connected to each other and their output terminals are connected to each other.
 - (b) Combine nodes that are connected in series into a single “and” node whose children are the nodes that were connected in series.
Two nodes are defined to be in series if the output of one is connected to nothing except the input of the other.

Since combining series nodes may create additional parallel sub-graphs (and combining parallel nodes may create additional series sub-graphs) the above two steps are repeated until no additional series or parallel reductions are possible.

Listing 1 File template example

```
typedef enum{False=0, True=1} boolean;
/* globals for all blocks */
%h
/* code */
void eval_%f(void)
{
%c
}
```

6.4 Code generation algorithm

The code generation phase is controlled by code template files. There are two types of template files: a file template and a set of block templates. The processing of template files is handled much as if they were format strings for the ubiquitous printf() family of library functions. The template file is read in and copied to the output with “%X” character sequences replaced by various formatted output. The set of output specifiers, %X, depends on the particular type of template file being processed.

6.4.1 The file template

The file template is used once per invocation of lad and controls the format of the output file. The results of the output specifiers available for use in the file template are shown below.

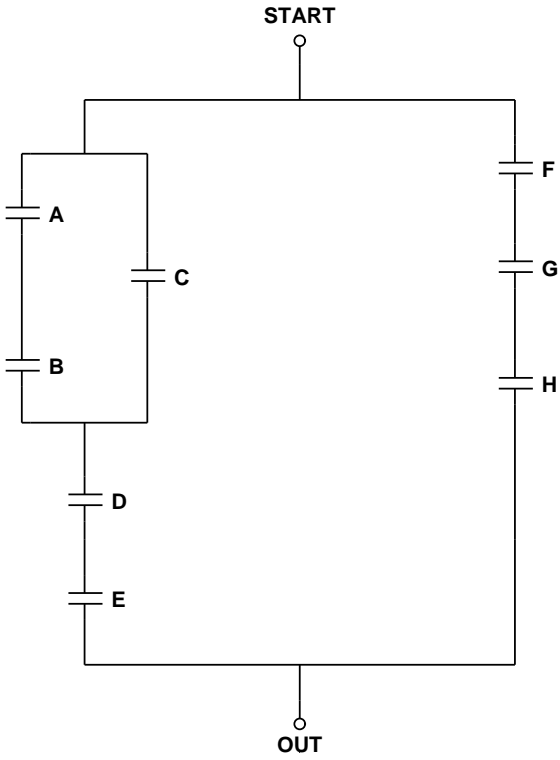
%f Output the “function” name (this defaults to the name of the input file, but may be over-ridden with a command-line option).

%S Output the results of processing the block template (for each function block in the net list) with the suffix *S*. *S* may be any single character other than ‘f’.

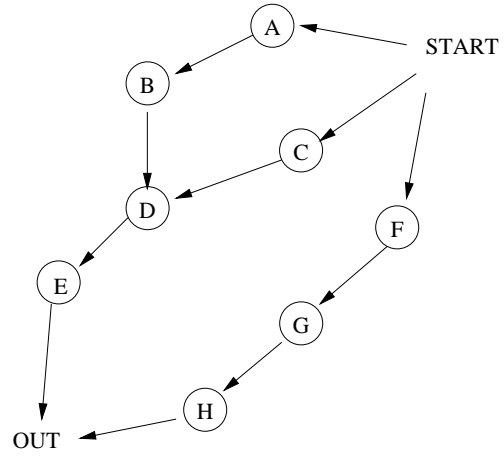
An example of a simple file template is shown in Listing 1.

6.4.2 The block template

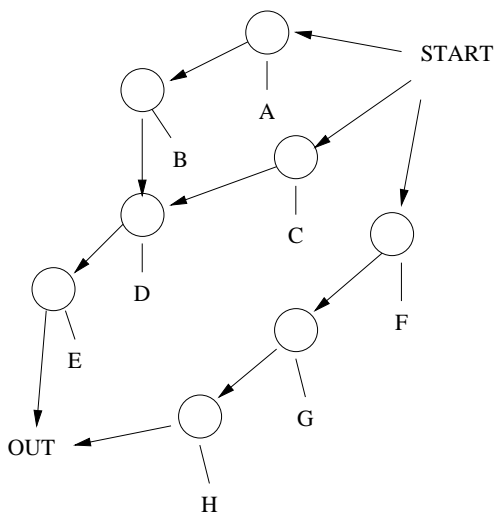
Each type of function block used in the ladder diagram must have a block template file for each output specifier



