

LOGO! PLC

Learning a Programmable Logic Controller

by
Francisco Glover, S.J.

**Ateneo de Davao University
Davao City**

May, 2008

Foreword

This manual focuses on a particular PLC brand and model. Even a limited search on the Internet shows the variety of PLC manufacturers and models. Siemens is said to be the world's top supplier of PLC hardware but the LOGO! appears to have minimal sales compared to their other lines. Some might characterize the LOGO! line as mere toys. Why, then, center this manual on LOGO! ?

The author is in academe, not industry, yet we aim at preparing our Engineering students in the best possible manner for their future industrial career, and PLCs are major players for industrial processing. Since programmability is the unique attraction of the PLC, it is the point of emphasis in student training. What software (each manufacturer has its own) is apt to offer the most gentle *learning curve* for students, and what companion hardware is the least expensive, to permit adequate units for student hands-on use? After taking a close look at the brands in use in the area, namely Siemen's *Simantic* and LOGO!, Schneider's *Twido* and Yokogawa's approach, it seemed that LOGO! is the best suited for introductory school use. The LOGO! software we judge to be outstanding, and the hardware is comparatively inexpensive.

LOGO!Soft Comfort[®] is the proprietary software for all LOGO! brand units. Perhaps it's most attractive feature is complete *off-line simulation* of the control program. This ability is such an attractive feature that the manufacturer provides a free download of the complete version, omitting only the ability to transfer the program to hardware (www.siemens.com) . A second notable feature of the software is the ease of creating *function block* or *ladder* diagrams using the appropriate graphical interface. The software provides a default complete identification label for every icon introduced, a feature which makes the programming quick and easy. Also the context-sensitive help is extensive and reasonably complete.

However, like in most other things there are shortcomings. Eight inputs and four outputs on the base unit (without additional hardware) makes LOGO! unsuitable for any practical applications other than the most simple. While it provides program translation between *ladder* and *function block* formats, no references are given for *instruction list* or *assembly language* programming. It appears that the software was created in German and the English translation of instructions and nomenclature is at times rather obscure.

While the various LOGO! models are comparatively inexpensive, none can be programmed without a *connecting cable* for program download. This cable contains nothing but a pair of opto-isolators and level-matching between RS232 and TTL (which could have easily been included within the main unit) yet the cost of this cable is approximately 80% that of the PLC unit with which it communicates!.

Of the eight available LOGO! PLC units the 12/24 RCo model has the combination of relay output, real-time clock, and the capability of two analog inputs and two high-speed counting inputs. For this reason the 12/24 RCo is the focus of this manual.

Could one hand to a foreign student an English grammar and dictionary and expect him with these alone to become the new James Joyce or Ernest Hemmingway of our times? LOGO!Soft Comfort® does provide detailed syntax on all functions and also gives an introductory tutorial for one's first program, but much more seems to be required to form a competent programmer. And this is the area that this manual proposed to fill. Not all the features of the software are discussed, not all the Special Functions are mentioned or explained. What is included is, hopefully, sufficient for a one-semester course for senior Engineering students.

To whom might LOGO! PLC be taught? At the author's institution this includes majors in *Electrical Engineering*, *Computer Engineering* and *Electronic & Communication Engineering*. The course seems most relevant to *Electrical Engineering* but only marginally so to the other concentrations. The restriction imposed by a Scan time of several tens of milliseconds severely limits the PLC application. Student's interest and need do affect the emphasis in presentation. LOGO!Soft Comfort® offers programming and simulation in both *function block* and *ladder* diagrams, However *ladder diagrams* are almost completely unknown in the *computer*, *communication* and *electronic* fields, in which the function block formulation is quite prevalent. Consequently the present manual gives primary emphasis to the function block approach. In practice this means that *ladder diagrams* of more than a dozen rungs are not reproduced in the text since their full display is readily available in the software package.

Where might a PLC be most appropriately used? It was designed to replace wire technology for the control of plant operations. But outside this area the elements of Scan time, physical size and price make it a poor second to micro controllers such as the PIC family. For timing the PIC has an advantage of the order of 10^5 . The package size excludes such applications as a personal blood-pressure heart-beat monitor or in a walking-talking Teddy Bear. Locally the cheapest bare PLC unit retails at about P15,000 compared to a low-end PLC of comparable functionality at P300. However within its own field of application it remains an excellent tool.

The author is deeply indebted to Engr. Jenith Banluta who has carefully read the entire manuscript and has offered very many helpful suggestions. Likewise acknowledged is the work of Engr. Reyman Zamora for the design of the bare LOGO! unit *mounting* and the auxiliary *linear motion module*.

Francisco Glover, S.J.
Ateneo de Davao University

Table of Contents

Chapter 1: Basic ideas

1.1	Once upon a time...	1
1.2	A system to be controlled	2
1.3	Wire Technology	2
1.4	Enter the PLC	3
1.5	What's Inside	4
1.6	The Scanning Cycle	4
1.7	Programming Languages	5
1.8	Ladder Diagram [LAD]	7
1.9	Function Block Diagram [FBD]	7
1.10	Instruction List [IL]	8
1.11	Structured Text [ST]	8

Chapter 2: Getting started

2.1	Making a choice	11
2.2	Drawing pictures	11
2.3	Closing and saving	15
2.4	Help! Help!	15
2.5	Ladder Diagrams	16
2.6	As simple as possible	18
2.7	Language Conversion	19
2.8	Program simulation	20
2.9	Printing your circuits	20

Chapter 3: ANDs and ORs

3.1	Switches and Relays	24
3.2	Memory Overview	25
3.3	Basic programming rules	26
3.4	Three IN one OUT...	26
3.5	Running in hardware	31
3.6	Multiplex and de-Multiplex	32

Chapter 4: Remembering...

4.1	Green START, red STOP	35
4.2	Set and Reset	36
4.3	Flags	38
4.4	The inverted coil icon, \neg (/)	38
4.5	Our first function	39
4.6	Create a memory cell	41
4.7	Memory matters	43

Chapter 5: Limits	
5.1 Limits left and right...	46
5.2 Automatic control	47
5.3 The Up / Down Counter	48
5.4 The Automatic Button–Pusher	51
5.5 Back to Pizza	51
Chapter 6: Time and Timers	
6.1 Classifying Timing Functions	54
6.2 Push On – Push Off	56
6.3 Time Delays	57
6.4 Edge-triggered, fixed delay, fixed duration	58
6.5 One Full Process	60
6.6 Turn Indicator	61
6.7 Motor overload protection	62
6.8 A simulated random overload	64
6.9 Random Pulse Generator	65
Chapter 7: Some Odd Functions	
7.1 Pulses per time interval	68
7.2 The duration of the <i>Scanning Cycle</i>	69
7.3 Longer time intervals	70
7.4 The Shift Register	72
7.5 The Shift Register as a rotary switch	74
Chapter 8: Digital Control Circuits	
8.1 2–station Lamp Control	78
8.2 Paging system	79
8.3 Linear motion	81
8.4 Stepper motors	84
Chapter 9: Analog	
9.1 Analog Ins and Outs	89
9.2 Sensors, Digital or Analog	90
9.3 The <i>Analog Amplifier</i> Special Function	91
9.4 Analog Limits	92
9.5 More of the same	94
9.6 Monitoring a value	95
9.7 Analog watchdog	96
9.8 Analog Comparator	97
9.9 Analog multiplexer	98
Appendix	102

LOGO! PLC

Chapter 1: Basic ideas

1.1 Once upon a time...

A long time ago when life was more simple than it is today, you might go out from your cave in the morning, trusty stone-axe in hand, to hunt a tiger for supper. Your mighty arm and eagle eye would have *controlled* the stone axe, a machine of sorts, to produce an effect on the tiger's head. At a later age you might place your hand on the pump-handle at the barrow well to draw water for a morning bath; the pump is the instrument or machine that you use to raise the water; your hand motion *controls* the flow of water. And today you may just push a button on the base of an electric fan to cool yourself on a hot afternoon. But unlike the pump-handle where you do the work, your pushing the button *controls* the flow of electricity that uses the fan to produce its pleasant effect. In each case you **control** a machine to produce some desired effect.

Normally the buttons of the fan are in the base, below the fan motor. It is the power cord (not shown in Fig. 1.1) that provides the electricity. The *off*, *low*, *medium*, and *high* buttons only *control* the flow of power to the fan motor. This simple control system is typical of what this book is all about. Here the fan motor is comparatively small and does not draw much current from the power source. But industrial applications use motors drawing quite large currents, too large to be conveniently connected directly to the control system box, so a relay is used. The control circuit (composed of the push button) controls the relay and the relay in turn controls the heavy current through the motor as shown in Fig. 1.2.

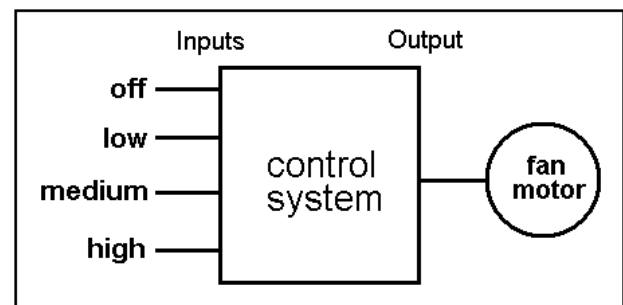


Fig 1.1 A simple control system

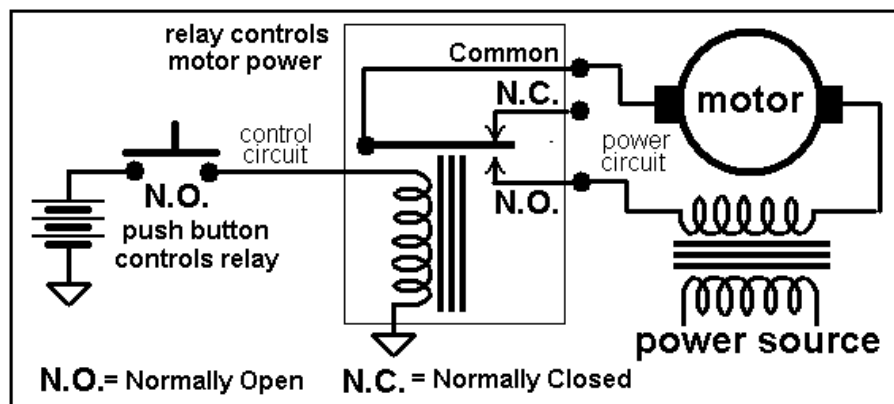


Fig. 1.2 A relay used to control a large current

When the *normally open*, **N.O.**, button is pushed, the control circuit energizes the relay, closing and allowing the **N.O.** relay contacts to complete the motor circuit. Actually this circuit is not ideal, for who wants to keep a finger on a button just to keep the motor running. More on this later.

1.2 A system to be controlled

Consider driving an automobile. The driver is the *controller*. The *inputs* are what the driver sees, hears and feels. The *outputs* are the driver's hands on the wheel and feet on the pedals. The *machine* that is controlled is the vehicle itself. But it is also possible to have a robot car where the inputs to the remote driver and the actual manipulation of the vehicle are all done through a radio link. The driver still controls the vehicle but now from a distance.

An experienced driver knows how to respond to any situation; swerve to the right to avoid an oncoming truck, slow down to avoid a pedestrian, pull over at the sound of a siren. But how about putting all this knowledge into another machine, an *automatic controller*, to which all the inputs and outputs are connected? Then the automatic controller could drive the automobile, day and night, freeing the human driver to do something more productive or more enjoyable. This is exactly what has happened during the last century, in automobile productions, oil refineries, steel mills, power-generating stations. The automatic controller, to which all the input and output lines are connected, was a large collection of relays, timers, counters all wired together in just the right way to produce the desired results. The art and science of connecting together all these interacting units was referred to as **wire technology**.

1.3 Wire Technology

A particularly simple example of *wire technology* is an automatic home washing machine. The human puts in the dirty clothing, adds the soap powder or detergent and then sets the dial. The cams and levers of an internal rotating drum connect to valves, pumps, temperature sensors, interlocks and motors. This rotating drum sequences the whole operation from start to finish, adding and draining water, tossing the contents, finally spinning them dry. In a large industrial operation the controls are neatly arranged in one or more steel cabinets, to which all the input and output line are connected. Inside the different relays, timers, counters and other components are neatly arranged along a series of horizontal rails, like the rungs of a ladder, for easy maintenance and service.

All this was a great advance but there was still room for improvement. When production requirements changed so did the control system. This became very expensive if changes were frequent. Since relays are mechanical devices they also have a limited lifetime which requires strict adherence to maintenance schedules.

Troubleshooting was also quite tedious with so many relays involved. Now picture a machine control panel that included many, possibly hundreds or thousands, of individual relays. The size could be mind boggling. The design and initial wiring of so many individual devices was a major operation. All the components would be individually wired together in just the right manner to yield the desired outcome. And if changes were needed in the machine operations, some or all of the control wiring would have to be redone.

Digital computers were developed in university laboratories during the 1940s, and began to appear in business and industry during the 1950s. Those installations normally occupied an entire large room. With the advent of the transistor and then the integrated circuit, the computers of the 1960s were dramatically decreasing in size yet increasing in capability. It was then realized that one small computer might replace the entire array of relays, timers and counters wired together inside control cabinet. When production changes are needed, simply change the computer program, the *software*, while the external input and output lines, the *hardware*, remain the same.

A small American company, Bedford Associates, proposed to a major U. S. automobile manufacturer something called a Modular Digital Controller to replace the entire wire technology inside the control cabinets. This was the start of what we now know as a **Programmable Logic Controller**, a **PLC**.

1.4 Enter the PLC

These new controllers had to be easily programmed by maintenance and plant engineers. The lifetime had to be long and programming changes easily performed. They also had to survive the harsh industrial environment. To this single PLC, ultimately made no larger than a single conventional relay, all the input and output lines are attached, so that it could sit alone on a single rail or rung within the cabinet, with all the other hardware removed. The mounting attachment of today's PLC units is exactly what was the standard for all relays, timers and counters of *wire technology*. The older mechanical parts were replaced by solid-state technology.

In principle almost any computer, IBM or Macintosh, could be used as a PLC. The original IBM Personal Computer had a thousand ports, each of which could be configured as an *input* or *output*. However keyboards, disk drives, monitors and disk drives are not needed if only PLC functions are required. For this reason the central processing unit, CPU, of the PLC is specifically designed to optimize the desired functions.

1.5 What's Inside

The innards of all PLCs are basically the same, as shown in Fig. 1.3 . The *Data memory* is equivalent to the EPROM of a PC, the *Scratch-pad memory* equivalent to RAM. The contents of *Program memory* may come from an external computer, or from an inserted *Flash memory* chip . However, unlike the usual PC, the input levels may be 12/24 VDC, or even 230 VAC/DC . The output ports may be 12/24 VDC, or serve as the **N.O.** or **N.C.** terminals of an internal solid-state relay. Some models have input and output ports adapted to *analog signal levels* in place of the more common **ON** or **OFF** logic levels.

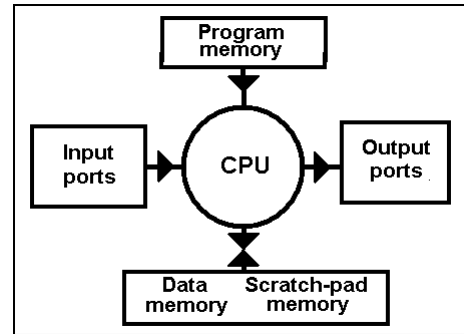


Fig 1.3 A typical PCL

Since the basic PLC unit comes in a small package (typically some nine centimeters, top to bottom, when mounted on the rail) there is the practical limit to the number of input and output connectors in this small space. Therefore *expansion modules* may be mounted side-by-side with the basic unit to provide additional *input* and *output* terminals. Higher-level individual PLC units can be wired together to permit parallel processing or communication between remote units. Normally the greater the capability of the PLC model, the greater the cost. Yet even the simplest unit can do amazing things.

1.6 The Scanning Cycle

A PLC operates by moving sequentially through a continuous loop, the *scanning cycle*, consisting of three steps, as suggested in Fig. 1.4.

Step 1: Examine Inputs First the PLC takes a look at each input to determine if it is **ON** or **OFF**. In other words, is the sensor connected to the first input **ON**? How about the second input? How about the third... It records this information in its *Data memory* to be used during the followingt step.

Step 2: Execute Program Next the PLC executes one instruction at a time the program stored in *Program memory*. Suppose the first program instruction states that if the first input is **ON** then the first output should also be **ON**. Since the states of all *inputs* were recorded in the first step, the program decides if the *first* output should be turned **ON** or **OFF**. However in this step it only stores this information but does not change the *output*. So it continues, instruction by instruction, until the end of the program.

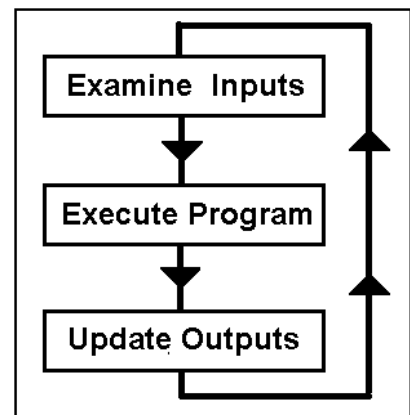


Fig. 1.4 Scanning cycle

Step 3: Update Outputs Finally the PLC simultaneously updates all the *outputs* based on the state of all *inputs* found in the first step and the results of executing the program during the second step

After the Step 3 the PLC goes back to Step 1 and repeats the steps continuously. Actually there is some housekeeping done at the start of Step 1, just before the inputs are read. One **scan time** is defined as the time it takes to execute the 3 steps listed above. Of course the *scan time* depends on the clock speed or crystal used in the particular unit. The number of *inputs* to be checked in each Step 1 and the number of *outputs* to be updated in each Step 3 have little effect of the *scan time*. The number of program instructions is the most significant factor in setting the *scan time*. Later on we will attempt to measure the scan time for a particular PLC and program.

Another thing to notice in this sequential processing is that although a scan time may be of the order of milliseconds, all inputs are read and all outputs updated in a matter of microseconds, so that *to a first approximation* we may consider *all* inputs to be read at the same time and likewise all *outputs* updated at the same time. This also means that any input change during Step 2 will not be recognized until the next scan. These points must be considered in situations where timing is critical.

You may wonder if an array of TTL chips or a *PIC* or *ATMEL* micro-controller could do the work of a PLC, perhaps at a much lower initial cost. In principal the answer is YES but physical size and shape are also a consideration. The PLC is in a durable mounting adapted to traditional control equipment. To control the reactions of a stuffed toy teddy bear a PIC is surely a better choice than a PLC. And while a suitable array of TTL or CMOS integrated logic circuits could duplicate any PLC program, even a slight change in the system operation would mean a reworking of the chip configurations, the same problem met in the older hard-wired systems replaced by a PLC. The moral is to select the hardware that is most suited to the control system at hand.

1.7 Programming Languages

In the early days of radio manufacture, the *schematic diagram* was introduced, a standard manner of describing the internal connections by means a uniform set of symbols with interconnected lines, representing the actual configuration of resistors, capacitors, tubes and switches. Once you recognize each symbol and follow the interconnecting lines you have a complete understanding of the entire circuit.

When integrated circuits or “chips” were developed, another new set of symbols were introduced, the AND, OR and NOT gates, with inputs and outputs which could be interconnected in almost endless ways. The symbols and connecting lines tell the whole story. Likewise *wire technology* developed its own set of standard symbols representing relay contacts and coils, switches and timers to describe the entire control system.

The PLC is basically a small digital computer dedicated to the performance of a limited set of operations. As a computer it operates on instructions in the form of bits

and bites, its own **machine language**. Unfortunately, machine language is now really “user friendly” to our human way of thinking. For personal computer programming *flow charts* and *high level languages* were invented to bridge the gap between humans and machines.. For PLC programming two graphical were developed, one based on the traditional *ladder diagrams* of wire technology, the other based of the logic gates and function blocks of integrated circuits, the equivalent of the computer *flow chart* .

In this manual we develop PLC programs with both the ladder diagram [**LAD**] approach and also with the gates and function blocks diagram [**FBD**] approach.. The software program we use to do the translating. is **LOGO!Soft Comfort**.