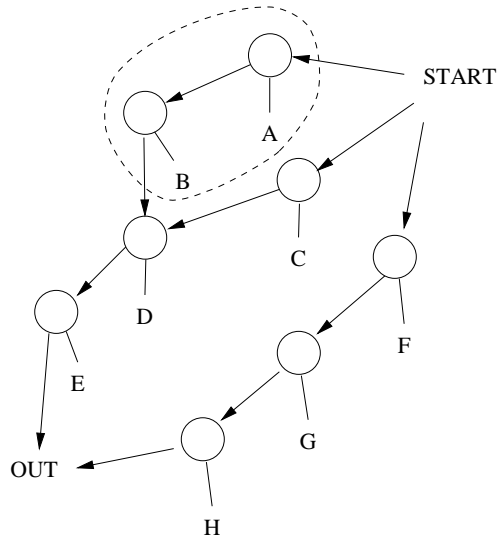
Ladder diagram for reduction



Step 1: Initial graph

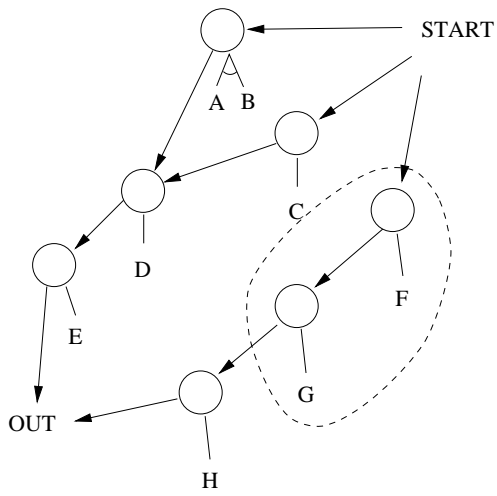


Step 2: And/or trees added to graph

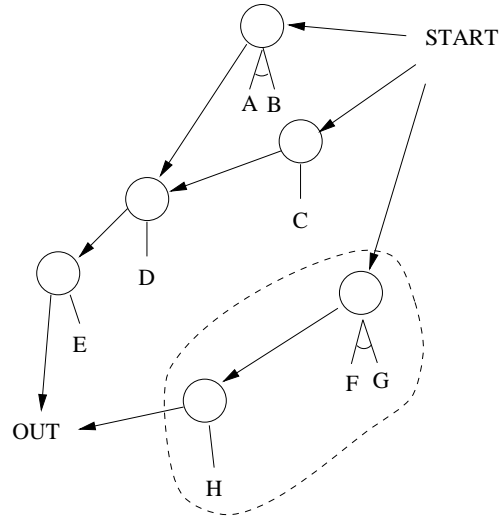


Step 3(b): Series nodes to be combined

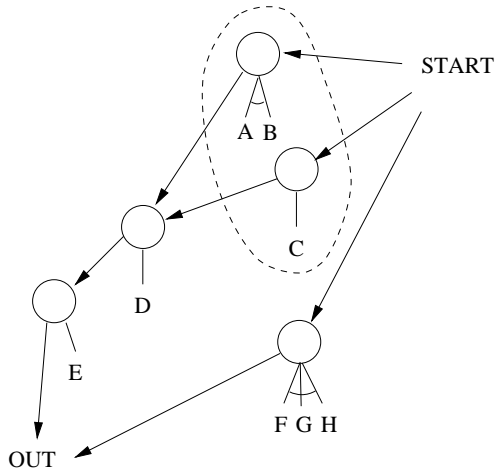
Figure 5: Graph Reduction Example



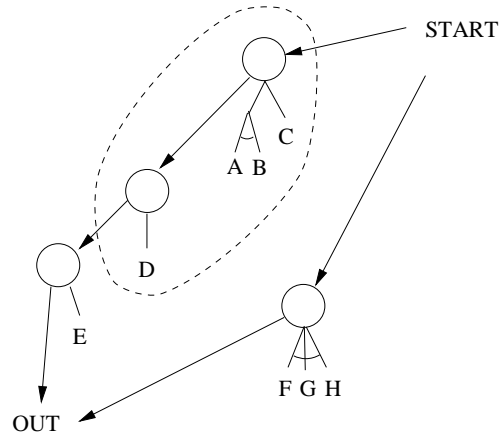
Step 3(b): Series nodes to be combined



Step 3(b): Series nodes to be combined

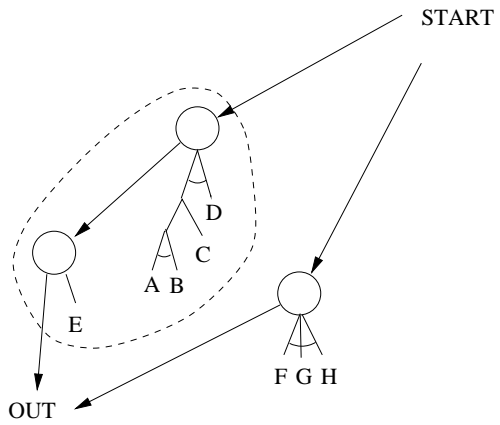


Step 3(a): Parallel nodes to be combined

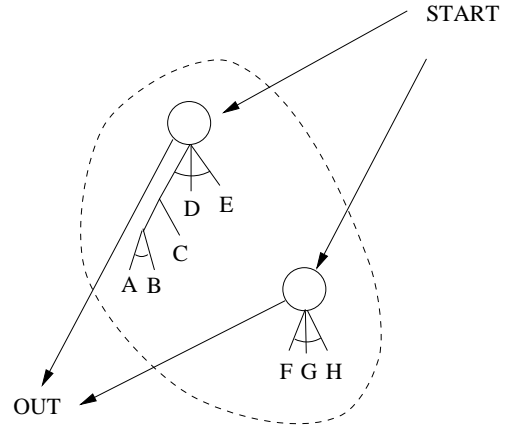


Step 3(b): Series nodes to be combined

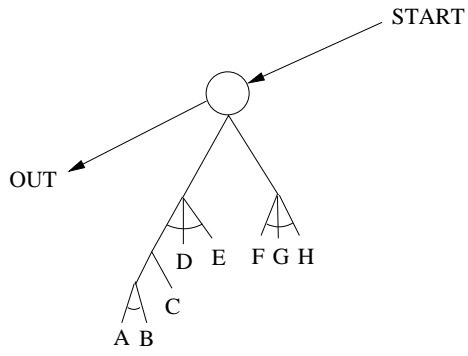
Figure 6: Graph Reduction Example (cont.)



Step 3(b): Series nodes to be combined



Step 3(a): Parallel nodes to be combined



Final graph with and/or tree

Figure 7: Graph Reduction Example (cont.)

Listing 2 Latch block template example

Header file: *latch.h*

```
/* latch %n state variable */  
static boolean %n;
```

Body file: *latch.c*

```
/* latch %n evaluation code */  
{  
if (%3)  
    %n = False;  
else if (%2)  
    %n = True;  
}
```

type other than %f that is present in the file template. All block templates are processed identically, and the output specifiers that are available for use in block template files are shown below.

%n Output the device “name” (this is the name the user entered using the LEd diagram editor).

%N Output the Boolean expression that evaluates to the value present on the function block’s control terminal *N* (where *N* is a value from 2 through 9).

The example file template shown in Listing 1 contains two output specifiers that require block template files: %h and %c. Therefore, each function block must have two block template files whose names are *blocktype.h* and *blocktype.c*. The file template is set up so that the .h block template output is located outside of any C function and is to be used for declaring global variables.

The .c block template output appears within a C function body and therefore contains executable program statements.

Examples of the block templates for a latch function block are shown in Listing 2. The latch function block .h template file simply declares a single variable that is used to hold the state of the latch. The .c template file contains C statements that evaluate the states of the two control inputs (set and clear) and set the latch’s state accordingly.

The latch template example illustrates the two basic operations that must be performed by the block templates for any type of function block:

- Declare a state variable with the same name as the device. This variable may be used in Boolean expressions that are generated for other blocks’ control terminals, so it must be visible to any code produced by the file template. It may or may not be visible outside the file, depending on the wishes of the user.
- Produce code that controls the state variable by evaluating the control inputs and performing whatever computations are required.

6.5 Control terminal evaluation

When the code generator is producing code from a function block template and it encounters a token of the form %N ($2 \leq N \leq 9$) it emits a Boolean expression that represents the value present on control terminal *N*. This expression takes one of two forms. If the terminal was connected to a label, then that label’s value is used. This may be any string and could represent a constant value or a variable. Such usage of a label can be seen later in Section 7 where the value “100” is attached to a control terminal for a counter block.

If the terminal is connected to other devices, then a depth first search is performed to find possible paths from the terminal to the source node. The and-or trees belonging to nodes found along the paths are formatted and combined to form an infix Boolean expression which is then emitted.

6.6 Example templates for a counter

To further illustrate the operation of block templates, an example will be shown for a counter. This device will have two state variables: the required Boolean state associated with all devices, and an integer counter that is manipulated based on the control inputs. The example counter device will have four control inputs:⁴

⁴Remember, terminals 0 and 1 are the input and output terminals which are used to construct the and/or trees, and need not be addressed by block templates

Listing 3 Counter block template example

Header file: counter.h

```
/* counter %n state variable */  
static boolean %n;
```

Body file: counter.c

```
/* counter %n evaluation code */  
{  
static int count;  
static int lastinc;  
static int lastdec;  
int inc = %2;  
int dec = %3;  
  
if (inc && !lastinc)  
    ++count;  
  
if (dec && !lastdec)  
    --count;  
  
if (%4)  
    count = 0;  
  
lastinc = inc;  
lastdec = dec;  
  
%n = (count > (%5));  
}
```

- Terminal 2: Increment. A false to true transition will increment the integer counter associated with the device.
- Terminal 3: Decrement. A false to true transition will decrement the integer counter.
- Terminal 4: Clear. A true value on this input will set the integer counter value to 0.
- Terminal 5: Compare. This terminal is to be connected to a “label” device which is to represent an integer value.

The Boolean state of the device shall be true iff the counter is greater than the “Compare” value. Listing 3 shows the template files for the counter device requirements defined above.

Listing 4 Output block template example

Header file: output.h

```
/* output %n state variable */
```

```
boolean %n;
```

Body file: output.c

```
/* output %n evaluation code */  
  
%n = %2;
```

6.7 Example templates for an output

An output device’s templates are considerably simpler. The purpose of an output device is to evaluate the value on its single control terminal (terminal 2) and assign that value to a globally visible variable. The templates to accomplish this are shown in Listing 4.