All present manufactures of PLC units have agreed to implement a certain *standard set of operations* (relating *outputs* with *inputs*) but each manufacturer uses its own *machine language*, the bits and bytes, needed to execute on their product each standard function. This is true of almost all computer CPUs (Central Processing Units) The machine language of the CPU found in an IBM PC is quite different from that in a Macintosh. Since no programmer in his right mind wants to memorize the bits and bytes of machine language, PLC manufactures and professional programming groups commonly use four different approaches or *languages* for programming the PLC.

A real-world circuit is shown in Fig. 1.5, consisting on a series connection of a *normally open* push-button **A** and a *normally closed* push-button **B** to control a small motor **C**.. To use a PLC to control the motor, the physical connections would be as shown in Fig. 1.6 In practice you would not use a PLC for such a simple circuit; here it is used only as an illustration.

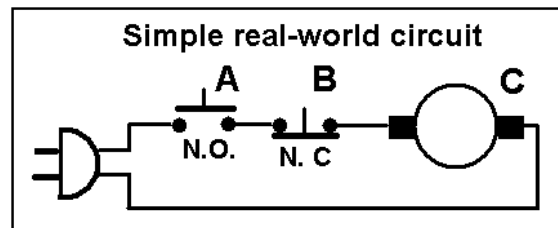


Fig. 1.5 The physical control circuit

Notice that the two **A** and **B** push-buttons are not connected in series as shown in Fig. 1.5, but rather are placed between the hot power line and a PLC input terminal. The **A** and **B** *inputs* will be high or low according to the state of each push-button. Initially **A** terminal is *low* and **B** terminal is *high*. The *output* PLC terminals act in pairs as contacts of an internal relay.

Inside the PLC three memory locations are reserved, **A**, **B** and **C**, corresponding to the *I/O* terminals.. During Step 1 of each scanning cycle, *memory locations A* and **B** are set to correspond to conditions at the *physical input terminals* .During Step 2 *memory location C* is set *true* or *false* as determined by the *program* and the *memory values A* and **B**.

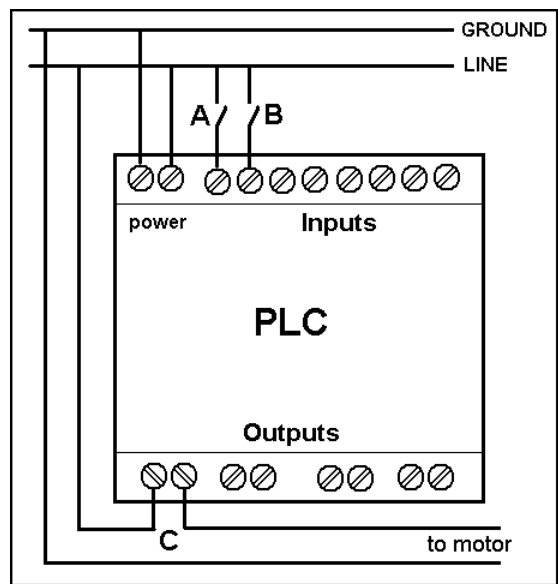


Fig. 1.6 Real PLC connections

During Step 3, the *physical output relay terminals* at **C** are closed or opened according to the value at *memory location C*. Notice how the PLC does not know and does not even care what devices are connected to the *inputs* or *outputs*.

Of course the PLC needs electric power to operate, usually only a few watts.. Typical *input* terminal currents are of the order of a milliampere. With relay contact outputs, no output power is expended. However some PLC models provide an output current in place of relay contacts. What happens during Step 2 depends on the *machine language* stored in *Program memory*. Next we take a quick look at four different programming approaches to constructing this *machine language program*.

1.8 Ladder Diagram [LAD]

The PLC was developed to replace the wire technology for relay control systems then in use. So the graphical *Ladder Diagram* approach was developed to have approximately the look and feel of the relay diagram. As shown in Fig. 1.7 the horizontal line represents one *rung*. Each of the icons on the rung are considered to be either *true* or *false*. Symbols **A** and **B** suggest the contacts of a switch. Symbol **A** (corresponding to memory location **A**) is *true* only if the real normally-open switch **A** is closed (that is, *pressed*). Symbol **B** (corresponding to memory location **B**) is *true* only if the normally-closed switch **B** is *open* (that is, also *pressed*) The symbol **C** at the right (corresponding to memory location **C**), suggests a relay coil. It is *true* provided all the icons to its left on the same rung (**A** and **B**) are also *true*. To summarize, inputs are represented with NO $-|-$ or NC $-|/|-$ *contacts* and outputs are represented by *coils* $-()$.

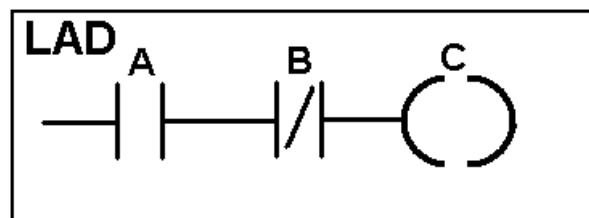


Fig 1.7 Ladder diagram

1.9 Function Block Diagram [FBD]

Logic gates and their implementation in integrated logic circuits were well known when the PLC was designed.. For **Function Block** programming the different symbols

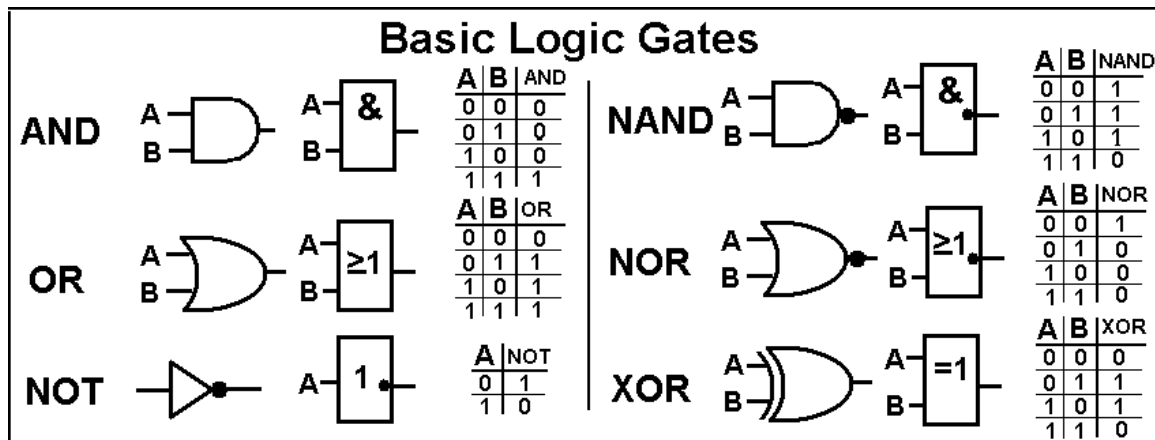


Fig. 1.8 Function Block Diagram

are joined by connection lines using a special graphics drawing program. From such a diagram the machine language equivalent may be automatically created.

A typical [FBD] is shown in Fig. 1.9 For the output at C to be high, The A input must be high, the B input low (that is, **A** memory location is *true*, **B** memory location is *false*) The **AND** gate output is *high* or *true* only if both its inputs are also high or true . In this graphical approach the flow of control is easy to trace.

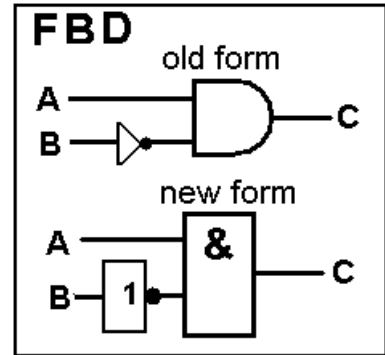


Fig. 1.9 An [FBD]

1.10 Instruction List [IL]

Manufactures of PLC units have developed combinations of two or three letters, suggestive of the instruction name which can be used in place of the bits and bytes of *machine language*. For example, **MOV** can be used in place of the *machine language* instruction to MOVE a data item from one memory location to another. Each set of such memory-jogging letters is known as a *mnemonic* (pronounced as if the initial letter **m** were not present) and the whole collection of *mnemonics* is known as **assembly language**. Each manufacturer must then provide a special computer program, called an *assembler*, to convert the mnemonics into the *machine language* for their particular CPU. The *machine languages* differ for various PLCs, yet they all share a common standardized *assembly language*. **Instruction List** programming is just *assembly language*. in a different package and under a new name. Fig. 1.10 shows the complete **Instruction List** program. Every PLC has a particular section of internal memory called the **accumulator**. The first instruction, **LD A**, tells the PLC to load (**LoaD**) or copy the contents of *memory location A* into the *accumulator*. The next instruction, **ANDN B**, says **AND** the contents of the *accumulator* with the *inverted* contents of *memory location B*, and leave the result in the *accumulator*.. The final instruction, **ST C**, copies or **ST**ores in *memory location C* the contents of the *accumulator*.

IL		
0	LD	A
1	ANDN	B
3	ST	C

Fig. 1.10 Instruction List

1.11 Structured Text [ST]

Programming in **Structured Text**. [ST], is much like programming in *C* or *Pascal*.. Here the program consists of the single instruction, “Store in *memory location C* the result of **AND**ing together the contents of *memory location A* and the **negation** of the contents of *memory location B*.” So if button **A**

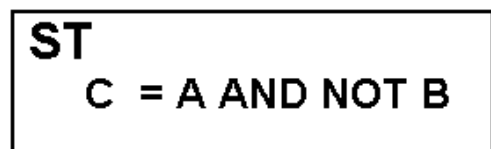


Fig. 1.11 Structured Text

is pushed and button **B** is *NOT* pushed then the contents of memory location **C** is *true*. Notice we need not *allocate memory* or declare the *type* of **A**, **B** or **C** since memory is pre-allocated for all *inputs* and *outputs* and the *type* of these is assumed to be *boolean*.

Structured Text also provides conditional program controls such as IF...THEN, and WHILE...DO constructions

So which language should *you* use? It partly depends on what the manufacturer has provided for your hardware.. **TwidoSoft**, supplied with *Schneider* PLCs offers *Ladder Diagram* and *Instruction List*. **LOGO!Soft Comfort**[®] supplied with Siemens low-end PLCs offers *Function Block Diagram* and *Ladder Diagram*. When you do have a choice, use the programming language with which you feel most comfortable.

This first chapter has presented general ideas common to all PLC s. The following chapters focus on a particular PLC, LOGO!, along with it's available software..

Highlights

A PLC is an electrical control unit with input and output terminals, accepting binary or analog signals. It can replace an entire cabinet of relays, timers, interlocks in an process control system.

A **relay** is a set of one or more switches, whose position is controlled by a separate electric current

The response of the *output terminals* to the state of the *input terminals* is completely determined by a controlling *software program*.

The PLC controlling software program continuously repeats a **Scan Cycle**, consisting of three steps.

Step #1: Store in internal memory the state of all *input* terminals;

Step #2: Execute the instructions in the stored program;

Step #3 :Update from internal memory all *output* terminals.

Internal housekeeping checks are made once each Scan Cycle.

The controlling software program is a series of bytes, called **machine language** which is not easily understood by humans. Four other computer languages have been developed which are easier to understand, and programs exist to translate these into *machine language*

Function Block Diagram [FBD]: graphical array of logic gates and function blocks

Ladder Diagram [LAD]: graphical ladder array of contacts, coils and function symbols.

Instruction List [IL]: sequence of brief letter/number combinations representing program instructions. similar to *assembly language*.

Structured Text [ST]: a high level language of control statements, somewhat similar to C, C++ or *Pascal*.

LOGO!Soft Comfort[®] is a software package for one series of Siemens PLC units, containing a *Graphical User Interface (GUI)*. [FBD] and [LAD] Editors, machine language translators, program simulation and extensive *Help* files

Looking Backwards

- 1: Approximately when did PLCs first appear?
- 2: Are there any mechanical relays, inside a PLC ?
- 3: What is the meaning of **N.O** or **N.C.** as applied to a switch?
- 4: Can a relay have both **N.C** and **N.O.** contacts?
- 5: Can a PLC detect the difference between a temperature sensor and a limit switch when connected as an input?
- 6: In the circuit shown in Fig. 1.5, if the position of the switches **A** and **B** were to be interchanged (i.e. switch **A** placed to the right of switch **B**) could the PLC detect any difference? Explain.
- 7: If an input were to go *high* and then quickly returns *low* during Step 2 of the same scanning cycle, would the PLC ever know about this? Explain.
- 8: In the diagram of Fig. 1.6 the external hardware was connected to the *input* and *output* terminals starting at the left end. Could we have selected different *input* or *output* terminals? Give reasons for your answer.
- 9: Fig. 1.6 suggests the *output* terminals come in pairs, while there is a single screw terminal for each *input*. Why do you think the PLC does things this way?
- 10: If you were to short a pair of *output* terminals of the PLC in Fig. 1.6, would this damage the unit? Explain.
- 11: Could you *safely* connect a **N.O.** or **N.C.** switch between any two *input* terminals? Explain. Can you think of any practical reason for making such a connection ?
- 12: In the *Ladder Diagram* shown in Fig. 1.7 could we interchange the positions of the icons $\neg|$ and $\neg|/$ and obtain the same performance? Could we interchange the positions of the $\neg|/$ and $\neg()$ icons? Give reasons for your answer..
- 13: If the four different programming languages were available to you, which would you prefer. Explain your preference.

LOGO! PLC

Chapter 2: Getting started

2.1 Making a choice

In the previous chapter PLCs were considered in a very general way. Now is the time to get to work with specific hardware and software. The idea is that if you master the PLC of one manufacturer the transfer to other brands will be relatively easy. Since currently Siemens has the largest share of the PLC world market, with an extremely wide range of units from the simplest to the most advanced, we opt for their system. For reasons of economy we concentrate on the low-end **LOGO! 12/24RCO**, which provides 8 inputs and 4 relay outputs. Expansion modules may be added providing up to eight additional inputs and outputs. The companion *software*, to be discussed in detail, is **LOGO!Soft Comfort[®], Version 5.0**. With it you can create programs in both *Function Block Diagram [FBD]* and *Ladder Diagram [LAD]* language (You may program in either language and the software will duplicate it in the other.). You may de-bug and then simulate programs on your computer even without an actual connection to the hardware.

2.2 Drawing pictures

The software is extensive, almost overpowering, so we will proceed one small step at a time. The two available programming languages are both *graphic*, so our first step will be to learn how to draw and save diagrams, with no concern at first whether or not they make any sense.

Moving and resizing

Start the program by calling **LOGOComfort.exe**, Fig. 2.1 shows the upper left corner of the opening screen. Click on the word **File** or the *page* or *folder* icon to *create a new* or *open an existing* circuit diagram. The downward black triangle next to these two icons lets you select your preferred language (**FBD** or **LAD**) for a new circuit, or a listing of recently created circuits. Let's begin with *creating a new [FBD]*. Fig. 2.2 shows the entire screen display, which offers a great number of choices. The first thing to notice is the major window areas of the screen marked as **Circuit Diagram**, **Tools** and **Info Window** separated by two thick black dividing bars, one horizontal, the other vertical. *Left-click* on either bar and a double-headed arrow appears, allowing you move either bar. changing the size and shape .of the window areas.



Fig. 2.1 Opening Screen

Task 2.1: Exploring

A: Try resizing the three window panes by dragging the black bars. If things get too messed up, select **File** ⇒ **Close All** and start again... Just learn by doing ☺

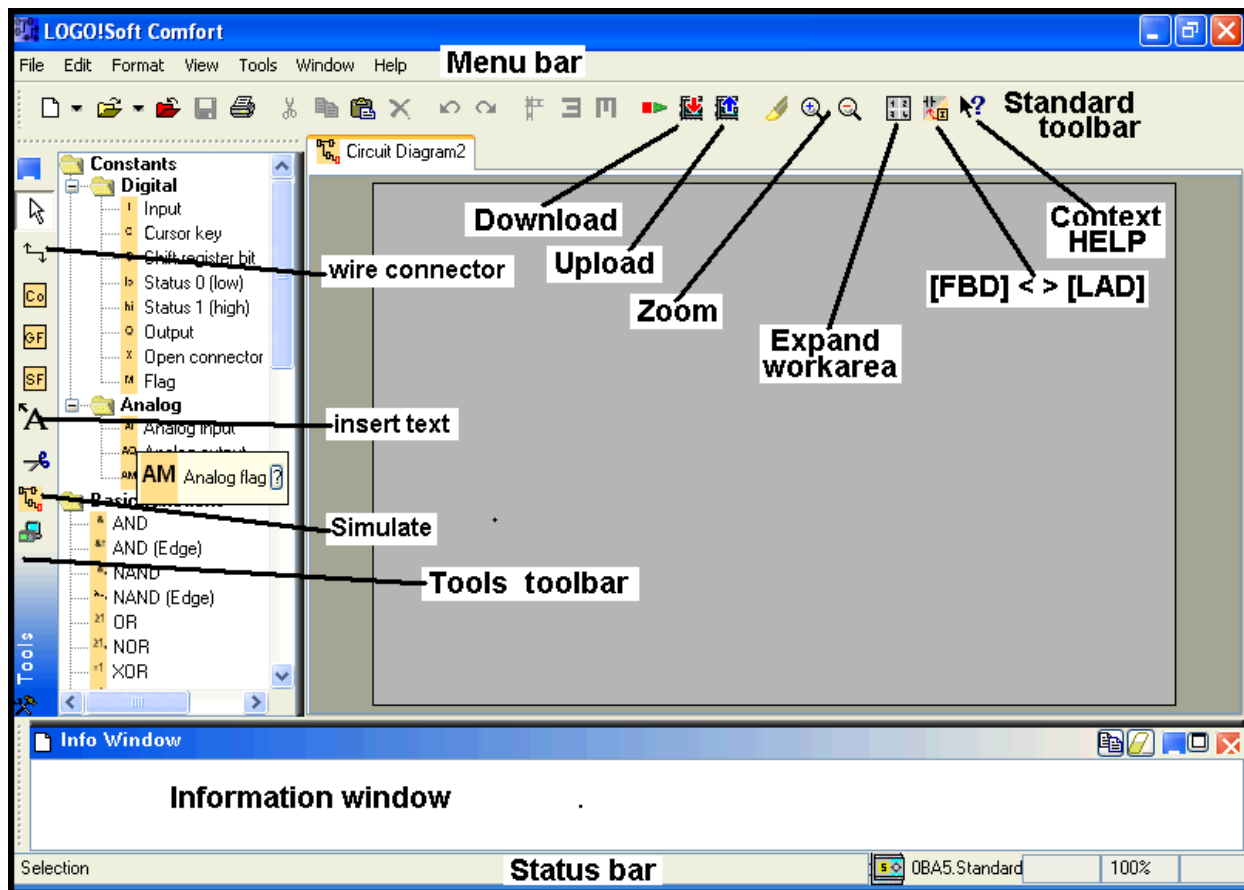


Fig. 2.2 The main screen of **LOGO!Soft Comfort**, where the action is...

Although Fig. 2.2 shows the *default* view it is possible for you to personalize this in a number of ways. However the one feature that almost always remains is the **Menu bar**, the second line from the top. Click on any item on this bar for a further menu box.

File Edit Format View Tools Window Help

Let's start with **View** and begin from the bottom entry. Check **Tooltips** so that whenever the *cursor* is over any *icon*, **Tooltips** identifies it by name. Try *checking* and *un-checking* the other items, **Status Bar**, **Info Window**, **Standard Toolbar**, or **Tools Toolbar** to see the meaning of each option..

Zoom ...
Toolbar ...Standard
Toolbar ...Tools
Info Window
Status Bar
Tooltips

Besides changing the size of the **Info Window** and **Tools Toolbar** by dragging either black *separator bar*, you can also *re-position* these panes, top or bottom, left or right, just by *left-clicking* on the *blue* part of either one and dragging it to the desired position.