7 A complete example

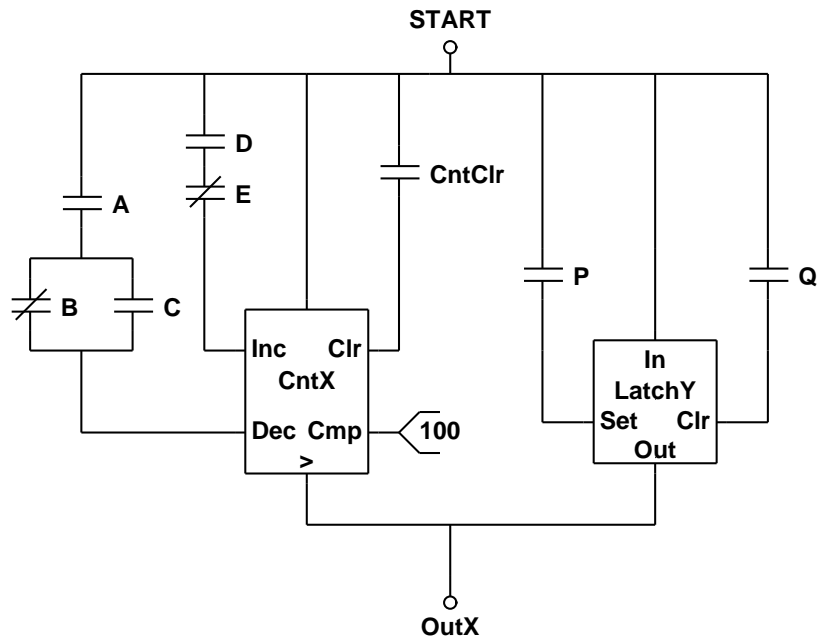
We will now see a complete example of a ladder diagram using the example devices defined in the previous sections. Figure 8 shows the ladder diagram created with the ladder diagram editor along with the code generated from that ladder diagram. This example shows many of the features of the tool set:

- Use of standard normally open and normally closed relay contacts.
- Use of two types of function blocks: a counter and a latch.
- Use of a label device to supply a value of “100” to one of the control inputs to the counter.

8 Analysis

8.1 Statistics

The LED editor is comprised of approximately 2800 lines of Scheme (comments and whitespace not counted). Of



```

typedef enum{False=0, True=1} boolean;
/* globals for all blocks */
/* latch LatchY state variable */
static boolean LatchY;
/* counter CntX state variable */
static boolean CntX;
int OutX;
/* code */
void eval_completeExample(void)
{
  /* latch LatchY evaluation code */
  {
    if (Q)
      LatchY = False;
    else if (P)
      LatchY = True;
  }

  /* counter CntX evaluation code */
  {
    static int count;
    static int lastinc;
    static int lastdec;
    int inc = (D&&!E);
    int dec = (A&&(!B|C));

    if (inc && !lastinc)
      ++count;

    if (dec && !lastdec)
      --count;

    if (CntClr)
      count = 0;

    lastinc = inc;
    lastdec = dec;

    CntX = (count > (100));
  }

  OutX = (LatchY||CntX);
}

```

Figure 8: A complete ladder diagram example

that total, code reused from other projects totals approximately 800 lines. The majority of the reused code is in the HTML display routines used by the help browser. These routines were modified to support relative URLs and anchors.

The lad code generator is approximately 1200 lines of C source code (comments, whitespace, and lines containing only braces not counted).

8.2 General comments

The tools have been used extensively only by the author (who naturally finds the user interface intuitive and simple). Since the author of a program is rarely a good judge of the user interface, additional input from users would be invaluable in tuning the mechanics of editing a drawing. As a whole, the editor's user interface turned out quite nicely. Thanks in part to the Tk toolkit, the resulting user interface has a polished and professional feel.

Some portions of the project involved more work than expected. These were the "auxilliary" user interface items such as the help browser and the file open/save browser and dialog box. Something as apparently simple as a file browser that allows the user to navigate through a directory tree and select a file to open becomes rather complex when it must handle things like wildcards and various error conditions such as files that can't be opened or files that can't be written to.

The HTML help browser also required more work than anticipated. The STk package included support for displaying HTML, and an example browser application. However, the HTML support wasn't quite complete and support for relative URLs and anchors had to be added. While the HTML help browser required more work than expected, the results were well worth the additional effort. Providing on-line help as a set of HTML files not only makes authoring the content much simpler, it also allows the user to browse, print, and search the help documentation even if he can't get the ladder diagram editor to run.

The most challenging part of the project was the development of the template mechanism that allows the user to not only add support for new function blocks, but also to add support for almost any high level language that provides infix Boolean expressions.

8.3 Areas for further work

8.3.1 LEd performance

At present, generating the net-list can take several seconds for a ladder diagram with a handful of components. Since the ladder diagram as saved on disk contains the net-list, this performance penalty is seen every time a file is saved.

8.3.2 LEd editing features

The LEd editor is missing a number of editing features that users have come to expect in this type of application:

- Copy/Paste objects and regions.
- Undo
- Multiple open diagrams

8.3.3 Template files

There exists a very real danger of collisions between the user-entered device names and the internal variable names used in some of the device templates (the counter for example). Such name-space pollution could easily be avoided by use of a standard for internal vs. external names (something similar to the leading underscore convention in C programming). The current set of template files is suitable for demonstration purposes, but the template files would probably need to be more carefully evaluated and re-designed for a production environment.