The main *Circuit Diagram* window appears in two shades of gray. The inner portion, where you will place all your icons and text, may be **zoomed in** or **out** in different ways: select **View** ⇒ **Zoom...**, or use *Ctrl* and the *mouse wheel*, if available, or select the **+** or **-** icons on the **Standard Toolbar**



Task 2.2: Dragging and zooming

A: Open the **View** menu and observe the results of *checking and un-checking* different items..

B: Drag the **Info Window** and **Tools Toolbar** to different edges on the screen.

C: Try out the different ways to *Zoom* the **Circuit Diagram** area,

Placing blocks in the circuit

Programming in the [FBD] language means that we place **blocks** into a *circuit diagram*, *wire* them together and optionally add *labels and explanations*. The *Tools Toolbar* has three icons, for three different types of blocks,



Constants

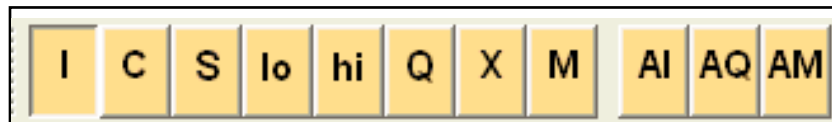


Basic Functions



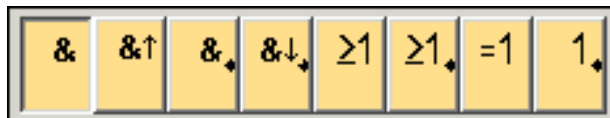
Special Functions

If in [FBD] you click on the **Constants** icon, a row of eleven icons appears:



The three you will most frequently use are **I**, *Input*, **Q**, *Output* and **M**, *Memory location or flag*. Every meaningful program has at least one *input* and *output* block, corresponding to the hardware terminals in use. *Memory locations or flags* are discussed in Section 4.3. The others less-used icons will be discussed in later chapters as we treat them in detail. If you let the mouse hover over any of these icons, the full block name will appear.

If in [FBD] you click the **Basic Functions** icon, a row of eight icons appear:



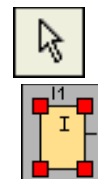
Six of these you have seen in Fig. 1.8. The two icons with *arrows*, representing *edge triggering*, are discussed in Section 4.6. You might well have expected the *Basic Functions* icon to be **[BF]** rather than **[GF]**; however there are several spots in this Siemens software where its original German origin peeks through ☺!

If you are really foolhardy, you might even clock in the **Special Functions** icon, although you are warned in advance not to do this. Disregarding sound advice, if you click, 28 strange looking icons appear. Chapters 5 to 9 are devoted to a step-by-step explanation of these, so better we wait till then. Too much too soon can cause intellectual indigestion.

To insert into the drawing area any one of these icon blocks, left-click the mouse on the icon (it should appear slightly depressed), then move the cursor to the desired location in the work area and left-click again. The block should appear together with its

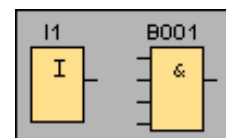
computer-assigned unique *block identifier*. This identifier may be edited by right-clicking on the icon and selecting *Block Properties* .If you wish several blocks of the same type, position the cursor and left-click for each, without going back again to *Tools*.

The **Tools Toolbar** has a **Selection** icon. Left-click on it (or press the keyboard <Esc> key) and the cursor becomes a *magic wand* to move anything already placed in the circuit diagram. Left-click any *block*, and little red squares appear at the corners, allowing you to drag the block anywhere. Right-click the *block* and a window opens to give you more options to **cut**, **copy** or **delete**. Many of the LOGO!Soft editing commands are similar to those used in word processors.



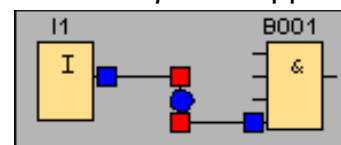
You may notice when dragging any block, it seems to jump rather than move smoothly. This is due to hidden *grid lines*, similar to *Microsoft Windows Desktop* icon alignment. Details on this are found at **Format** ⇒ **Grid Lines** on the *Menu Bar* . Use also **Context-sensitive Help**, explained in Section 2.4 below.. Snapping to grid lines helps make your diagrams neat and orderly.

All *blocks* are rectangular, and if you look closely you can little *stubs* on the sides for wiring blocks to each other, inputs on the left side, outputs on the right. These are the connection stubs used to logically join together the various blocks.

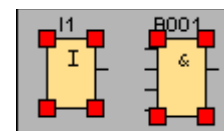


Left-click the **wire** icon on **Tools** and then with the cursor start connecting the stubs on the various blocks. Some connections are not allowed. You cannot connect together two outputs (one might wish to go *high*, the other go *low*!) If you try to do this the computer objects If needed use an OR gate to connect together multiple outputs. No problem with one output connecting to two or more inputs.

Just as with the **Selection** tool you can move and delete *blocks*, you can also move or delete *wires*.. Left-click the **Selection** tool on any wire. *Blue squares* appear at its ends, *red squares* at corners and a *blue circle* in between. Drag a blue square to change the connection, drag the blue circle to move the wire segment parallel to itself. If the computer is not happy with your selection, it will reject it! You can also use the **Selection** tool to outline several blocks and their interconnections. Just left-click and drag the cursor to form a *rectangular outline*. On release, *all* blocks and wires inside will have little red corners. If you drag any *one* block, they *all* move together.



Once a block (or group of blocks) has been selected you may *cut*, *paste* or *copy* . After the selection is made (little red corner squares appear) you may select the desired operation either by *right-clicking* the mouse or by selecting the *cut* or *copy* icon from the **Standard Toolbar**. To *paste*, move the cursor to the new location and *right-click*. No two blocks may have the same identification number, so block numbers are adjusted after a *copy* and *paste*, which is not done after a *cut* and *paste*



Task 2.3: A: From the **[Co]** selection place several **Input** blocks along the left side of the *Circuit Diagram* window. Place several **Output** blocks toward the right side. In between place several **[GF]** blocks of your own choice. (don't use the **[SF]** blocks at this time) Then use the wire connector tool to connect terminals.

Try to connect the *output* of any block back to it's own *input*..

Try to connect together the *outputs* of different blocks.

Try to connect together the *inputs* of different blocks.

Try to connect together the *inputs* of the same block.

B: Use the **Selection** tool to *move* an individual block and also to move a whole group of blocks

C: Try *cut* and *paste* and also *copy* and *paste* on a single block or a group of blocks. Notice carefully any changes in identifier numbers

D: Again use the **Selection** tool to rearrange the *wire connections* between *blocks*. Can you avoid one wire exactly covering over another? How artistic can you make your display?

2.3 Closing and saving

Perhaps many of your diagrams are not yet worth saving, but just in case you want to save, right-click the **Circuit Diagram** tab to open an option window. You can **Save** without **Closing** and **Close** without **Saving**. Before your first **Save**, your masterpiece is known only as *Circuit Diagram1* so you must give it a unique name. If it has already been saved, you can save it again under a new name, using **Save A...** . A window asks you where to save it and under what name. Better you make a special folder somewhere for all your creations, rather than just slapping them on the **Desktop**. Use a file name that suggests the contents, perhaps including date information as well. You pick the file name, the program determines the extension: **.lld** for [LAD] diagrams and **.lsc** for [FBD] diagrams (perhaps suggestive of **Logo Ladder Diagram** or **LOGO Soft Comfort**) You can also **Close** and / or **Save** by clicking on the appropriate *Standard Toolbar* icons. Opening an existing file is as in *Microsoft WORD*.

Close
Close All
Save
Save As ...



2.4 Help! Help!

The software provides you with a great deal of help which may be accessed by two different approaches, **Context-sensitive Help** or **Contents**.

Context-sensitive Help is the last icon on the *Standard Toolbar*. In this mode the cursor takes a new form, similar to the icon. Just move the new cursor over any item left-click. At once a new window opens, displaying information on the item just clicked..

For Help *Contents*, select **Help** ⇒ **Contents**

At the screen upper-left a short selection bar appears with three icons for a *Table of Contents*, an *Index* or a *Word Search* facility.



The **Table of Contents** is really a manual covering almost every aspect of the Version 5.0 LOGO!Soft Comfort package. The main headings are shown in Fig. 2.3 . Selecting items and sub-items follows the same pattern as in *Microsoft Windows*. The *User interface* heading covers everything in this chapter and much, much more. The *Tutorial* heading takes you through all the steps in creating a program. *Reference Material* gives detailed information on every available *block*. *Tips and Tricks* is “*For Adult Programmers Only*”



Fig. 2.3 Table of Contents



The **Index** is unlike the tour-guide approach of the Table of Contents. This *index* is not to be read *from A to Z* (that might be injurious to your health) but rather to refresh your memory on what you already learned.



The **Word Search** first asks you to enter a few letters of the item you wish to search. It then presents a *listing of sections* in which this letter combination occurs, and the frequency of occurrence. It is less useful than the other two options and is not as flexible as other Windows search engines.

Task 2.4: Trying things

Take a quick look at the selections found under **Content / User Interface** on the **Help** menu, and note how detailed are the explanations. Then on some rainy afternoon with nothing better to do, try reading some of this in detail .

2.5 Ladder Diagrams

To create a **Ladder Diagram [LAD]** display, many actions are similar to what you do for **[FBD]** but there are a number of significant differences. On opening a new diagram from the Standard Toolbar **File** ⇒ **New** you must specify **[LAD]** . It is possible to work with several *Circuit Diagrams* but one at a time, just as in *Microsoft Word* you can work with several documents, with each in its own window. So to notice easily the differences, open two new documents, first an **[FBD]** and then a **[LAD]**. The tabs above the display area give the document names. The selected diagram is marked with a colored bar above the names. *Diagram 2* is selected in Fig. 2.4. Notice the different style of icon indicating the language used by each diagram. Just click on either *Tab* to change Diagram windows.

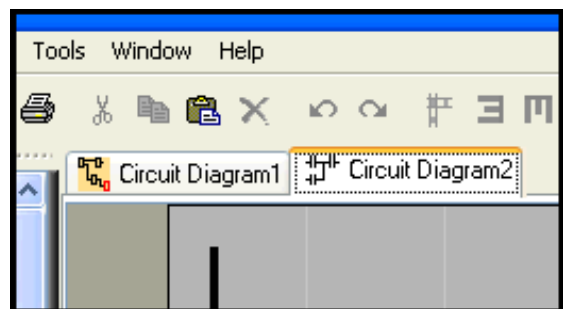
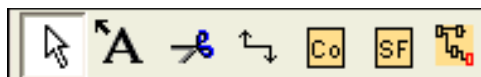
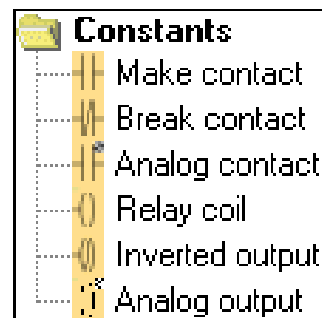


Fig.2.4 Two new diagrams

Next notice in the **Tools Toolbar** there is no reference to **Basic Functions**, **[GF]**, and the names and icons under **Constants** are also different.



In the **[LAD]** display notice the light lines dividing the area into eight *boxes*.. There is also a solid, vertical line passing through the left column of boxes. This represents the **power bus**.



No **box** may contain more than one **block**, and no block can sit on the fence, half in one *box*, half in another. Initially the only blocks we will use are **Make Contact** and **Relay Coil**. To insert a block click on its icon and drag in to an empty box. However if the block is **Make Contact** or **Break Contact** (here *Make* and *Break* are used as adjectives, not verbs!) a new window opens, asking about the **block number**, for in *Ladder Diagrams* the same *Contact block number* may be used over and over again. However every block must have its own unique number. You can use the Wire tool as before, connecting block inputs with outputs.

Because you cannot move the block to different position in the box, there is a danger that by mistake you place two blocks in the same box one exactly covering the other. The rung **A** in Fig. 2.5 is acceptable; rung **B** is incomplete. Rung **C** is what happens if rung **B** is moved on top of rung **A**. Indications of mistakes on rung **C** are the wire lines which run through icons **I1**, **I2** and **I3** and presence of the black dot right after **I3**, which is normally a symbol for a connection of two wires.

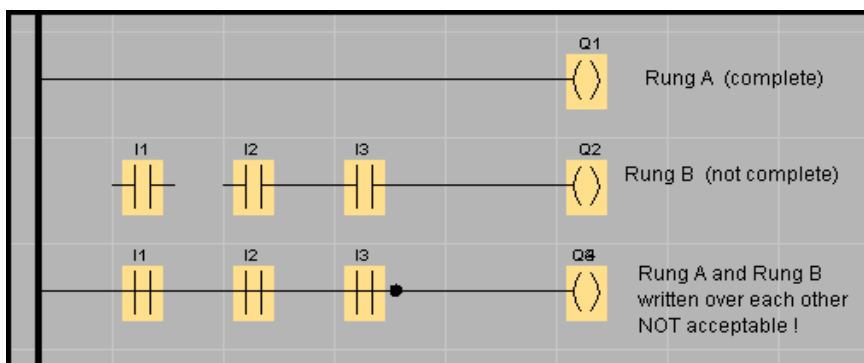


Fig. 2.5 Confusion when blocks are moved over each other

Task 2.5: Your first ladder

Try your hand at a *Ladder Diagram*, inserting, moving, copying, cutting and pasting blocks. Notice what happens to the block number when you move, cut or paste a **Relay Coil**, appearing as a **Q** block

So far in this chapter we have only been doing *finger exercises*. It's like learning a few of the most important key commands of *Microsoft Word* before starting with *Chapter One* of your world-awaited *Autobiography*. So many details we have skipped over, for too much at one time can cause intellectual indigestion. In making the **[FBD]**

and **[LAD]** drawings no explanation was given for the reasons behind the rules. So now it is about time to construct our first simple yet meaningful application.

2.6 As simple as possible

Our first attempt at a real circuit is as simple as possible.

Problem #1: *There is one input, one output, and the output value is to follow the input.*

The PLC does not know and does not care if the one *input* is connected to a **N.O.** or **N.C.** switch.. Also it does not know or care if the single *output* is a lamp, motor or even the input to a second PLC.. The **LOGO! 12/24 RCo**, used as the hardware companion to this manual, has *relay outputs*. capable of switching a 10 ampere current.. An alternate model, **LOGO! 24 RCo**, differs only in its transistor outputs, capable of a *high* output of 24 VDC at 0.3 amperes. Fig. 2.6 shows the PLC program for this circuit in two different programming languages.

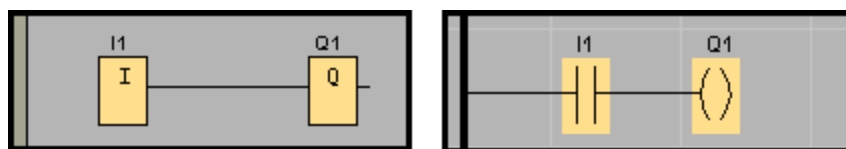


Fig. 2.6 Same configuration in **[FCB]** and **[LAD]** language

From Section 1.6, *Scanning Cycle*, recall the three steps, *Examine Inputs*, *Execute Program*, and *Update Outputs*. In both diagrams **I1** represents the contents of the memory location into which Step #1 copied the state (True or False, *high* or *low*) of *Input #1*.. **Q1** represents the memory location from which Step #3 copies the contents to *Output #1*.

The program executed in Step #2 contains just two instructions in *Instruction List [IL]* language, namely. **Load** contents of memory location **I1** into the accumulator, and **Store** the contents of the accumulator into memory location **Q1**.

Ld	I1
St	Q1

And things are still simpler in *Structured Text [ST]* language

Q1 = I1

In every **[FBD]** display, to each *input* terminal in use (**#1**, **#2**, ...) there corresponds one and only one *input block* (**I1**, **I2**, ...) . The same is true for *output* terminals (**Q1**, **Q2**, ...). These *blocks* represent corresponding memory locations *An output is high only if there is at least one continuously high path leading to it from any input. either directly or through any number of intervening blocks.*

In the **[LAD]** display each rung (only one here) starts from the *Power Bus* on the left with one or more *contact* icons, **-|** **-** or **-|/** **-** and ends with a *coil* icon, **-()**, or some *Special Function* icon, **-□**.. The *coil* icon corresponds to an *internal* memory location, while the *Contact* icons simply ask a question about some memory location. Thus the **Make Contact** icon, **-|** **-** is **true** only if the internal *memory*

location corresponding to *input #1* was **true** during Scan Step #1. Likewise a **Break Contact** icon --|/|-- (not used in Problem #1) is **true** only if the internal memory location corresponding to *input #1* is **false** during Scan Step #1. A **N.C** (normally closed) *real switch* may at any moment be either *open* or *closed* Likewise icon, --|/|-- and --||-- are *true* or *false* depending on how they correspond to what value *Input I1* had during Scan Step # 1. . The program sets *true* the coil icon at the right end of the rung, --()-- , only if *all contact icons* to its left are true. It is as if **truth** has to make its way from the *Power Bus* through all the *contact icons* in order to set **true** the coil icon --()-- at the rung right end..

If you think you have mastered all that, let's move on to the next problem which introduces one small change.

Problem #2: *There is one input, one output, and this output is to be the **inverse** of the input..*

Fig. 2.7 gives a solution.

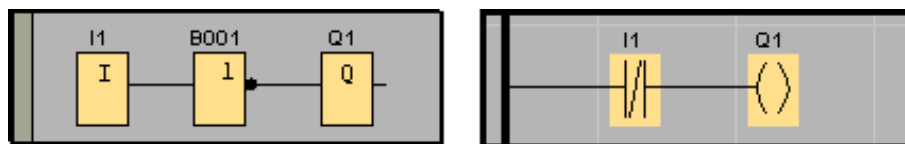


Fig. 2.7 [FBD] and [LAD] solutions for Problem #2

For the [FBD] the added **NOT** gate is all that is needed; a *high input* to the **NOT** gives a low output going to **Q1**. For the [LAD], when the **I1** input is *false* the **Break Contact** icon --|/|-- is *true* and passes along to **Q1** the *truth* from the Power Bus

In **Instruction List [IL]** we have the new instruction, **LdN I1** which copies to the accumulator the *inverse* of memory location **I1**

LdN	I1
St	Q1

In **Structured Text [ST]** the program is

Q1 = ! I1

or

Q1 = NOT I1

2.7 Language Conversion

Program solutions for the two very short problems above were shown in all four languages. Normally you will program in either [FBD] or [LAD], depending on your background and interests. However the computer can automatically convert between either of these two languages. After creating the program in your preferred language it can be very helpful for the computer convert this to the other language for your examination. To convert, go to **File** \Rightarrow **Convert** and click . Alternately you can click on the icon on the *Standard Toolbar*.




Task 2.6: Language conversion

For each of the two Problems above, create the solution in either language and ask the computer to convert it to the other language. Who is smarter, you or the computer?

2.8 Program simulation

So, you created the program, perhaps had the computer convert it to the other available language. But how do you know if the program will actually do what you think you told it to do. The final test is the transfer of the program from software to hardware and observe how it responds to all sorts of action. But perhaps you are programming at home and the hardware is in school. Or you are in school, but there is a line of classmates ahead of you waiting to use the PLC hardware. Not to worry! *LOGO!Soft Comfort* lets you **simulate** the full operation of your program right there on your PC with no wire connections to the PLC

To enter **Simulation Mode** click on the **Tools** icon.  The program you simulate must be in the active *Diagram Window*, and it can be either **[FBD]** or **[LAD]**. You can also conveniently enter or leave *Simulation* mode by pressing Hot Key <F3>. The **Simulation Toolbar** then appears at the lower portion of the screen. The Toolbar left-side is shown in Fig. 2.8 for a diagram with 3 inputs, **I1**, **I2**, **I3** and 2 outputs: **Q1** and **Q2**. Simply click the mouse any one of the three *Input* icons to flip its state. As suggested by the different color and shading **I1** and **Q1** are *high*, **I2** **I3** and **Q2** are *low*.

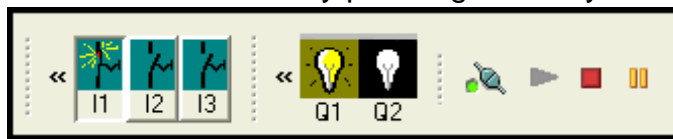




Fig. 2.8 Left portion of **Simulation Toolbar**

In the Simulation mode all connecting lines and block outlines are in color, **red** if the block or line is *True / high* and **blue** otherwise, so you can easily see the logic flow as inputs change. Clicking on any input block is another way to flip its logic state.. In Simulation mode you see what will be the actual performance of your circuit design. And with a *connecting cable* you can move your program into the PLC hardware for actual operation, as will be explained in detail later. But from software **Simulation** you already know exactly what to expect. Every program, it seems, has to go through a *debugging* stage, but with the easy shifting between the **Edit** and **Simulation** windows, your bugs will be short lived.

2.9 Printing your circuits

Imagine how much more Rembrandt could have done if he had a computer printer! So our software package provides an easy way to save your work for generations still to come. The computer prints your circuit in a rather professional manner, like an architect's plans. If you want to fill in the information to be placed in this standard print-out, go to **File** ⇒ **Properties** ⇒ **General** and fill in the blanks.

Before you print you may wish to add to your diagram *additional text* at any convenient location. To do this first click on the **Tools**  **A** icon. Then move the mouse cursor to any free area on your diagram and click. A white window appears, in which you type your text. Use the <Enter> key to start a new text line. If your text runs beyond the edge of the white area, scroll bars appear for moving the view area.. To end text entry, click the cursor anywhere outside the white area.

Next you can use go back the *Tools Toolbar* and click on the **Selection** icon, , and use it move your text, just as you move any block. While your text is selected (the little red corner boxes appear) you may *right*-click the mouse to open a text box offering **Cut**, **Copy** or **Delete**. There is also new option, **Font....** Click this and another window opens, providing you with options for changing *font style*, *size* and *text color* . But unless you use a color printer you only get the color effect on the computer screen.. Finally, if you know in advance your preferred text *size* and *style*, select it first by going to **Standard Toolbar Format** ⇒ **Font...** Changes in text *size* or *style* do not change the appearances of the standard blocks.



Now you are all set for your hardcopy.. Click **File** ⇒ **Page Setup** to set the paper size and desired margins. Use *Landscape* mode rather than *Portrait* . Next go to **File** ⇒ **Print Preview** to view what should come out. The **Zoom** selection there only changes the view on the screen.. Open the **Properties** window and use that **Zoom** selection to change the size of your diagram within the printable area. From this menu you see that are other things besides your *Circuit Diagram* which may be sent to the printer. We will come back to these later as the need arises, or when your circuit seems to be too large for the paper you use. One step at a time in our thousand-mile PLC journey!

A word of caution! If you ask the program to change language between **[FBD]** and **[LAD]** do not expect any of your added text to be transferred; the computer does not know where it should be placed on the new diagram. No big deal for you can always **cut** text from one window and **paste** it in another

Now all is ready for printing, so either click the **Print...** icon in the **Print Preview** window or go back and select **File** ⇒ **Print...** There will still a few more questions to be answered, depending on the brand and model of your printer.

Perhaps all this seems a bit confusing, but after you've been through it a few times, there's really nothing to it for a bright guy like you! ☺ The two programs we created in this chapter are only *baby-steps*. Most of the chapter has been about *housekeeping*. In the next chapter we explore just how much can be done with only **AND**, **OR** and **NOT** blocks.

Task 2.6: Print it!

If you have access to a printer, try out this whole printing process, for it's easier to understand something once you've done it. Even if no printer is handy, still try adding text and explanations to your circuit diagram..

Highlights

The **LOGO!Soft Comfort**[®] software provides a set of toolbars and icons for creating, de-bugging and simulating control programs in [FBD] and [LAD] languages.

The Editors permit the insertion of three types of **icons** into the software diagram, **Constants**, **Basic Functions** and **Special Functions** ([LAD] does not use *Basic Functions*). These *icons* may be re-positioned, interconnected and labeled.

A program may be *saved, recalled* or sent to a *printer*/

Information on specific commands and icons is available through a *Help* system.

Ladder diagrams, [LAD], consist of a series of rungs containing contact icons, as --| --|/ or --|/| and ending at the right in a coil icon, --() , or *function icon*.

Function Block diagrams, [FBD], consist of an array of interconnected logic gate icons and other symbols. representing the control flow of the program.

The software can translate programs between [FBD] and [LAD] formulations

The action of a program may be **simulated** without any connection to the hardware PLC. It may also be downloaded to the PLC and run independently of the computer that created it..

Looking Backwards

1: What commands should you give to make the **Info Window** as large as possible? What, if any, sections are still outside the Info Window?

2: Repeat the above question for the **Circuit Diagram** Window.

3: Do all commands respond in the same manner whether you click the *left* or the *right* mouse button?. Can you give examples?

4; In an [FBD] circuit diagram two wires may cross each other. How can you tell whether or not this means an actual electrical connection?

5: With some graphical drawing programs it is easy to flip either vertical or horizontal, or rotate any selected section. Do you think this can be done in LOGO!Soft Comfort? Would this be helpful for the diagrams we will be creating?

6: Use **Help** \Rightarrow **Contents** \Rightarrow **Word Search** to find references to the word LOGO!.

Describe what you find. . Do you think *Word Search* may be helpful to you in the future?

7 As you insert any block into your circuit diagram, the program automatically supplies a block identifier. Do you think it may be helpful if you also add to your diagram your own names and explanations?

LOGO! PLC

Chapter 3: ANDs and ORs

3.1 Switches and Relays

Every technology seems to have its own set of symbols or icons to document its work. Type setters and proof readers have their own; arc welders have their own. Electrical Engineering and Electronics circuit diagrams also have their own set of hieroglyphics for their circuit diagrams. Here we are concerned in particular with the symbols for pushbuttons, switches and relays.

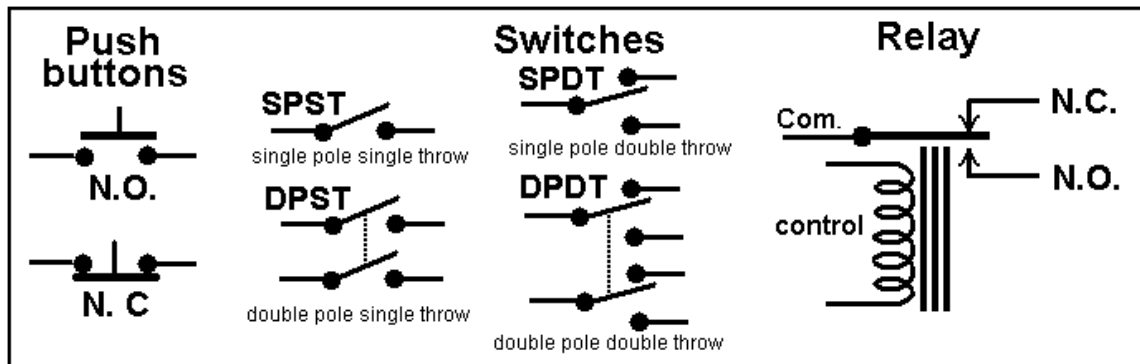


Fig. 3.1 Push buttons, switches and relays

The **basic push button** has two terminals and two states, *open* or *closed*, : *open* if electric current can pass through its terminals, *closed* if current cannot pass through. Only one of the states is stable. Once you take your finger off the button, it immediately returns to its stable state. The **basic switch**, SPST, has two terminals and two stable states. The switch may be either *open* or *closed*. More complicated configurations can be made from simpler units, as seen in Fig. 3.1. The **basic relay** is like a switch except an *electromagnet* replaces your *finger*. When no current is supplied to the coil, each of the two output terminals has its own stable state, **N.C.** or **N.O.**.

For the relay, the current through the coil may be quite different from the current through the contacts. The coil current is either zero or that of the coil rating. The current through the contacts may be AC or DC, 220VAC to drive a motor or a 48 peak-volt telephone conversation. But buttons, switches and relays are all *hardware*, and none of these exist as such inside the PLC. Fig. 3.2 shows a relay-equivalent inside the PLC

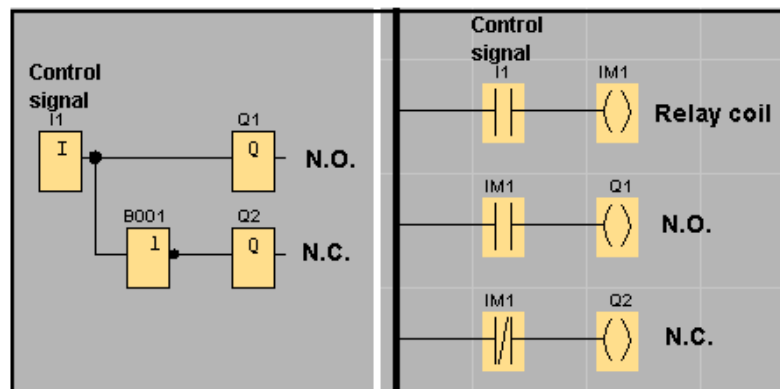


Fig. 3.2 A relay implemented inside a PLC

When the control signal is low, **Q1** is also low, equivalent to the **N.O.** relay

contact, and **Q2** is high, equivalent to the **N.C.** relay contact. This is inside the PLC.

But in digital logic circuits does a **N.C.** switch always produce a normally *high* output? Fig. 3,3 shows how the closing of a switch may produce either a *high* or *low* output.

Look again at Fig.1.6 for the normal switch connections to the LOGO! PLC. To make an input *high*, it is connected to the *line voltage*, usually 12V or 24V. If the input is disconnected from the *line voltage*, it is *low*.

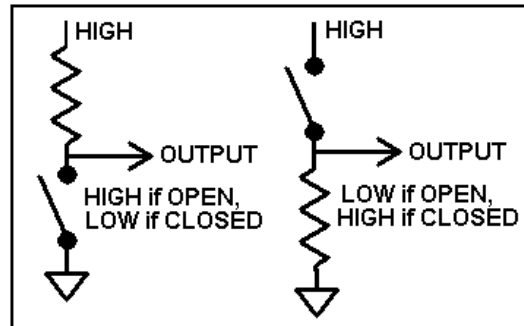


Fig. 3.3 Different outputs when the switch is closed

As we create a program for the PLC in either *function block* or *ladder* language, we arrange and re-arrange blocks or icons in the circuit diagram window. In **[FBD]** we position logic gate symbols, but within the PLC there are no hardware logic gates or connecting wires. In **[LAD]** we position symbols for relay coils and switch contacts, but within the PLC there are no switches, contacts or coils. What then is inside the PLC? Basically the insides of the PLC are memory locations, and circuits to transfer the **0** or **1** contents of these locations. So let's take a closer look at this internal memory

3.2 Memory Overview

Within the PLC there is a central processing unit (CPU) with a limited set of instructions, but mainly there is solid-state memory, divided into *program memory* where the control program is stored instruction-by-instruction, and *data memory*. *Program* memory does not change during PLC operation, but *data* memory may change from one Scan cycle to the next. The *data* memory may be further classified according to its function during the scan cycle.

INPUT memory ; Bit locations, into which during Scan Step #1, the state of all digital input is written. (Analog data to be discussed in Chapter 9). During Scan Step #2 this memory may be read but not changed by the program.

OUTPUT memory: Bit locations, from which during Scan Step #3, the state of all outputs are updated. The program can read from and write to these memory locations during Scan Step #2..

PARAMETER memory: Bit and word locations associated with the *Special Functions* (for example, the time of day, delay time for turning on or off some signal)..

DATA memory: Bit locations for intermediate values written and read during Scan Step #2

3.3 Basic programming rules

The function of the **program** is to modify continually during Scan Step #2 the OUTPUT memory in a predetermined manner based on current values of INPUT and PARAMETER memory

To see how all this fits together we repeat here Fig. 2.7.

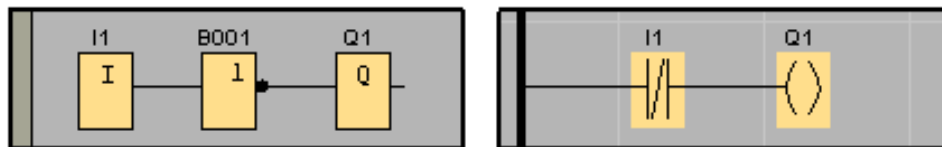


Fig. 3.4 A copy of Fig. 2.7

The whole program in [IL] is only two instructions **LDN I1 , ST Q1**. The **LDN** instruction **copies into the accumulator the inverse of the contents of memory location which follows**, in our case INPUT memory location **I1**. The **ST** instruction **copies the contents of the accumulator into the memory location which follows**, in our case OUTPUT memory location **Q1**. Recall that both these memory locations are single bits, **1** or **0**, **True** or **False**. The left panel of Fig. 3.4 presents the standard logic diagram with one **NOT** gate, block B001 , the output of which is the inverse of its input. The *ladder diagram* in the right panel might suggest a wire diagram of a relay coil connected through a **N.C. contact** to the *power bus*. However an alternate view is that the figure represents *one rung of a ladder diagram*. At the right end of every rung is always a **coil icon**, **-()** , (or a *Special Function* block, as will be seen later) which represents the logic state of a bit either in OUTPUT memory or DATA memory.

This **coil** icon is normally connected to the power bus through one or more **contact** icons, either **make** or **break**, **-|** **|**- or **-|/|**- , in any parallel or series combination. These icons reflect the *true* or *false* state of a particular INPUT, OUTPUT or DATA memory bit. Each *contact* icon is either *true* or *false* (the **break** contact is taken as *true* if the memory bit it references is *false*, and vice versa). The **coil** memory bit is set *true* only if there exists a path back to the power bus passing only through *true* **contact** icons.

3.4 Three IN one OUT...

Let's consider a *PLC with three of its inputs and one of its outputs* connected to the outside world. We wish to create a program that makes the output true for only certain input combinations. For such problems we proceed step by step.

1] **Assign input and output identification.** Here this means assigning **I1**, **I2** and **I3** as the three *inputs*, and **Q1** as the single *output*

2] **Define a truth table** relating outputs to inputs based on the particular requirements of the problem.

3] **Select, insert into the Edit Window and connect icons**, to satisfy the truth table

Case #1 Output is high only if all three inputs are high.

The *inputs* and *outputs* have been designated, so we move to step #2, the appropriate **truth table** . For Case #1 the truth table is shown at the right. This *truth table* is identical to that of a **three-input AND** gate so in step #3 we insert such a block into the [FBD] For the [LAD] a series combinations of three *make contacts*, corresponding to the last row of the truth table, provides the required output. The circuits are shown in Fig. 3.5. .

I ₁	I ₂	I ₃	Q ₁
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

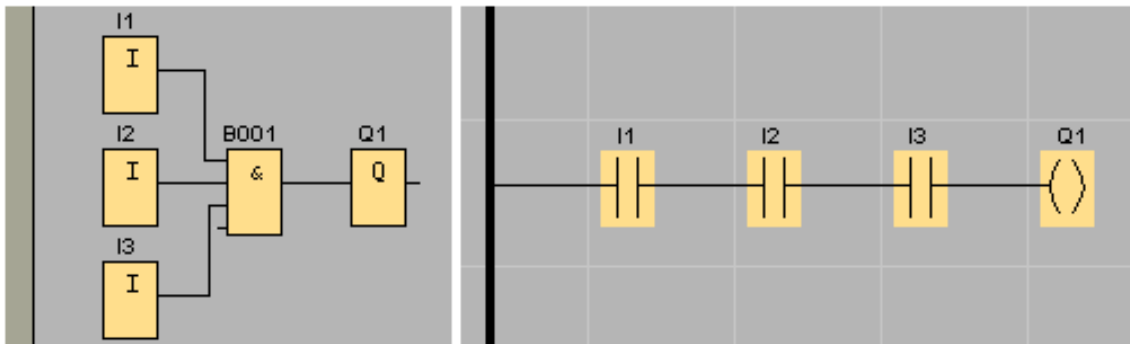


Fig 3.5 Case #1: All inputs high for a high output

The available AND icon permits 4 inputs, but since our gates here are software creations, not real TTL or CMOS chips, the program simply ignores any un-used inputs. For the *Ladder diagram*, when the three hardware inputs are *high*, the three *make contact* icons **I1**, **I2** and **I3** are all true, providing a *true* path from **Q1** to the *Power Bus*.

The *Instruction List* program, to which the above two diagrams correspond is

LD I1 , AND I2 , AND I3 , ST Q1 .

The *memory address* appearing in each instruction, **I1**, **I2**, etc., is called the **operand**. In *Instruction List* programming notice how the instructions are focused on the *accumulator*.. **LD** copies or *loads* its operand into the accumulator, **ST** copies or *stores* the accumulator contents into its operand. **AND** performs an AND operation between its operand and the accumulator and leaves the result in the accumulator. The *Ladder* and *Function Block* diagrams are graphical representations of the statements of the *Instruction List* .And the LOGO!Soft Comfort[®] software easily translates the [FBD] or [LAD] into *Instruction List* statements and from there into the bits and bytes of *machine language*, the only language the PLC understands.

Case #2 Output is high only if the first two inputs are high and the third low.

For this case only the next to the last row of the truth table will have a **1** in the **Q1** column. Here the necessary changes are quite simple. For **[FBD]** simply insert a **NOT** gate, between **I1** and the **AND** gate. For **[LAD]** replace the **I3 make contact** **-|** **-** with a *break contact*, **-|/|-** .. .

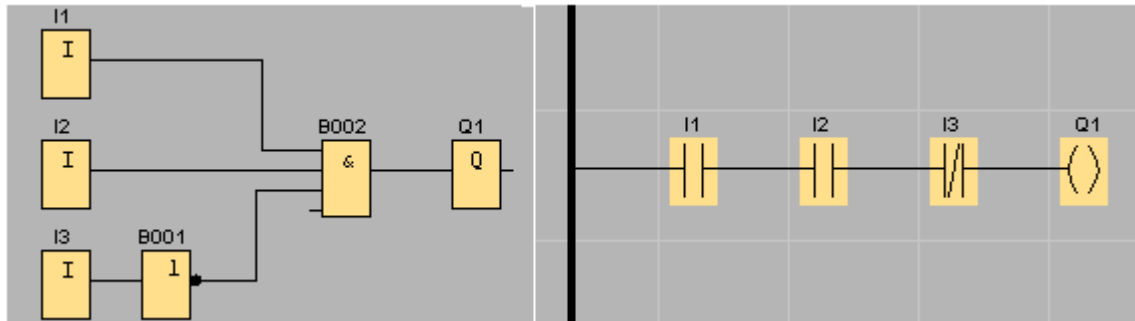


Fig 3.6 Case #2: First two inputs high, the third low, for a high output

Task 3.1: Five in, one out

Given a system with five inputs, **I1** to **I5** and a single output **Q1**. For a *high* output it is required that only **I2** and **I5** be high.

A: Create a [LAD] for the circuit. This should be easy for you.

B: For the [FBD] notice that an **AND** icon has only *four* inputs. Could you use additional **AND** gates? Try to create a workable [FBD]. Then let the program translate your [LAD] to [FBD] and compare this with your own creation. (Note that a • at an input is the equivalent to inserting there a **NOT** gate)

With the original “three IN one OUT” lets go to the next case.

Case #3: output is high if any input is high.

In the *truth table* for this case all rows except the first have a **1** in the **Q1** column. This means that *either I1 or I2 or I3* must be *high*. This suggests an **OR** gate.

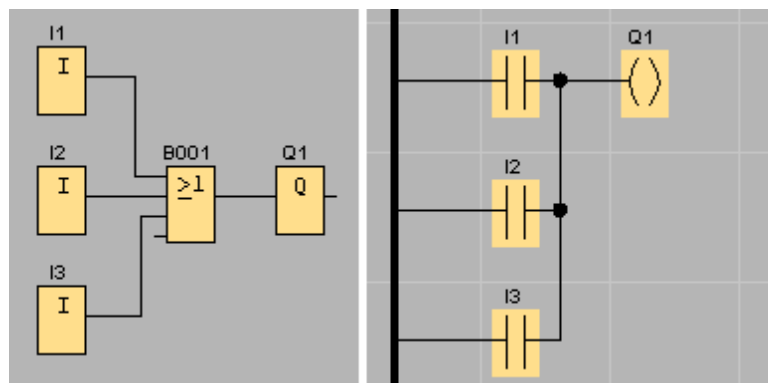


Fig 3.7 Case #3: Any input high, for a high output

Notice the symbol on the **OR** icon, ≥ 1 , suggesting that *one or more inputs be high* for a *high* output. (The symbol for an **exclusive OR** gate, **XOR**, is $= 1$, suggesting *one and only one* input may be *high* for a *high* output.) The *Instruction List* program, to which the above two diagrams correspond is

LD I1 , OR I2 , OR I3 , ST Q1 .