8.3.4 Extensibility

The user may add new function block types by editing a text file containing Scheme data structures. This file contains information about how each function block is to appear and where the devices terminals are located. This task would be much easier if a function-block editor was developed.

8.4 Conclusion

The project as a whole was a success. While it lacks a few features that might be needed in a production environment, it does demonstrate the feasibility of automatically generating source code from ladder diagrams. After more

extensive testing and a few enhancements, the tools developed during the project could easily be used as a practical way to generate source code for discrete control applications that aren't suitable for a PLC.

References

W. Clinger, J. Reese (eds), "Revised⁴ Report on the Algorithmic Language Scheme," ACM Lisp Pointers IV, July-September 1991 (also available as MIT AI Memo 848b).

R. Devanathan, Foo Yung Kuan, Chang Chiew Jun, Choo Siew Aun, "Computer aided design of relay ladder logic via state transition diagram", IEEE Proceedings IECON '87, 764-772 (1987).

E. Gallezio, "STk Reference Manual", Université de Nice – Sophia Antipolis, France (1996)
URL=<ftp://kaolin.unice.fr/pub/STk-doc.tar.gz>

IEEE Standard 1178-1990, "IEEE Standard for the Scheme Programming Language", IEEE, Piscataway (1991).

IEC Standard 1131-3: "Programmable controllers - Part 3: Programming languages." International Electrotechnical Commission (1993). (also available as ANSI/NEMA IA 2.3-1993)

A Krigman, "Relay ladder diagrams: We love them, We love them not", InTech, 32 (10), 39-44 (1985).

J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesely, Reading Mass (1994).

LEd Man Page

NAME

LEd - A Ladder Diagram Editor

SYNOPSIS

```
led [filename]
stk [options] -f led [filename]
```

OPTIONS

led itself accepts no options; only an optional filename argument specifying a ladder diagram file to be initially opened. See the stk man page for stk options available when running led.

DESCRIPTION

led is a graphical ladder diagram editor running under STk. Stk is a Scheme R4RS interpreter which provide a simple access to the X11 Tk toolkit.

If you have problems running led, make sure that both the STk and LEd libraries are installed. If the programs have not been properly installed, you may need to set the library path environment variables.

If LEd still won't run, browse through the help files (particulary the hints section) by pointing your favorite HTML browser at the file \$LED_LIBRARY/help/index.html.

ENVIRONMENT VARIABLES

led and stk use the following shell variables:

STK_LIBRARY This variable indicates where the STk library files are located. This variable allows to overload the default value of the Scheme variable *stk-library* which is automatically calculated by the interpreter.(i.e. stk or snow).

LED_LIBRARY This variable indicates where the LEd library files are located.

FILES**SEE ALSO**

stk(1)

LEd Hypertext Help Documents

Help - Index

Help is available on the following topics

- About LEd
 - Overview
 - Menubar
 - File
 - View
 - Options
 - Help
 - Toolbar
 - Object Types
 - Relay Contacts
 - Labels
 - Function Blocks
 - Placing New Objects
 - Naming Objects
 - Wiring Connections
 - Deleting Objects
 - Moving Objects
 - Code Generation
 - References to "node-XX" in Code
 - Hints
 - Error loading file
 - Unrecognized bitmap data
 - Unbound variable: option
-

About

The Project

The LEd ladder diagram editor was written as part of a project submitted to the Graduate College of the University of Minnesota to fulfil the requirements for a MS in Computer Science. Together with the accompanying code generation program, *lad*, LEd provides a way to automatically generate C source code from a graphical representation of Boolean logic based system requirements.

Development Tools

LEd is written in Scheme using STk 3.0, a scheme interpreter linked with the Tk X11 user interface toolkit. The *lad* program is written in ANSI C. The author's editor-of-choice for most of the project was the *jed* editor with a home-made lisp mode added.

Platform

The vast majority of the development was performed on a 25MHz Intel 80486 machine running the Linux operating system and X11 (R5 and R6). LEd has also been tested under SunOS 4.1 running X11R5 and Solaris 2.[345] running Openwindows. It should run on any *nix machine upon which STk 3.0 runs. The *lad* program has also been run under VMS and MS-DOS.

Many Thanks To:

- **Erick Gallezio** *eg@unice.fr*
The author of the STk Scheme package.
- **Larry Ousterhout**
And the rest of the people who developed the Tk toolkit.
- **John E. Davis** *davis@amy.tch.harvard.edu*
The author of the JED text editor with which most of the source code was created.
- **Linus Torvalds** *Linus.Torvalds@Helsinki.FI*
And the rest of the authors of the Linux operating system, utilities, distributions, and the Linux community in general.
- **The XFree86 Project, Inc.**
For providing a complete, GPL'd X11R6 package for Linux and other Intel based *nix systems. Thanks also to the X Consortium, Inc. for the base X11 system.
- **The Free Software Foundation**
For gcc, RCS, groff, and the countless other gnu tools without which none of the free *nix systems would exist.
- **My Employers,**
FMC Naval Systems Division and Rosemount, for paying my tuition.