In a [LAD] as was already mentioned, for the *coil* icon to be *true* or *high*, there

must be a continuously *true* path from it back to the Power Bus. The three *make contacts* in *parallel* provide three parallel path options This tells us that:

AND gates correspond to *contacts* in **series;**
OR gates correspond to *contacts* in **parallel**

For some circuits we may need to mix **AND** and **OR** gates.

Case #4: Output *high* if both **I1** and **I2** are high or **I3** alone is high.

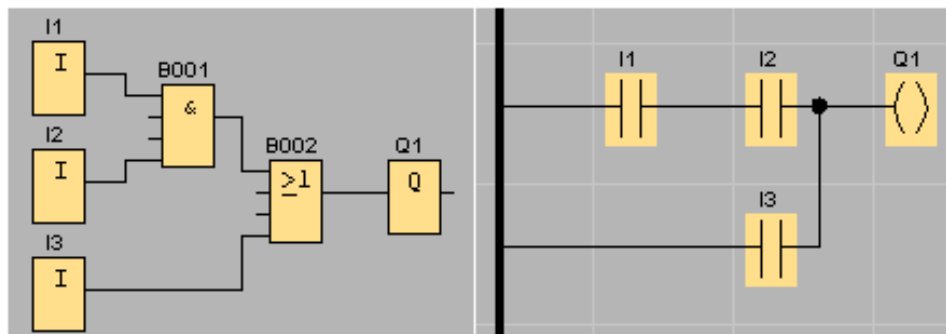


Fig 3.8 Case #4: Output high if both **I1** and **I2** or **I3** alone are high

Task 3.2: ANDs and ORs

A: Create the *truth table* for Case #4

B: Create the *truth table*, **[FBD]** and **[LAD]** for the case of the output *high* only if **I1** is *high* and also either **I2** or **I3** but not both are high. Hint: Try an XOR gate.

Depending on your background (perhaps also on your genes) you may prefer one language formulation rather than another. However it is important to be able to deal with either presentation. Sometimes one description appears more *elegant* than the other. Consider the following: .

Case #5 Output *high* only if any two inputs are high and other is low

Notice in the truth table for this case **Q₁** is **1** for only for those rows with two **1**s and one **0**. This happens for three rows. This suggests three OR operations may be needed. Also the wording suggests “take this *and* that *and not* the other”. Fig. 3.9 shows acceptable diagrams using **AND** and **OR** operations. Notice in the **[FBD]** version each input is connected to all three **AND** gates. One input of each **AND** gate is inverted (the meaning of the black dot, •) . To place such an icon first position the standard gate and then right-click on the input you wish to invert. A window opens and you select **Invert connector**. We could have used **NOT** gate icons , but the use of the • makes our diagram more compact.

I₁	I₂	I₃	Q₁
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

For Case #5

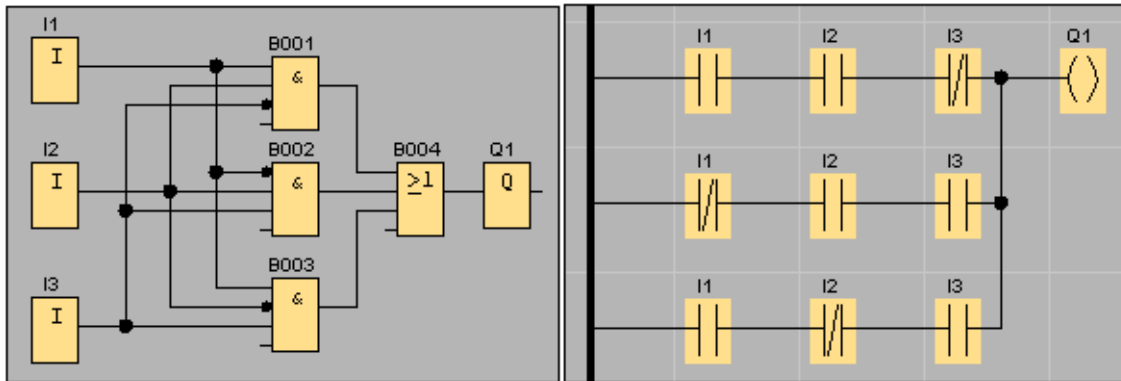


Fig 3.9 Output high only if any two inputs are high and other is low.

Task 3.3: Case #5 and beyond

A: Create the [FBD] circuit shown in Fig. 3.9. Then verify its performance in Simulation mode

B: Make the truth table and the circuit for a system in which *the output is high for one and only one high input*. Verify the circuit in Simulation mode

C: If a printer is available print your displays, along with your own notations.

Have you noticed that if you create your circuit as [LAD] and then ask the program to translate it to [FCB], the result is not always very neat. However there is never a problem in going from [FBD] to [LAD]

The *Instruction List [IL]* code for this problem introduces new instructions. If you are not into *assembly language* and *stacks*, perhaps you can just skip to the next section..

Instructions 0 to 3 **AND** together **I1**, **I2** and the inverse of **I3**, leaving the result in the *accumulator*.. The opening parenthesis, (, of instruction 3 does quite a lot. It suspends the **OR** operation just before it, **pushes** the *accumulator* on to the **stack** and move **I1** into the accumulator. There is nothing new in instructions 4 and 5. The) of instruction 6 looks back to 3, and the suspended instruction just before it, the **OR**. operates on the contents of the *accumulator* and the *top of the stack* and leaves the result in the *accumulator*. Instructions 7 through 10 work the same was as instructions 3 through 6. If you are comfortable with using parentheses on a scientific calculator, you may understand more clearly their use in [IL] programming. It is a bit unfortunate that *LOGO!Soft Comfort*® does not provide an automatic translation to and from [IL] to check this instruction code. For this reason the following chapters of this manual will focus only on [LAD] and [FCB].



0	Ld	I1
1	AND	I2
2	ANDN	I3
3	OR(I1
4	ANDN	I2
5	AND	I3
6)	
7	OR(N	I1
8	AND	I2
9	AND	I3
10)	
11	St	Q0

3.5 Running in hardware.

In the software program *LOGO!Soft Comfort*[®] the hardware PLC unit is referred to as the **LOGO!**. In *Section 2.7 : Program Simulation* we considered how we can run our program on the PC to simulate its actual behavior. We flip the inputs with a click of the mouse, and view the response of the whole system. But it is also possible to download our program from the PC into the LOGO! unit, while at the same time monitoring the operation of the hardware on our PC. In this mode the inputs do not come from the PC but from the real hardware connections, However our PC displays on screen the actual state of all ports and blocks, just like in the simulation mode.

The Siemens' family of LOGO! PLC units come with a variety of capabilities; the later models can do more than the earlier ones. If your program is small and simple it may be run on all modules, but more elaborate programs may only work with the later models. To find out the models that support your particular program go to the *Standard Toolbar* and click on **Tools** ⇒ **Determine LOGO!**. The computer analyzes your program and shows a list of suitable models. You may also find the capabilities of all the models by selecting **Tools** ⇒ **Select Hardware**.

The next step is to *download* your program from your computer (PC) to LOGO!. To make this transfer the computer requires that your program is expressed in the [FBD]. If you had created it in [LAD] just have the computer translate it for you. Of course the LOGO! unit must be supplied with appropriate line voltage (12 or 24 VDC) and be connected to your computer by a suitable cable. Also to each input and output terminal used by your program there must be corresponding wire connections from your hardware to the LOGO!.. To *download* your program use **Tools** ⇒ **Transfer** (or Ctrl-D.) If there are problems (loose connections, power not turned on, nasty virus) the screen will report the error. Otherwise the screen will display the progress of the transfer. Incidentally, each time you make a change in your program to fix a bug and make things better you must again transfer the new version.

So the program has been loaded, now you must start it *running*. Select **Tools** ⇒ **Online Test**, or click on the corresponding icon on the **Tools Toolbar**.  At the screen bottom a new toolbar appears somewhat like that in the Simulation mode. The three actions you can take are indicated by three of the icons. Normally either the *green triangle* or the *red square* is lighted, but not both at the same time. The *green triangle* tells you the program is waiting to be run, so click it to **start**. The *red square* tells you the program is running, and by clicking it the program will **stop**. The *eyeglasses* icon is selected to enable the PC to monitor the actions of the LOGO!. Inputs are no longer determined by the PC and *mouse*, as is the case in the **Simulation** mode.. As the programs you create become larger and more complex, this method of monitoring the program execution in hardware becomes quite significant. 

Task 3.4: Hands On!

Reading an explanation is one thing. Actually carrying out a procedure, step by step, is something quite different. If a LOGO! unit is available to you, run in *hardware* the program you created in Task 3.3.

3.6 Multiplex and de-Multiplex

Industrial control applications frequently include rotary selector switchers, selecting one from a number of inputs, or sending one data input to a number of different outputs. Such operations are known as *multiplexing* and *de-multiplexing*.

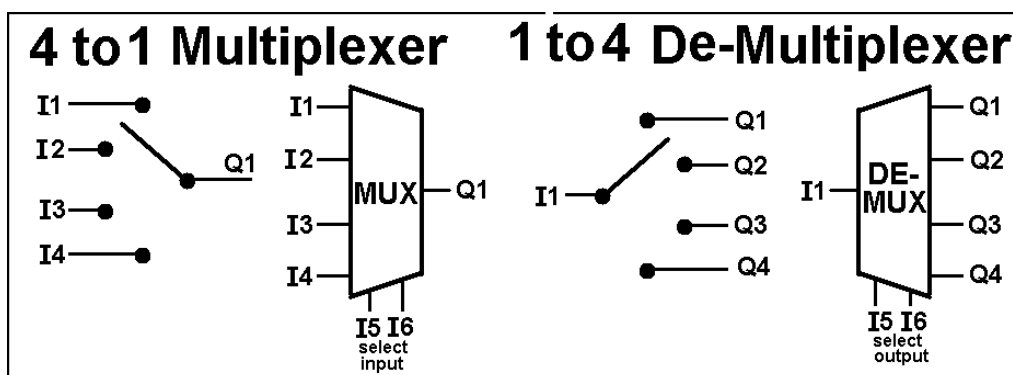


Fig. 3.10 Hardware and schematic of a Multiplexer / De-Multiplexer

Programming our PLC unit to replace the equivalent hardware switches is easy.. Two input terminals, **I5** and **I6** are used to *select* the appropriate *input / output* line. Only *high / low* data from the *selected* line is transmitted to the output. In this case [LAD] gives a more compact presentation than [FBD]. The selection picks one *or* another input, which suggests **parallel** lines of *make* and *break* contacts. The [LAD] program for a 4 to1 multiplexer is shown in Fig. 3.11. Notice that each row contains icons for one data input contact and both selector contacts, **I5**, **I6**... For the [FCB] diagram, we might use four AND gates, each with three inputs (two selectors inputs, **I5**, **I6** and one data input). These four AND gates come together through a single 4-input OR gate.

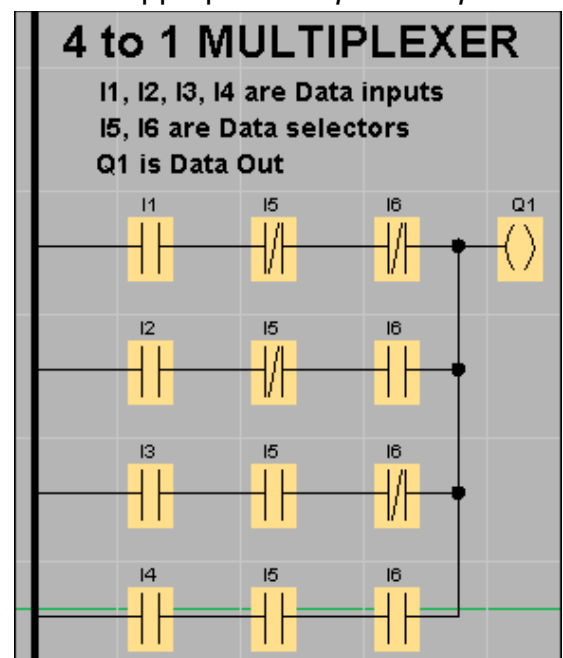


Fig. 3.11 4 to 1 Multiplexer

Hint: for Simulation . . .

Switches at **I5** and **I6** select the input channel that is passed through to the output. These may be *toggle switches* set either *high* or *low*. For the data inputs (**I1**,..., **I4** it would be convenient if during simulation **these** are *momentary high pushbuttons*. This is possible. Select **Tools** ⇒ **Simulation parameters**. A list appears of all inputs used in the program. Click in the *Type* column of any row and three choices for this input type are shown: **Switch**, **Momentary pushbutton (make)** and **Momentary pushbutton (break)**. For **I1**,..., **I4** **Momentary pushbutton (make)** is most convenient.

Task 3.5: Mux and De-Mux

A: With the *Ladder Editor*, enter the program of Fig. 3.10 and *simulate* it, to verify that it is correct.

B: Let the computer translate the program from [LAD] to [FBD] . Do you agree with the translation?. (Notice the computer does not translate your labels or comments)

C: Create a program in [LAD] for a **1 to 4 De-Multiplexer**, and verify it by software simulation. Then let the computer translate this into [FBD].

D: If a PLC unit is available, download and execute in hardware each of your programs.

Highlights

A PLC unit functions independently of the *type of hardware* connected to its terminals. The state of the *output terminals* is determined by the state of the *input terminals* and the *program*

The user-created program resides in the PLC in *program* memory. The *operational* memory area is dedicated to *Input*, *Output*, *Parameter* and *Data* locations

A ladder **rung** is composed of *contact* icons, **—|—** or **—|/|—**, and terminated on the right by a **coil**, **—()** (or function}, icon. A *contact* icon is considered *true* or *false* depending of the state of the memory location it represents. A *coil* icon is considered *true* only if there is a path to the left end of the rung that passes through only *true* contact icons.

A circuit with **N** inputs may have **N²** different input configurations, each determining a specific *output* response, depending of the user program

Multiplex and *de-multiplex* functions may performed by software.

Looking Backwards

- 1: Can the LOGO! *hardware* tell if the switch connected to **I1** is type **N.O.** or **N.C.** ?
- 2: When you are creating the program to run on a PLC unit, is it important that *you* know if the switch connected to **I1** is type **N.O.** or **N.C.** ?
- 3: In most computer memory chips, one byte contains eight bits. If a particular PLC has eight digital inputs, how many *bytes* of memory might be needed to record the states of all inputs?
- 4: Explain the difference between **ANDing** together two bits and **ADDING** together the same two bits. In **ADDING** together two bits, it is possible to get a non-zero *CARRY* .bit. Can **ANDing** together two bits even produce a *CARRY*?
- 5: For standard PLC units explain the difference between a *transistor* output and a *relay* output. With suitable external hardware would it be possible to convert either output to the form of the other? If so, explain how to do it..
- 6: Name the four different areas of PLC memory and explain the function of each area. Are the solid-state memory cells the same in each area, so that the difference is only in how they are used?
- 7: Is there any real difference between stating that a particular memory location is **high**, or **true** or **1** ?
- 8: A particular ladder diagram has three contact icons in series, **I1**, **I2**, and **I3** reading from left to right. Will the program give the same results if the order of the contacts are interchanged.? Explain your answer.
- 9: Someone says that in [**LAD**] *make contacts* are always **true** and *break contacts* are always **false**. Do you agree?
- 10: In a *ladder diagram*, is it possible to have a *rung* that does not have a *coil icon* at the right end? Explain your answer.
- 11: Consider a hardware configuration with five inputs and one output. For the case of *output high only if all inputs are high*, could you program this in [**FCB**] where the **AND** gate has a maximum of four inputs? If it could be programmed in [**FCB**] is there one unique way to do this or might there be many ways? In [**LAD**] what would the diagram look like?

LOGO! PLC

Chapter 4: Remembering...

You wish the door bell to stop ringing once the finger is removed from the button, but once you push the button on your electric fan you wish it to keep running all by itself. Fans and doorbells use different kinds of switches. But the PLC just asks if the inputs are *high* or *low* and doesn't know (or even care about) the kind of switches beyond its terminals.

4.1 Green START, red STOP

For controlling large motors a two-button system is common, push *green* (**I1** input) to start, push red (**I2** input) to stop. Such a circuit is called a **Latch**; if **I1** goes *high*, however briefly, the output **Q** remains high until **I2** goes *high*, however briefly. The **Latch** remembers. The circuit shown in Fig. 4.1 is what we might expect using AND and OR hardware chips. It is basically a *feedback loop*; the output of the OR gate loops back to one of its own inputs. The AND gate acts only as a switch to break the feedback loop. With no input signals, the biasing resistor **R_A** keeps **I1** *low* and **R_B** keeps **I2** *high*.. By definition, an OR gate output is *high* if at least one input is *high*. So if we briefly make **I1** high, the AND gate conducts since both its inputs are now high, output **Q** goes *high* and the feedback loop is complete. Even after the *high* is removed from **I1**, the OR still conducts and output **Q** remains *high*. With a brief low at **I2** the AND gate stops conducting, the feedback loop is cut and **Q** goes *low*, until another *high* is applied to **I1**. This circuit works well with hardware, *but the PLC will not accept it!*

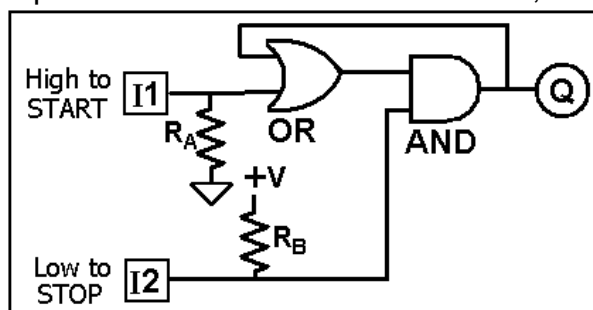


Fig. 4.1 This looks good, but...

There are no AND nor OR gates inside the PLC. Our *Function Block Diagram* is a convenient pattern that LOGO!Soft Comfort® can read to create the *Instruction List*. The first instruction should copy into the accumulator the contents of memory location **I1**, **LD I1**. The next instruction should OR the contents of the accumulator with the contents of some other memory location, **OR** **Q1**. But that value in memory is nowhere to be found. In fact, the *[FBD] Editor* refuses to connect the output of any gate back to its own input.

Fig 4.2 makes one small change: the feedback loop no longer starts from the AND output, which is not a memory location, but from **Q1**, which is a memory location. Therefore the

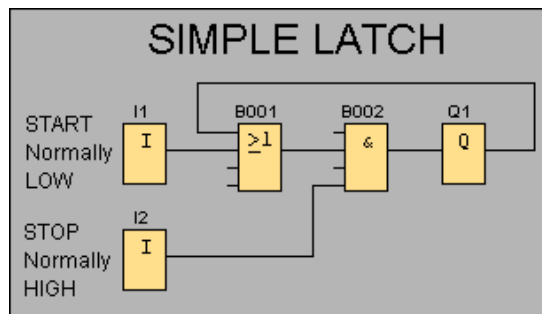


Fig. 4.2 Looks good, and also works!

full *Instruction List* is **LD I1 , OR Q1 , AND I2 , ST Q1** . The **Q1** that is **ORed** is from the previous Scan cycle; what is **STored** in **Q1** will be moved to the output terminals during Step #3 of the present Scan cycle.

Recall from Section 3.4 that **OR** gates correspond to contacts in *parallel*, **AND** gates to contacts in *series* . Therefore we may easily construct the corresponding ladder diagram, seen in Fig. 4.3 .