Copyright

Except where noted in the source files, all software is

Copyright (c) 1996, Grant Edwards, grante@winternet.com

Permission to use, copy, and/or distribute this software and its documentation for any purpose and without fee is hereby granted, provided that both the above copyright notice and this permission notice appear in all copies and derived works. Fees for distribution or use of this software or derived works may only be charged with express written permission of the copyright holder. This software is provided “as is” without express or implied warranty.

Grant Edwards
grante@winternet.com
grante@rosemount.com
edwards@grad.cs.umn.edu

Help - Overview

Purpose

LEd is a ladder diagram editor. It allows the user to draw schematic diagrams that represent boolean system requirements. LEd can generate a netlist for the entered diagram and then call an external code generation program that will read that netlist and generate source code that implements the boolean logic represented by the ladder diagram.

Invokation

The LEd program is normally invoked by typing the following at the shell prompt.

```
$ led [filename]
```

If that doesn't work, try the following

```
$ stk -f [path]led [filename]
```

If that doesn't work, then it's probably that the STk scheme interpreter isn't installed. Try running STk alone:

```
$ stk
```

If that doesn't work, you need to install STk.

Operation

The user interacts with the program via the menubar located horizontally across the top of the application window and via the toolbar located vertically along the left side of the application window.

The menubar allows the user to do file operations, set program options, view help documents, and run the code generation facility.

The toolbar is used to edit the ladder diagram. It allows the user to place new objects, edit the names of objects, move/delete objects, and make connections between objects.

Help - Menubar

The menubar across the top edge of the application window provides the user with most of the functions not associated with direct manipulation of the ladder diagram. It contains four drop-down menus: the File, View, Options, and Help menus.

File

The File menu contains the following entries:

Open

A file selector dialog box is created to allow the user to select the ladder diagram file to be loaded. The current ladder diagram is erased before the selected file is loaded.

New

The current ladder diagram is erased.

Save

The current ladder diagram is saved in the current filename (shown at the right hand end of the toolbar).

Save As

A file selector dialog box is created to allow the user to specify the filename in which the current ladder diagram is to be saved. The specified name then becomes the current filename (shown at the right hand end of the toolbar).

Print

A printer dialog box is created to allow the user to select either the print command or the file name to which the postscript output is to be sent. The print command entered must accept postscript input on stdin (for example: "lpr -Pps" on many Unix systems will send data from stdin to a printer named "ps").

Close

Exits the LEd program.



You will not be warned if the drawing is unsaved.

View

The view menu contains entries that will pop up a text viewing window containing various interesting or

not-so-interesting information. The things that can be viewed are:

Object List

Will display a list of all of the "objects" in the current ladder diagram. Each line of the displayed text has the format

```
("obj-6" "xyz" "normopen")
```

where "obj-6" is a unique, internally generated label, "xyz" is the object name that the user entered, and "normopen" is the object type.

Net List

Will display a netlist showing the interconnections present in the current ladder diagram. Each line of the displayed text has the format

```
(node-7 (("obj-5" 1)("obj-4" 3)("obj-15" 1)))
```

where node-7 is the node name, and the rest of the line is a set of object/terminal specifications. In the above example, terminal 1 of obj-5, terminal 3 of obj-4 and terminal 1 of obj 15 are all connected and that node has been given a unique, internally generated label of node-7.

If the node name is not of the form node-99, but rather is a string enclosed in double-quotes, then the node name was taken from the user-entered name of a "label" device that was placed on the ladder diagram by the user. An example of such a line would be

```
("START" (("obj-5 0)("obj-1" 0)))
```

Formatted/Merged

Will display the object list and net list as it will be sent to the code generate program. This information consists of two separate sections which are separated by a single blank line. These two sections contain the same information (formatted slightly differently) as is displayed by the Object List and Net List menu entries.

The first section is object list, each line of which looks like

```
obj-5 xyz normclosed
```

where obj-5 xyz and normclosed are the object name, user entered value, and object type.

The section section is the net list, each line of which looks like

```
node-25 obj-6 1 obj-7 1 obj-8 0
```

where node-25 is the node name and the remainder of the line consist of object/terminal specifiers.

Generated Code

Will pass the formatted/merged object table and netlist to the code generation utility and display the generated source code.

Options

The options menu contains entries that allow the user to modify certain features and behaviors of the editor. These settings are stored with the diagram when it is saved to disk and restored from the file when a diagram is loaded from disk. The options are

Drag Bounding Box

This option determines how the editor will visually represent objects when they are being moved as part of a group move. If this option is enabled the objects being moved will be represented by a bounding box drawn around the selected objects. If this option is disabled, representations of the objects themselves will be dragged.

Line Width

This option determines the line width used to draw the ladder diagrams. It may be set to 0, 1, or 2. A line width of 0 will allow the X server to use a fast, machine-dependent line drawing algorithm. However, lines with a width of 0 may not behave in a portable fashion. For example where two lines meet, there may or may not be a small gap. Lines of width 1 or 2 will always behave in a predictable, albeit somewhat slower, manner.

Drawing Size

This option determines the size of the drawable area available. It may be set to five different values A,B,C,D or E. The A size is portrait mode, the others are landscape mode. The postscript output does not do any scaling or translation depending on drawing size. It is assumed that the postscript will be printed to queue or device that will correctly render the drawing.