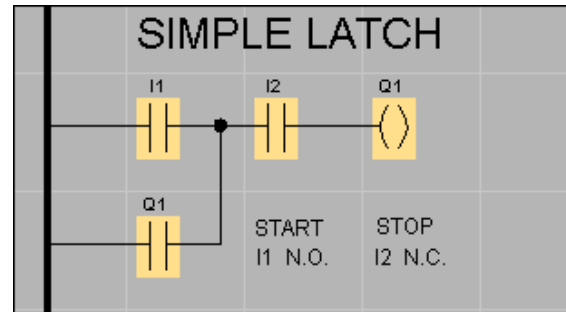


Fig. 4.3 The Latch ladder diagram

Task 4.1:

A: Use the [FBD] Editor to enter the Simple Latch program of Fig. 4.2, and verify it operation by simulation. Select **Tools** \Rightarrow **Simulation parameters** to set the inputs as suitable pushbuttons..

B: Let the computer translate this into [LAD], and verify by simulation

4.2 Set and Reset

The **START** and the **STOP** buttons outside the PLC are different, **N.O.** to **START** and **N.C.** to **STOP**. For some applications it would be convenient to do the switching by using the same style switch at both inputs. Such circuits have been around for a long time, going back to Integrated Circuits, discrete transistors, vacuum tubes and even mechanical latches, and are known various names: **RS gates**, **RS flip-flops**, or **bi-stable multivibrators**. In circuit and also computer language to **Set** an *input* or *memory bit* means to make it **high**, **true** or **1**, while **Reset** makes it **low**, **false** or **0**. Such circuits have two inputs, **S** and **R** . A signal to the **S** input, however brief, **Sets** the output, **Q** and likewise a signal to the **R** input, however brief, **Resets** the output, **Q**. It is common practice in such circuits to provide a second companion output, **Q-bar**, which is the inverse of **Q**.

In the conventional logic gate version shown in Fig. 4.4, the inputs **R** and **S** are normally *low* due to the pull-down resistors to ground, so if one output is *high* the other must be *low*. A high signal, however brief, applied to **S** assures that **Q** will be high. Only a *high* signal applied to **R** will bring the **Q** output *low*.

Fig. 4.5 shows the [FBD] version of this circuit.. No reference is made to the pull-down resistors since these would be outside the PLC. Also the NOT gates are replaced by the \bullet symbol at the OR gate inputs.

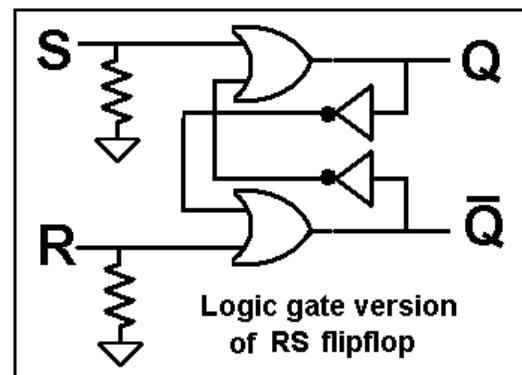


Fig. 4.4 Conventional R S flip-flop

In the [FBD] we see an **OR** gate with **I1** and the inverse of memory location **Q2** as inputs and **Q1** as output. As noted already the [LAD] equivalent is a parallel pair of contacts. If there exists a *true* path from the coil icon **Q1** at the end of the rung back to the Power Bus, then the memory location **Q1** is set *high*..

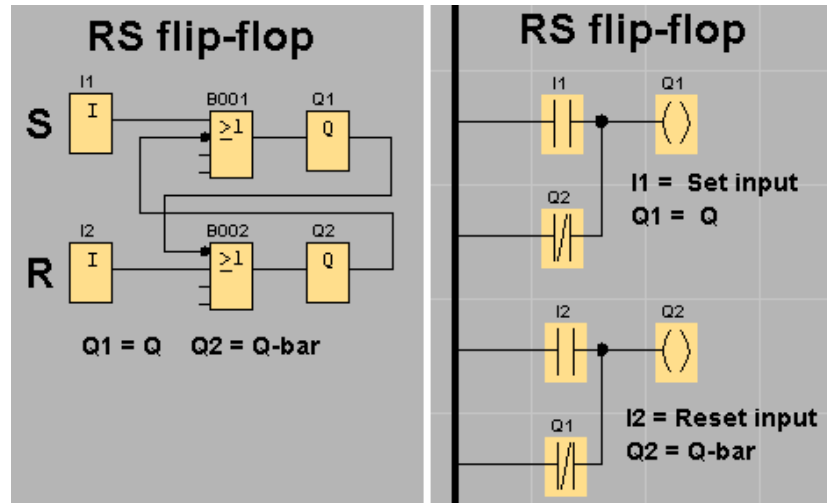


Fig. 4.5 RS circuit implemented in [FBD] and [LAD]

To become someday a really first-rate PLC programmer, you should be able to easily switch back and forth between [FBD] and [LAD] descriptions.

Hint: Drawing Ladder Diagrams . . .

Perhaps you may have already noticed the computer translations between [LAD] and [FBD] are not exactly what we would have expected. The [FBD] translated from [LAD] is often not as neat and symmetric as we would like. The arrangements of the rungs on the computer-translated [LAD] may sometimes appear out of order or hard to follow. This is expected. After all, are you not a lot more artistic than some old computer? ☺

In [FCB] you may initially draw the blocks in almost any order, for [LAD] the sequence is important. Recall that the **contact** icons \neg | \neg and \neg | \neg represent memory locations, either those associated with real *input terminals* or other *already-defined memory locations*. When you place one of these in the ladder diagram, the computer displays a listing of *real inputs* and also *other memory locations that have already been defined*. These *other memory locations* are defined by placing the **coil** icon, \neg (), at the end of some rung. This coil icon or \neg () (or an icon for a **SF** to be explained later) represents a memory location associated with either a *real output terminal* or any other *defined memory location*.

When you place a coil icon, \neg (), the program assumes you want this to represent the next available **Q** for a real output terminal, and no questions asked.. However if you wish \neg () to represent some other memory location, just *right-click* of the icon and a window opens. Select the option **Block Properties. . .** and a list of all allowable labels are shown.

So, for *ladder* diagrams, first place **contact** icons that represent real input terminals, next place **coil** icons that represent real output terminals, then **coil** icons representing other memory locations, and finally place **contact** icons that refer to already defined locations. The bottom line is this: **you can place a contact icon only for an input terminal, or for an already-defined memory location.**

Task 4.2:

The RS circuit shown in Fig. 4.4 is triggered by a brief *high* signal at either input. Re-design the circuit using **AND** gates so that it is triggered by a **low** signal at either input. Verify your results by *simulation* (configure inputs as **Momentary Pushbutton (break)**).

4.3 Flags

The RS flip-flop shown in Fig. 4.5 has both **Q** and **Q-bar** outputs, which is similar to RS circuits implemented on IC chips. The number of output terminals, **Q1**, **Q2**, , , on PLC units is rather limited and in many PLC applications the RS circuit is used only for internal programming and need not be connected to any output terminal. In such cases memory flags, **M**, may be used in place of the **Qs**. Both **M** and **Q** labels refer to memory locations that may be *written-to* or *read-from*. Using the ideas of Section 3.2, the **M** locations are in the DATA area, the **Q** locations are in the OUTPUT area. In Step #3 of each scan cycle only the contents of the **Q** locations are transferred to the output terminals.

In [FBD] to change an already placed **Q** block, from the **Tools Toolbar** select a **M** and place it in the *Edit Window*; then *right-click* on the old **Q** icon to *delete* it, then move in the new **M** and wire it in place. To make the change in [LAD], *right-click* on the old **coil** icon, select **Block Properties...** and select the desired memory flag, **M**. Notice that when you change a *coil* type the program automatically make the necessary changes for any *contact* icon referring to that coil.

4.4 The inverted coil icon, **-(I)**

On selecting **Toolbars** \Rightarrow **Tools** \Rightarrow **Constants** you may have noticed that there are twelve selections for [FBD] and only six for [LAD]. Of the [LAD] selection, there are three **contact** and three **coil icons**. The two icons with the additional small circles refer to *analog* rather than *digital* terminals (to be discussed in Chapter 9). Every ladder diagram rung ends with a **coil** (or a SF icon, to be discussed below); The **coil** represents a memory location either in the OUTPUT or DATA area on memory. Every **coil icon** is connected to the *Power Bus* on the left through one of more **contact icons**, **-|** **-** or **-|/** **-**, each representing a memory location either in the INPUT or DATA area of memory. A **make contact icon**, **-|** **-**, is *true* only if the corresponding *memory location* is **high**; the **break contact icon**, **-|/** **-**, is *true* only if the corresponding *memory location* is *false*. Notice that the icons **I¹-|** **-** and **I¹-|/** **-** both represent the same *memory location*. The *icons* are *true* or *false* depending on the state of the *memory location*.

The memory location corresponding to the **coil** at the end of each rung is set *true* or *false* during Step #2 of each *Scan Cycle*. Only if the **icons** on the left provide a *true* path back the *Power Bus* is the **memory location** corresponding to the *coil* set to 1. The *icon* at the right end of the rung may be either **-()** or **-(I)** and each may be *true* or *false* depending on the *memory location* it references.

But there is a difference with the *coil* icons. If a **-()** icon is at the end of a rung, Only if all *contact* icons on its left provide a *true* path back to the *Power Bus*, is the associated *memory location* for **-()** set to **1**; if the same rung ended in a **-(/)** icon, the associated *memory location* is set to **0**. The Ladder Editor will not let you enter **Q1-()** and **Q1-(/)** in the same program.

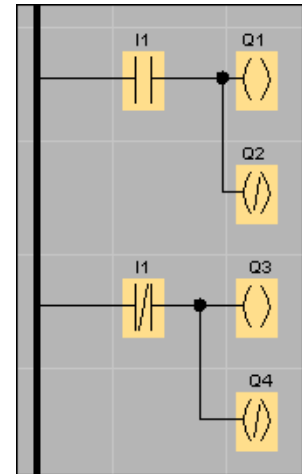


Fig. 4.6 The **-(/)** icon


Refer to the ladder diagram of Fig. 4.6. The four *coil* icons, **Q1**, **Q2**, **Q3**, **Q4** refer to four separate *memory locations* each controlling a separate output terminal. Only one input terminal is wired, **I1**. If this **I1** terminal is *high* during *Step #1* of the *Scan Cycle*, then during *Step #2*, *memory locations* corresponding to icons **Q1** and **Q4** are set to **1**, **Q2** and **Q3** *memory locations* set to **0**. However if the **I1** terminal had been *low*, *memory locations* for **Q2** and **Q3** are set to **1**, **Q1** and **Q4** are set to **0**. During *Step #3* of the *Scan Cycle*, the **Q1** to **Q4** *memory location* values are transferred to the output terminals.

However either end-of-rung coil icon, **-()** or **-(/)**, may refer to a **flag**, with an **M** label, corresponding to a *memory location* in the *DATA memory area*, not the *OUTPUT area*.

In the following section we examine our first *Special Function* which does what the circuits of Figs. 4.4 and 4.5 do, plus a little bit more.

Task 4.3:

A: Enter into the computer the ladder circuit shown in Fig. 4.6 and then *simulate* it, to verify all the statements made above. If available, execute the program in *hardware*, and verify that the *output terminals* agree with the values of the **Q** icons

B: Ask the program to translate your [LAD] into an [FBD]. Does the result look pretty? By editing the diagram with the  icon can you improve it?

4.5 Our first function

The circuit shown in Fig. 4.5 used two of the available output terminals, **Q1** and **Q2**, and each terminal is the inverse of the other. To economize on output terminal use, it is possible to replace the output **Q2** by a memory flag **M2**.so the circuit only uses a single output terminal, **Q1**.

It is easy to make the change. On the [FBD] display, select the Q2 icon (the little red corner boxes appear) and press the on the keyboard. Then from the *Constants sub-menu*, select the **M** icon, and assign it as **M2**. Finally use the wire tool to reconnect to the appropriate OR blocks.

On the [LAD] display, just *right-click* on the **Q2** icon, select **Block properties** and then select **M2** from the list. The **Q2** label on the coil icon becomes **M2** coil icon, and also $Q^2-|/|-$ becomes $M^2-|/|-$.

You may even go one step further and also replace the **Q1** output by an **M1** flag so the circuit is totally internal to the PLC and no output terminals are used.

Because the RS circuit is so useful in control programming, there exist a special package icon and function that duplicates the circuit of Fig. 4.5 but with a single output. It is known as **RS** or **Latching Relay**. In either [FBD] or [LAD] you select the icon from the **SF** sub-menu and insert it into the diagram. The displays are shown in Fig. 4.7

In [FBD] the **RS** icon appears similar to any AND or OR gate; two inputs, one output. In the [LAD] the **RS** icon represents a memory location in the DATA area, which is 1 or 0.

A list of all available functions may be found at **Help** \Rightarrow **Index** \Rightarrow **Special Functions**. Clicking on any one gives a full description. Part of the available help on RS is reproduced here;

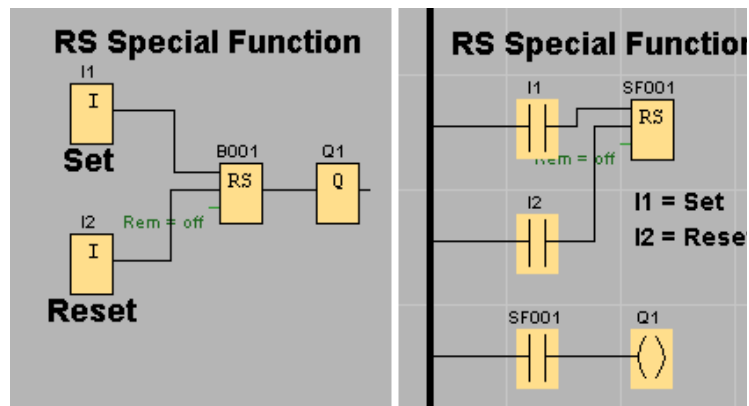
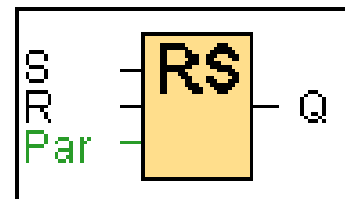


Fig. 4.7 The RS Special Function

Connection	Description
Input S	Set output Q with a signal at input S (Set).
Input R	Reset output Q with a signal at input R (Reset). Output Q is reset if S and R are both set (reset has priority over set).
Parameter	Retentivity set (on) = the status is retentive in memory.
Output Q	Q is set with a signal at input S and remains set until it is reset with signal at input R.



Note the parameter **Retentivity**. Many functions have this option. In the event of a power failure, the values of all functions with *Retentivity* = *On* are stored in a special memory and automatically restored when power returns. Notice near the **RS** icon in Fig.4.7 the text **Rem = off** indicating that this convenient **Remembering** has been switched off. To change this Parameter *right-click* the icon and select **Block Properties...** Check or un-check *Retentivity*. When you inserted the function icon the program assigned to it an identifying label, which in *Block Properties* you may change to a more meaningful *Block name* of not more than eight characters. This can make your circuit more understandable. You may ever make a more lengthy *Comment*, to be displayed near to the icon.

In discussing ladder rungs in Section 2.6, it was mentioned that every rung must end on the right in a *coil* item or a *function block*. Here we meet the first example of a

function block. Both *coils* and *function blocks* are associated with their own *memory locations*. As already mentioned, for a **coil** icon to be *true*, a *true* path must be found leading back to the *Power Bus*. This is not *the* case for functions at the rung end. The *icon* is *true* or *false* (memory location **1** or **0**) depending on input values and the nature of the function. Notice in the [LAD] of Fig. 4.7 how each input of the function block needs a true path back to the Power Bus.

Task 4.4:

- A:** Set up each of the circuits shown in fig. 4.7.
- B:** Place labels at the inputs and give the RS icon some special name.
- C:** Place on the diagram your own name and the date of creation.
- D:** Configure the inputs as *pushbutton momentary make*
- E:** Simulate each diagram to verify its operation
- F:** If a printer is available, print both diagrams (and hang them on your bedroom wall))

4.6 Create a memory cell

Digital cameras and USB drives measure their memory capacity in megabytes and gigabytes. Here we want to design a 1-bit (just $\frac{1}{8}$ byte) memory cell. It is to meet the following specifications:

- A:** One data input, **I1**, one trigger input, **I2**, and one data output, **Q1**.
- B:** The moment the trigger goes high, the input data is to be captured and held
- C:** The output is to continually display the captured data

It looks like we should use the **RS** or *latching relay* since we have to hold the data between captures. The **RS** function requires *two* inputs, not both *high* at the same time, but we may have only a single input.. This suggests we might connect the **RS** Set pin *directly* to **I1** and the *Reset* pin indirectly to **I1** through a **NOT** gate. In this way the *Set* and *Reset* pins always have opposite values. Except when triggered, both **RS** inputs should remain low, which could be nicely done by a pair of AND gates controlled by the trigger input at **I2**. Fig. 4.8 shows a [FBD] for this circuit.

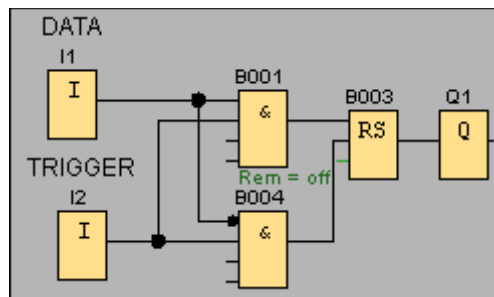
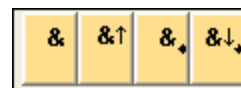


Fig. 4.8 Almost OK, but not quite

Fig. 4.8 may looks OK, but there is still a problem! We are to capture the data only at the **moment** the trigger goes high. but the circuit shown in Fig. 4.8 captures data **as long as** the trigger is high



Perhaps you noticed on the [GF] sub-menu the icons group which offers four different types of AND gates. **&** represents the standard AND gate, **&•** represents a NAND gate, a standard AND gate followed by a NOT. The icons with

the up and down arrows, $\&\uparrow$ and $\&\downarrow$, represent *edge-triggered* AND gates for which the output is high *for just one scan cycle*.

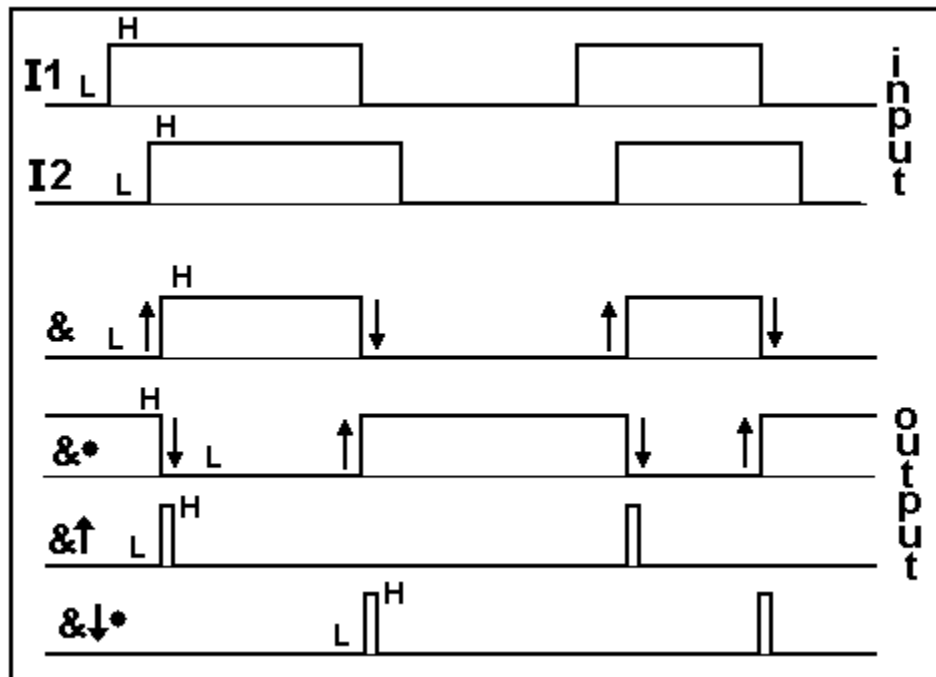



Fig. 4.9 Different versions of the 2-input AND gate

The $\&\bullet$ gate behaves the same as an AND gate in series with a following NOT gate.. Notice how the $\&\uparrow$ is high **for just one scan cycle** following a low-to-high transition (the rising edge) of an AND gate. The $\&\downarrow\bullet$ is high **for just one scan cycle** following a high-to-low transition (the falling edge) of an AND gate.. As usual any unconnected inputs are simply ignored

Of course this is not the way hardware AND gates act, but inside the PLC there are no *gates*, *contacts* or *coils*. This type of *edge triggering* is quite common in the more complicated logic gates. Of these four types AND icons the $\&$ and $\&\bullet$ act as real *gates* (their results stay in the accumulator) but the two edge-triggered versions act as functions, that is, they may be placed at the right end of a rung and they write their result to a memory location just as is done by any function,.

Back to our assignment, if we place an $\&\uparrow$ immediately following the *trigger* input, **I2**, then data from **I1** can only get to the **RS** during a single Scan Cycle. Fig. 4.10 shows the final solution as [FBD] and [LAD]. In the [LAD] you can see clearly how $\&\uparrow$ acts as a function, placed at the right end of a rung.

This is a good circuit to illustrate **retentivity**.. The simulation toolbar icon  may be clicked to produce a brief power interruption. This lets you see the difference if retentivity is on or off for the **RS** block (*right-click* on the icon and select **Block Properties....** For the $\&\uparrow$ retentivity is not meaningful and so it cannot be set.

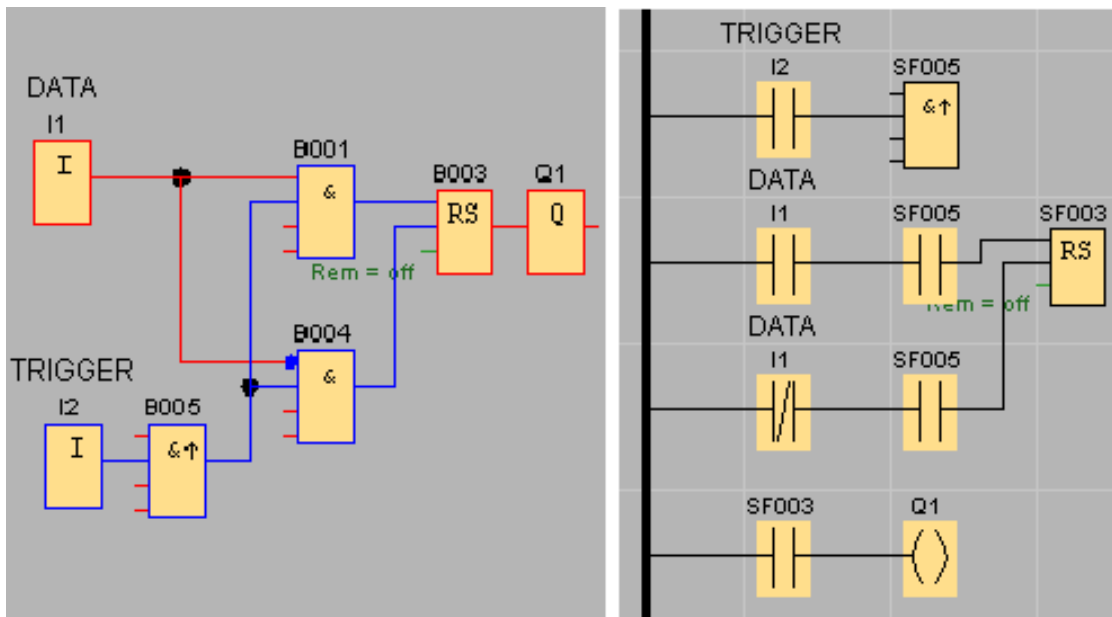


Fig. 4.10 Final form of the Memory Cell circuit

Task 4.5:

A: Set up each of the circuits shown in fig. 4.10.

B: Configure the input **I1** as a switch and **I2** as *pushbutton momentary make*

C: Simulate each diagram to verify its operation

D: Explore the effects of **Retentivity on or off** after a *power interruption*

4.7 Memory matters

As you may have expected, there is a limit to the amount of memory available for your program. Any [FBD] may contain up to a maximum of 130 blocks. Apart from the *Shift Register* Special Function, the same type of function block may be used over again in many parts of your program. Different blocks use different amounts of memory, so besides the limit of 130 blocks, there is also a memory limit of 2,000 bytes. You can check in the *Help Index* under *Memory usage*. for the exact number of bytes needed for any gate or function. You may also view a complete listings of all system resources used by your program .by clicking on the Standard Toolbar **Tools** ⇒ **Determine LOGO!**

Highlights

Some program operations depend not only on the present state of all inputs but also on the *memory configuration* during a prior program step or program cycle.

A **latch** circuit may retain the state of an *input* even after the *input* has changed.

As *nouns*, **Set** refers to a logical *true*, **1** and a voltage *high* ; **Reset** refers to a logical *false*, **0** and a voltage *low*.. As *verbs*, they signify placing an element in such a condition.

A **flag** is a *read / write memory location* under direct control of the program. Up to 24 such flags are available, **M1, M2, ..., M24**

A **function** is a *per-packaged set of specific operations* accepting *inputs / parameters* and returning a *true / false* value. In [LAD] it may be placed the end of a rung, acting like a *coil*.

The **latch** or **RS** *function sets or resets* its output according to the most recent *Set or Reset* input.

After a power interruption some functions can retain their settings (**retentivity**) until power has been restored.

The hardware behavior of *edge-triggered inputs* common in integrated circuits may be duplicated through the use of the edge-triggered AND gates **&↑** and **&↓**.

Looking Backwards

1: Can you have a ladder rung with only a **—|—** and **—|/—** icon in series? Explain.

2 Can you have a ladder rung with only a coil icon **—()—** at the right end?. If so, what would be the corresponding [FBD] ? Could such a circuit serve any practical purpose?

3: A certain PLC installation uses only 3 inputs. How many times can the **I¹—|—** icon occur in its ladder diagram? Explain:

4: Discuss the significance of this rung: **I¹—|—** and **I¹—|/—** and **Q¹—()** in series.

5: Is it possible for two different flags, **M1** and **M2**, to refer to the same location in memory.? Can you change the number of a flag that already appears in your diagram? If so, how?

6: In the **74XX TTL** series of integrated circuits, there are AND gates with 2, 3, 4 and 8 inputs, but no AND gate with a single input. In [FBD] is it possible to have an AND gate with a single input? Give reasons for your answer..

7: How can you find a complete listing of available functions for the LOGO! 12/24 RCo model PLC ?

8: What is the output of the **RS** or **Latching Relay**. if the Set and Reset are both *high*?

9: Is the icon for the **RS** or **Latching Relay**. identical in the [FBD] and [LAD] descriptions? If there are differences, what might this mean?

10: Can an **&↑** icon appear in a [LAD] diagram? In not, then why not?

11: Can a *break contact* icon, $-|/|-$, appear in a [FBD] diagram? In not, then why not?

LOGO! PLC

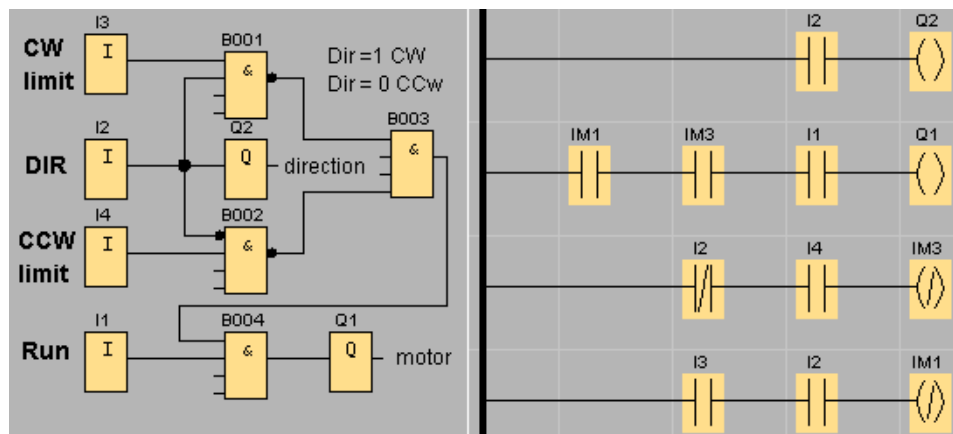
Chapter 5: Limits

Limits are a part of life. While driving a car or writing on a sheet of paper, if you go too far to the left or right you are off the road or the paper. So your eye sees the limit, provides feedback to your brain and you (hopefully) adjust accordingly. Certainly in any automatic control system limit detection and correction are essential.

5.1 Limits left and right...

We start simple. We are to control a constant-speed motor and gear train, which can move an object *right* or *left* . Suppose output **Q1** to be *high* for the motor to run. output **Q2** controls direction, *high* for right, *low* for left.. The operator controls two inputs: **I1** (*momentary pushbutton (make)*) to keep the motor running, **I2** (*switch*) for **DIRection**.. Two additional inputs, **I3**, **right limit** and **I4**, **left limit**, not under operator control, are set *high* if the object is at the end-of-travel position at either end. The operator should be able to run the motor in either direction, until a limit is reached, at which point it automatically stops. It can only be re-started if the *direction* input, **I2**, is such as to move away from the limiting position.

Consider the [FBD] in Fig. 5.1. Even if **I1=1**, (**Run** motor) certain conditions (**DIRection** and **limits**) may prevent this, so **I1** should only reach **Q1** after passing through the controlling B004 AND gate. Either of two conditions can prevent the running: **Dir=1 (right) AND right limit=1** , or **Dir=0 (left) AND left limit=1** . Either of two means an OR gate. The pairing of **Dir** and **limit** conditions means two NAND gates. B001 and B002.



Notice in the ladder diagram the use of *internal flags*, **IM1** and **IM2**, which are not *explicitly* referenced in the [FBD] display. The output of the NAND gates B001 and B002 are stored in flags, which in turn feed the B003 AND gate, but this use of flags is done behind the scene in [FBD]. Notice also in the [LAD] the use of the *inverted coil* icon, **-(I)** discussed in Section 4.4 . A ladder rung with a series combination of two

contacts and an *inverted coil* icon is the equivalent of a NAND gate.

Notice that in an [FBD] a particular input, as **I2**, can appear only once, which in a [LAD] it can appear many times. There is something a bit artificial in simulating the circuits shown in Fig. 5.1. In the problem as given, the limit switches **I3** and **I4** are controlled by the external system, not by the operator, but this cannot be done in our software simulation. If at all possible try to obtain a hardware implementation of this problem, to deepen your own understanding of the problem and its solution.

Task 5.1: End-of-travel limits

A: Enter the [FBD] circuit. Then let the program translate it into a [LAD] . Is the computer translation identical to the diagram in Fig. 5.1 ?

B: Simulate the [FBD] circuit.

C: In simulation is it possible to have both limit switches high at the same time? Is this possible in hardware? Is there any way for the motor to run if both limit switches were to be high at the same time?

5.2 Automatic control

In the previous example the user controlled the shaft position as long as it was between the fixed limits. Now we consider in Fig. 5.2 a case in which the motor control is automatic. The float switches are *open* when above the water level and closed when below. So that the pump is not turned on and off too often, the motor starts refilling the tank only when the water level is below the low limit, and continuously runs until the water is at the high level . Water outflow at the bottom of the tank is random, not under the control of the program. For a water level between the low and high limits the pump motor may or may not be running.

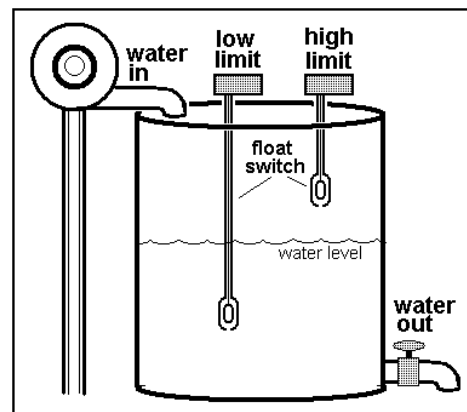


Fig. 5.2 Automatic pump control

To implement the PLC control system, let the only output **Q1** run the pump motor. Let inputs **I1** and **I2** be the low and high level switches respectively.. An **RS** function seems to be a good choice. Connect the *inverted low limit* to *Set* and the **high limit** to *Reset* . When the system is in operation, the water level should always be somewhere between the two level sensors. Simulating this circuit in software is not fully realistic since the program cannot simulate the water outflow at the bottom of the tank. If possible, create a system in hardware. Perhaps recycle the **water out** back to the *pump inlet*..

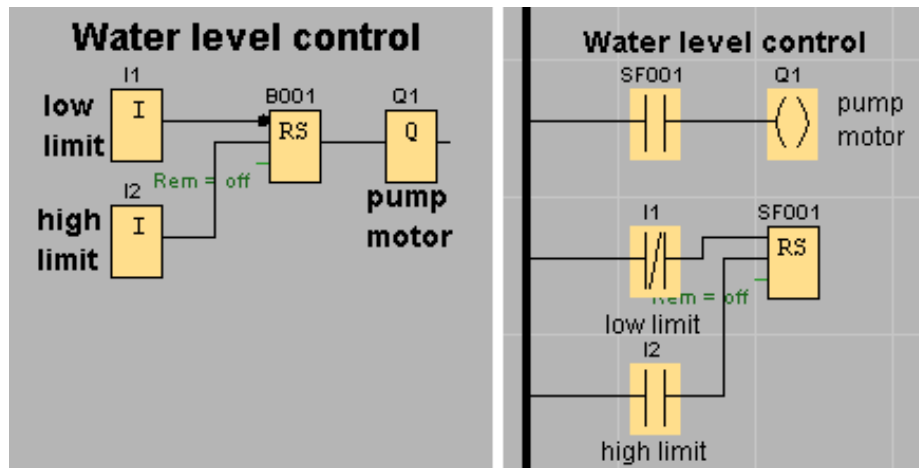


Fig. 5.3 Control circuit for system of Fig. 5.2

Task 5.2: Water level control

- A:** Enter the [FBD] circuit of Fig. 5.3 and verify its performance by simulation.
- B:** Modify and simulate the program for the case in which the level switches are *closed* when above water and *open* when below water.

5.3 The Up / Down Counter

In our first application on end-of-travel, the limit switches were outside the PLC and the user controlled the action (change of position). In the pump application, the *limit* switches were still outside, but the PLC made the decision on the *action* (starting and stopping the pump). We now move on to an application where both the *action* and the *limits* are inside the PLC.

The Pizza Problem

In a new Super-Mall one company has the exclusive pizza franchise: It maintains one central kitchen, but many automatic dispensing stations all over the Super-Mall. At each station the *keep-warm oven* can hold up to 15 pizzas, which is to be automatically refilled by conveyor from the central kitchen whenever there are less than 5 pizza in the oven. Each station has a *single input*, the **Serve** button. Press it and out comes a piping hot pizza! Each station has a *single output*, the command to fetch one pizza from the central kitchen via the conveyor.

To program the PLC to do the job, we need to introduce one new *function*, the Up / Down Counter and a couple of new programming tricks. The formal description of the **Up / Down Counter** is shown in Fig. 5.4

Connection	Description
Input R	You reset the output and the internal counter value to zero with a high signal at input R.
Input Cnt	This function counts the 0 to 1 transitions at input Cnt. It does not count 1 to 0 transitions.
Input Dir	Input Dir (Direction) determines the direction of count: Dir = 0: Up Dir = 1: Down
Parameter	On: On threshold Value range: 0...999999 Off: Off threshold Value range: 0...999999 Retentivity set (on) = the status is retentive in memory.
Output Q	Q is set and reset according to the actual value at Cnt and the set thresholds.

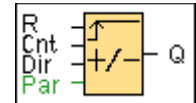


Fig. 5.4 Properties of the Up /Down Counter Special Function

The function output, **Q**, is a single bit, 0 or 1 . The function maintains an internal variable, **count**, which may have a million different positive value from 0 to 999999. When *input R* (for **Reset**) is *low* each positive transition at *input Cnt* increases or decreases by **1** the value of **count**, according as *input Dir* is selected for *Up* or *Down*. But the real secret of the function is in the selection of the **On** and **Off** threshold parameters. The **On** value may be greater than, equal to or less than **Off**.

On > Off	On = Off	On < Off
<div style="text-align: center;"> $Q = 1$ $Q = ???$ $Q = 0$ </div> <div style="display: flex; justify-content: space-between; padding: 0 10px;"> On Off </div>	<div style="text-align: center;"> $Q = 1$ $Q = 0$ </div> <div style="text-align: center;">On=Off</div>	<div style="text-align: center;"> $Q = 0$ $Q = 1$ $Q = 0$ </div> <div style="display: flex; justify-content: space-between; padding: 0 10px;"> Off On </div>

Fig 5.5 **On** and **Off** parameters determine output **Q**

The **On > Off** setting tells us how to get back if either limit is exceeded. For **count** within the middle range, **Q** retains its most recent value. **On = Off** is used to know if **count** is above ($Q=1$) or below ($Q=0$) some definite value. The **On < Off** is useful for assuring that **count** stays within limits ($Q=1$), but if **count** goes out-of-bounds ($Q=0$), we do not know if it is too low or too high. For our purpose in counting pizzas the **On > Off** seems to be the best choice,.

It will be helpful to experiment a bit with this function before we go back to the pizza problem. The test circuit is shown in Fig. 5.6.

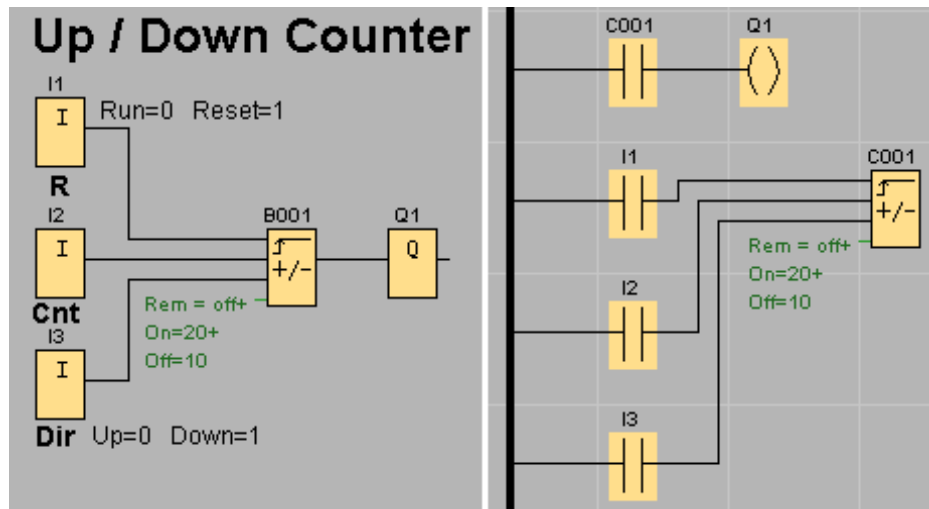


Fig 5.6 Test program for the **Up / Down Counter** function

You already know how to place icons and connecting wires. How do you set the **parameters** of a function block? Right-click on the function icon and then select **Block Properties...** This opens a window, as shown in Fig. 5.7, into which you enter the **Off** and **On** values. You may also enter an optional *Block name* (not more that eight characters) to appear just above the icon next to the program-assigned *ID number*. The *Comment* tab allows you to enter any amount of text, similar to what you can do through selecting the **A** icon on the **Tools** toolbar. The text you enter using this tool may later be moved anywhere on the diagram, and its *size* and *font* modified.