Help

The help menu contains entries that will invoke a viewer that will allow the user to browse through accompanying hypertext documents. The entries in the help menu are

Help

This entry will display an index to the hypertext help documents.

About LEd

This entry will display a document containing miscellaneous information that does not pertain to the actual operation of the program (copyright, author, etc).

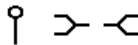
Help - Toolbar

The toolbar along the left edge of the application window provides the functions that are used to create and manipulate ladder diagrams. The buttons perform the following basic functions:


- The top seven buttons (from the top button through the function block button) are used to place new objects of various types on the drawing.
 - The wire button is used to interconnect objects.
 - The text button is used to edit the name associated with an object.
 - The move and delete buttons are used to move and delete either a single object or a group of objects within a rectangular region.
-

Object Types

There are four classes of objects that can be placed in the drawing area

 Labels


All of the label objects perform the exact same function. They assign their user-entered name to the node to which they are connected. The only differences between the label types are the physical location of the name text and the appearance.

 Contacts

A relay contact represents a single boolean value with the name given by the user entered object name. The normally open contacts (without the slash) represent the non-inverted value. The normally closed relay contacts (with the diagonal slash) represent the inverted value.

The output (bottom terminal) of a normally open contact is true IFF the input (top terminal) is true AND the named value is true.

The output of a normally closed contact is true IFF the input is true AND the named value is false.

 Function Blocks

Function blocks are similar to contacts in that the output (bottom terminal) is true iff the input (top terminal) is true AND the value of the function block is true.

While contacts represent the state of a single boolean value, a function block's value is controlled by one or more inputs located on the sides of the block. A function block may also have internal

state information so that it may count events and remember it's previous states.

Outputs

An output object assigns the value on it's input (top terminal) to the user-entered name.

Placing a New Object

To place any of the above objects on the diagram, left-click on the button showing the desired object. In the case of the function block button, a menu will pop up showing the various function block types -- select a type by left-clicking.

Move the pointer into the drawing area. A new object (colored red) will be created and will follow the pointer as it is moved. Another right-click will place the new object at it's current location on the diagram. A right-click will discard the new object without placing it on the diagram.

Naming an Object

To edit the name of any of the above objects, click on the text button (the one with the large T) and then click on either the object itself or the name. A text cursor will appear and you will be allowed to edit the object's name.

Wiring Connections

To wire connections between devices

- Left-click on the wire button.
- Move the pionter to the point where a wire is to begin.
- Left-click to start the wire.
- Move the pointer and a wirepair will follow it. The first wire will always be in the directly the pointer is first moved. The second wire will be orthogonal to the first. The mouse buttons are now defined so that
 - Left-click will place the wire pair and begin another wire pair from that spot.
 - Middle-click will place the wire pair without beginning another.
 - Center-click will discar the wire pair without placing it.

Two wires that cross or meet in any way are considered connected. A wire must meet or cross the endpoint of a device's terminal to be connected.

Moving Objects

Objects may be moved individually or in groups.

To move an individual object

- Left-click on the move button
- Left-click on the object to be moved.
- The object will change colors from black to red and move as the pointer is moved.
 - Left-click will place the object.
 - Right-click will return the object to it's original position.

To move a group of objects

- Left-click on the move button
 - Center-click at one corner of a rectangular region containing the objects to be moved.
 - A rubber-band box will follow the pointer as it is moved.
 - Right-click will remove the rubber-band box without selecting any objects.
 - Center-click will select the objects that are entirely contained within the rubber-band box.
 - Once the objects are selected they will either change color from red to black or a bounding box will appear (depending on the setting in the Options menu).
 - The selected objects (or the bounding box) will now follow the pointer as it moves.
 - Left-click will place the objects.
 - Right-click will return the objects to their original positions.
-

Deleting Objects

DELETE


Objects may be deleted individually or in groups.

To delete an individual object

- Left-click on the move button
- Left-click on the object to be deleted.

To delete a group of objects

- Left-click on the move button
- Center-click at one corner of a rectangular region containing the objects to be deleted.
- A rubber-band box will follow the pointer as it is moved.
 - Right-click will remove the rubber-band box without deleting any objects.
 - Center-click will delete the objects that are entirely contained within the rubber-band box.

 **There is no undelete.**

Help - Code Generation

References to "node-XX" in Generated Code

This occurs when an object's input is not connected to anything.

All inputs must be connected to something. If there is an input that you don't want to use (for example the "clear" input on a counter) then attach a label to that input and give it a value of "0".

Help - Hints

Explanations/excuses for miscellaneous things that might go wrong.

Error messages about loading files

If you get an error message like this

```
*** Error at line 20 of file ./led.stk:
load: cannot open file: "fileselect"
```

then STk couldn't find one of the library files. These files exist in two places: the standard STk library directory and the led library directory.

If the message complains about files named `fileselect`, `textview`, `strcmd`, `printdialog`, `netlist`, or `htmlviewer`, then it is probably the led library that can't be found. If it complains about other files like `dialog`, then it is probably the standard STk library that can't be found.

STk Library

By default STk will look for its library files in a location based on the location of its binary. If this isn't working (or STk wasn't installed properly) the location of STk library can be set explicitly by setting the environment variable `STK_LIBRARY` to the pathname of the directory containing the STk library files. The usual place these files get installed is `/usr/local/lib/stk/[version]/STk`.

LEd Library

By default LEd will look for its library files in the location `/usr/local/lib/led`. If the library is installed elsewhere, you will need to set the environment variable `LED_LIBRARY` to the pathname of the LEd library directory.

Unrecognized bitmap data

The help viewer occasionally doesn't seem to recognize an X11 bitmap file as such. Try hitting reload. If that doesn't work, ask for your money back.

Unbound variable: option

Example error message:

```
*** Error at line 13 of file [path]/ fileselect.stk:  
    unbound variable: option  
Current eval stack:  
-----  
0      (option (quote add) (quote *listbox*font)  
[...]
```

This happens when you try to run led using the non-X11 version of the scheme interpreter. This probably happened because you didn't have the DISPLAY environment variable set properly. Either set (and export) the DISPLAY environment variable, or provide the -display flag to stk:

```
$ stk -display hostname:0 -f [path]led.stk
```

lad Man Page

lad(1)

lad(1)

NAME

lad - Ladder Diagram Code Generator

SYNOPSIS

lad [options] [filename]

OPTIONS

lad accepts a number of command line options that may be used to modify the default code generation behavior. The optional argument specifies an object-table/net-list file to be processed. If no filename argument is present, input will be read from standard input.

-d level	Set debug level (0:none 9:too much) (default 0)
-i	Enable auto-indent of boolean expressions
-c	Enable case-insensitive handling of names
-t tabsize	Set tab size for auto-indent (default 3)
-n op	Not operator string (prefix) (default '!')
-a op	And operator string (infix) (default '&&')
-o op	Or operator string (infix) (default ' ')
-r fmt	Value reference format (printf) (default '%s')
-l path	Library path for template files (default '.')
-f name	Function name in output code (default '[input-filename]')

DESCRIPTION

lad is a post-processor that will generate program source code from information in ladder diagrams as produced by the LED and ladformat programs.

ENVIRONMENT VARIABLES

April_1996

1.0

1

FILES

Template files are expected to be found either in the current directory or in the path specified by the `-l` command line option. The overall template used to generate output is called `file.template`. Other template files have names of the form `blockname.S` where `S` is a single character suffix determined by the `file.template`.

There are two types of template files: a file template and a set of block templates. The processing of template files is handled much as if they were format strings for the ubiquitous `printf()` family of library functions. The template file is read in and copied to the output with `%X` character sequences replaced by various formatted output. The set of output specifiers, `X`, depends on the particular type of template file being processed.

FILE TEMPLATE

The file template is used once per invocation of `lad` and controls the format of the output file. The results of the output specifiers available for use in the file template are shown below.

- `%f` Output the `''function''` name (this defaults to the name of the input file, but may be overridden with a command-line option).
- `%S` Output the results of processing (for each function block in the net list) the block template with the suffix `S`. `S` may be any single character other than `'f'`.

BLOCK TEMPLATES

Each type of function block used in the ladder diagram must have a block template file for each output specifier type other than `f` that is present in the file template. All block templates are processed identically, and the output specifiers that are available for use in block template files are shown below.

- `%n` Output the device `''name''` (this is the name the user entered using the LED diagram editor).
- `%N` Output the boolean expression that evaluates to the value present on the function block's control terminal `N` (where `N` is a value from 2 through 9).

SEE ALSO

`led(1)` `ladformat(1)`

ladformat Man Page

NAME

ladformat - Ladder Diagram Netlist Formatter

SYNOPSIS

```
ladformat filename
snow [options] -f ladformat filename
```

OPTIONS

ladformat does not support any command line options other than the required filename argument.

DESCRIPTION

ladformat is a utility that reads a ladder diagram file as produced by LEd and produces an object table and netlist in the format required by the lad code generation program.

ladformat is written in Scheme and requires the STk package to run. It does not use any X11 user interface features and may be run using the snow interpreter rather than the stk interpreter.

The formatted object table and netlist are written to standard output.

ENVIRONMENT VARIABLES

ladformat and snow use the following shell variables:

STK_LIBRARY This variable indicates where the STk library files are located. This variable allows to overload the default value of the Scheme variable *stk-library* which is automatically calculated by the interpreter.(i.e. stk or snow).

LED_LIBRARY This variable indicates where the LEd library files are located. These files are used by ladformat to read and process the ladder diagram file.

FILES**SEE ALSO**

led(1) lad(1)

Toshiba EX14B Specifications

Toshiba EX14B PLC Specifications

- Programmed using ladder diagrams. Programs may be edited from local 8x10 LCD display or they may be downloaded via serial port.
- In addition to relay logic, instructions are included for timers (56), up/down counters (62), flip-flops, and bidirectional shift registers.
- Basic I/O consists of 8 inputs and 6 outputs. I/O may be expanded to include up to 32 points (20 input, 14 output).
- Inputs types: 120VAC or 24VDC (dry contact input available).
- Output types: Relay (max load 2A resistive, 1A inductive)
- Optional analog inputs (2) available.
- Internal relays: 128 retentive, 112 non-retentive, 16 special purpose.
- Dimensions: 235 x 180 x 40mm
- Weight: 1.5kg