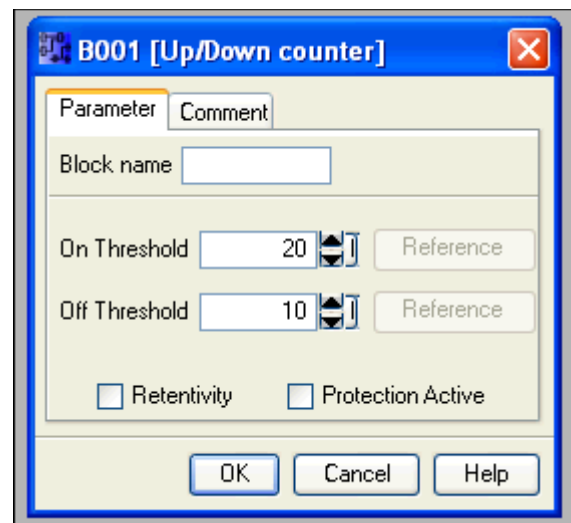


Fig. 5.7 Setting Parameters

While creating and editing the program we do not see the value of the internal variable count. However in the *simulation* mode a small window opens and displays the current value of **count.**, as shown in Fig. 5.8

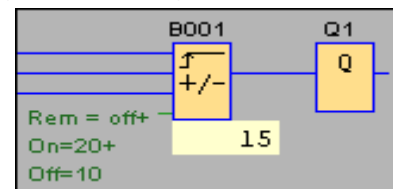


Fig. 5.8 In *simulation* mode

Task 5.3: Up and Down

A: Enter the [FBD] circuit shown in Fig. 5.6 . Set the simulation property of input **Cnt** as **Momentary pushbutton (make)**,. and **Dir** and **Reset** as **switch**, Set **On=20** and **Off=10** .Then in *Simulation* mode experiment counting *Up* and *Down* .Also experiment with **Reset** and with **Power interruption** for **Retentivity** on or off

B: Change the Parameter settings first to **On = Off = 10** and experiment; then use **On=10, Off=20**.

So the **Up / Down counter** is just what we need. Each time a pizza is *served*, **count** goes down by 1. When **count** drops below 5, we want an *automatic refill* to increase count back up to 15. This means that the *program*, not the *operator*, has to do the button-pushing. Let's look at how this might be done.

5.4 The Automatic Button-Pusher

The diagram in Fig. 5.9 shows the entire *Button-Pusher*. In [LAD] there is a single rung, executed once each Scan Cycle. The instruction says “Write to **M1** the inverse of what **M1** currently holds”. If **M1** were read as 1 it would be written back as 0; if it were 0 it is changed to 1. There is no *input*, there is no *output*. In *Step #1* and *Step #3* of each *Scan Cycle* nothing is done; all the action is in **M1** during *Step #2*. Will this really work?



Fig. 5.9 Button–pusher circuit .

Fig. 5.10 is a test circuit shown in the *simulation* mode. This sets **Reset** and **Dir** permanently *low* (count upwards), sets **On=500,000** and connects the button-pusher circuit to the **Cnt** input. Output **Q1** is *low* since **count** is still less than 500,000. The wire to **Cnt** alternately changes color, but since the counting is so fast the two colors blend together. You may stop and restart the counting by clicking the *red square* or *green triangle* on the *simulation* task bar

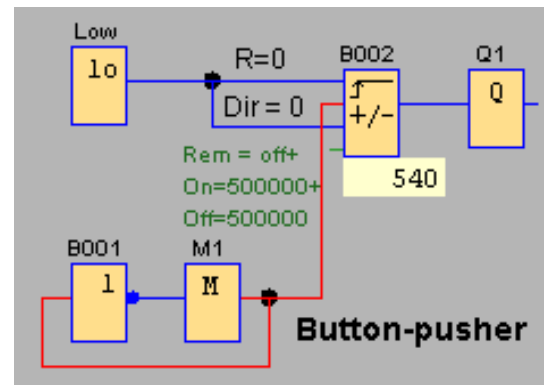


Fig. 5.10 The button-pusher in action

Task 5.4:

A: Enter the button-pusher circuit shown in Fig. 5.10. The **On** and **Off** parameters may have any values, since they can only influence **Q1** and have no effect on **count**. *Simulate* the program.

B: Experiment with the *Stop*, *Start* and *Pause* icons on the Simulation taskbar, as explained in Sec. 2.7

C: With your wristwatch measure the approximate time for **count** to increase by 100. From this estimate the time needed for count to reach 999,999.

5.5 Back to Pizza

Finally we are ready to put together all the pieces for our Pizza control problem. The diagrams are shown in Fig. 5.11.

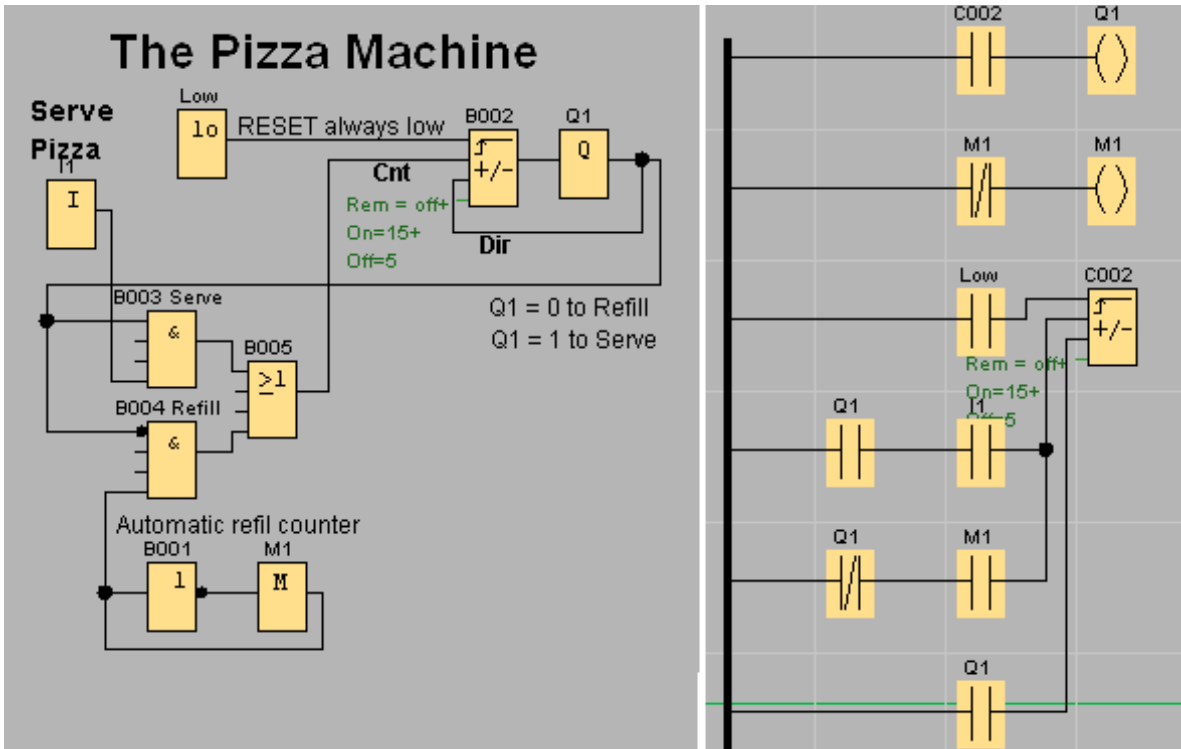


Fig. 5.11 The complete Pizza machine

Try out the program in *Simulation* mode and see how it works. You will meet a difficulty in entering the program in [LAD], for the vertical size of the *Editor window* is not large enough and there are no visible scroll bar. This problem was bound to come up sooner or later as your programs grow larger...

You can increase the width and height of either Editor window, and permit scrolling in either direction by selecting **File** ⇒ **Properties...** ⇒ **Page Layout**. A portion of the window is shown in Fig. 5.12. Here you can select the number of scrollable panels you may need to be able to view your entire diagram. For this [LAD] one horizontal and two vertical panels are needed.

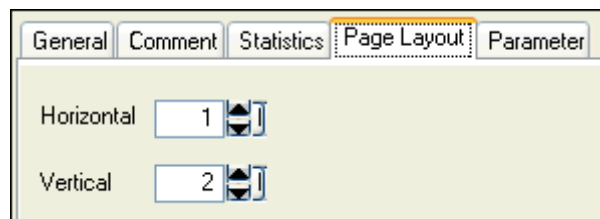


Fig. 5.12 Increasing the Editor window



An alternate approach is to select the icon from the standard toolbar.

Task 5.5:

A: Enter the circuit shown in Fig. 5.11 and start serving Pizza. Try different **On** and **Off** parameters

B: Experiment with the *Stop*, *Start* *Pause* and *Power* icons on the *Simulation* taskbar, and also try re-sizing the *Editor* window for both [FBD] and [LAD].

Notice that the *pizza* process is basically different from first two processes, *object position* and *water level*, which simply ran until a limit was reached. In pizza the process advanced in discrete steps under the control of the PLC and limits were determined by counting the steps involved. For this we used the Up / Down Counter. In later chapters we will explore the use of internal timers and external analog sensors to provide further forms of control.

In these applications, the ability to **remember** is essential. The *present* condition and the remembrance of the *prior* state determine the *following* step.

Highlights

In any control process there is need of some form of feedback. For humans the eye provides feedback for the motion of the *hand*. The simplest form of feedback with the PLC is the use of *limit* sensors.

An **analog** sensor can determine how far, how much, how fast, how hot or cold. A boolean sensor only determines *yes* or *no* in terms of some reference standard.

If what is to be controlled can be quantized, that is, expressed in discrete steps, as the number of pizzas, or the number of degrees of rotation, or the number of liters of water, a PLC can approximate analog limits using the **Up / Down Counter** function.

The proper setting of the **On** and **Off** parameters of the **Up / Down Counter** function is the key to the function's usefulness in practical applications

Looking Backwards

- 1: Explain why an RS latch was used for the circuit in Section 5.2 but not for the circuit in Section 5.1
- 2: Can the internal variable **count** of the **Up / Down Counter** function ever be *negative*? How can you prove your answer?
- 3: If you are counting *down* with **Dir=1** what might happen when **count** reaches zero? How can you find out?
- 4: From the timing of the automatic button-pusher circuit, how would you compare the *processing speed* of a PLC and a PIC chip? How would you compare the *number of inputs and outputs*? How would you compare their *price*?

LOGO! PLC

Chapter 6: Time and Timers

6.1 Classifying Timing Functions

A glance at the *Help* files for *Special Functions* shows a wide variety for some form of *timing* control. The output of all the timing functions is *boolean*, that is, *high* or *low*, not a numerical value expressing time. An electronic stopwatch displays the duration of some process or time interval. The PLC is not designed for that purpose. True, in the *simulation* mode we can view time intervals, with an accuracy of a few hundredth of a second. But in actual operation the PLC does not display numerical values, unless additional communication hardware is added to the system.

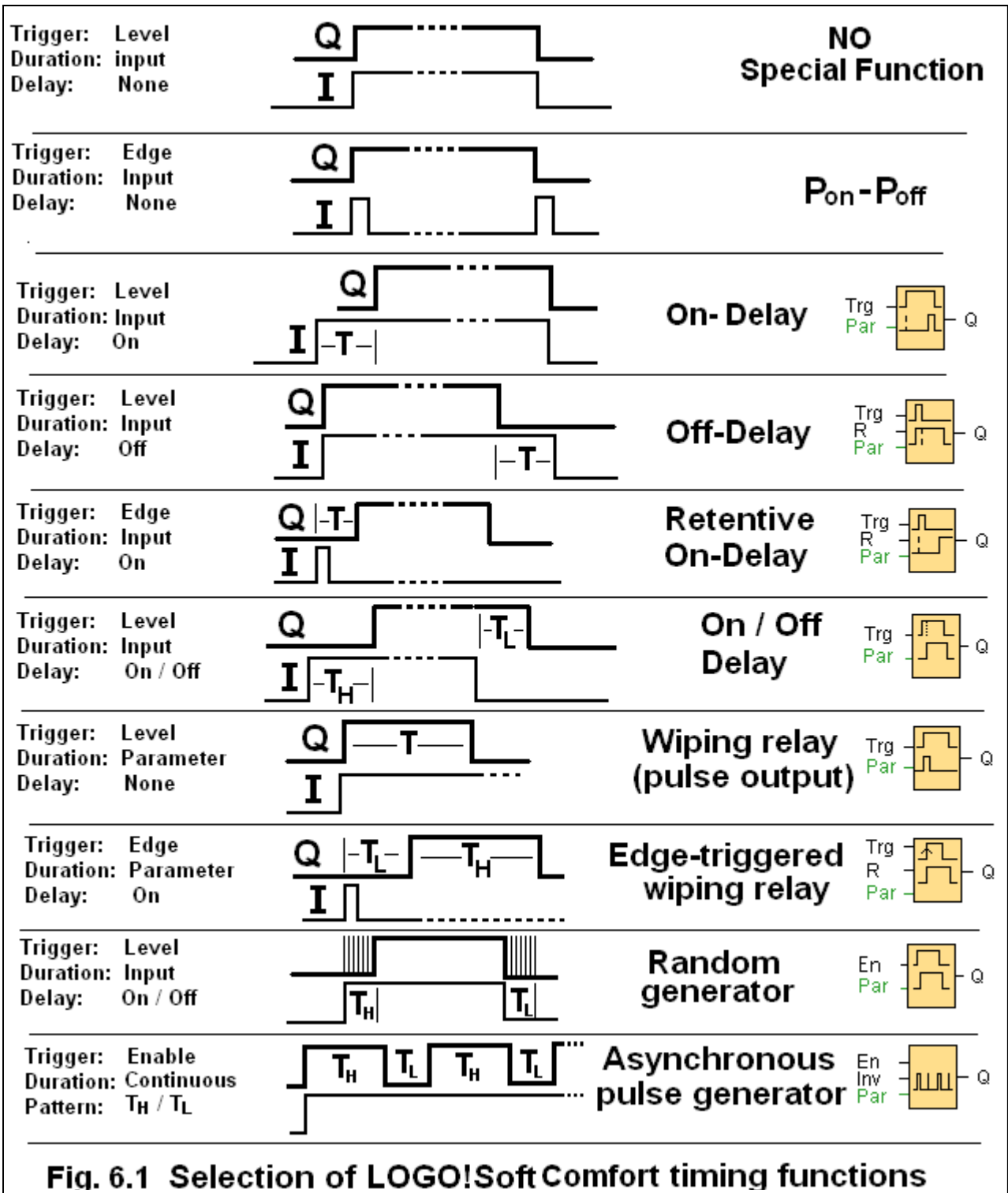
For each timer *Special Function*, *output* depends on *input* and the settings of **parameters** proper to each function. The input controls either by **level** or **edge**, that is, *while* the input is *high* or *low*, or *when* the input *changes levels*. This is different from the ideas of a *pushbutton* or *switch* of Section 3.1. which is outside the PLC. *Level* and *edge* are programming concepts inside the PLC. Recall Section 4.6 on *edge-triggering*..

Another point of interest is **duration**, (how *long* is the time interval), which may be determined by the **input** or by the values assigned to the function **parameters**.

In some cases it is also desirable to have some pre-programmed **delay** when executing the command to *start* or *stop* some process. A large motor may be started anytime, but it is usual to have a *definite delay* *before* full current is applied.. Also for some projection lamps it is desirable to *delay* for some time before turning off the cooling fan *after* the lamp itself is turned off. As a programmer it is important for you to understand the unique characteristics of each available timing function, and to select the one most appropriate for your application.

A classified summary of the LOGO!Soft Comfort[®] timing functions is presented in Fig. 6.1 below. We will examine in some detail a number of these, to understand how they may be used in practice.

We have seen the simplest configuration, back in Fig. 2.6 . The output level, **Q1**, follows exactly the *input level*, **I1**, with no **delays**. The *duration* depends on how long the **input** remains high.



6.2 Push On – Push Off

A common style of switch, found in desk lamps and some electronic equipment is *Push On – Push Off*. Such an *external* switch may be imitated *internally* in the PLC. Output **Q1** goes high at the rising edge of input **I1**, continues *high* even while **I1** has returned to *low* **Q1** only goes *low* on the next rising edge of input. Fig 6.2 shows how such a *Push On – Push Off* circuit may be formed from an *edge-triggered* AND, $\&\uparrow$ (explained back in Section 4.8) and the **latch** or **RS Special Function**. The **latch** has two inputs. The added pair of AND gates keeps these *low* except at the rising edge of the **I1** signal as formed by $\&\uparrow$. The inverse feedback from output **Q1** assures that both AND gates cannot be open at the same time, so **Q1** is alternately high and low.

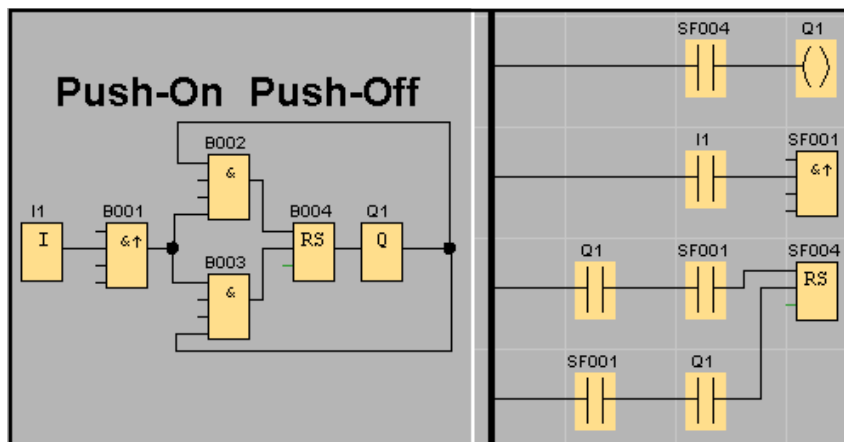



Fig.6.2 Push On–Push Off circuit

Apart from just being a particular kind of switch, this circuit has the interesting property that for every two low-to-high transitions at the input there is a single low-to-high transition at its output. Basically it is a *divide-by-2 circuit*.

Task 6.1: Divide-by-16

A: Enter the [FBD] circuit shown in Fig. 6.2 and verify its performance by simulation.

B: Make three additional copies of this circuit in the same *Editor window*, using *Copy* and *Paste*. Notice how *Copy* and *Paste* adjusts all *block ID numbers* to avoid duplications. You may need to provide vertical scrolling by *File* \Rightarrow *Properties* \Rightarrow *Page Layout* or the by clicking the toolbar icon 

C: Delete the **I2** input block and connect the following $\&\uparrow$ block back to **Q1**. In this way the output of the first *divide-by-2* circuit becomes the input for the following. Again delete **I3** and connect the following $\&\uparrow$ block back to **Q2**. Repeat for the last *divide-by-2*. This should give you a single input, **I1** and four outputs, **Q1**,..., **Q4**.

D: Simulate this circuit. Notice that 16 input cycles at **I1** are needed to produce one complete output cycle at **Q4**. You have made a **divide-by-16** circuit!

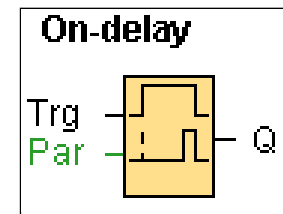
E: As a comparison have the program translate your divide-by-16 from [FBD] to [LAD]. Which display do you find easier to understand? In [FBD] you can trace the *program flow* from **I1** right through to **Q4**

6.3 Time Delays

In starting large motors, full current is not applied is delayed a few seconds after starting, and in some projectors after the lamp is turned off, there is a few seconds delay before the cooling fan is stopped. There are examples of an **On-Delay** and **Off-Delay**. Our software program provides both of these and also some variations.

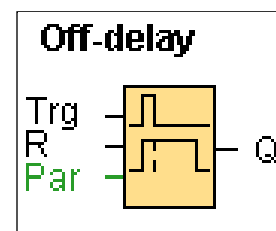
On-Delay

Connection	Description
Trg input	The on delay time is triggered via the Trg (Trigger) input
Parameter	T represents the on delay time after which the output is switched on (output signal transition 0 to 1). Retentivity on = the status is retentive in memory.
Output Q	Q switches on after a specified time T has expired, provided Trg is still set.



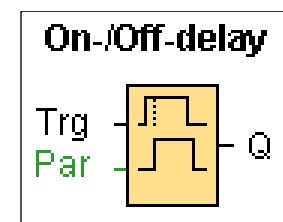
Off-Delay

Connection	Description
Input Trg	Start the off delay time with a negative edge (1 to 0 transition) at input Trg (Trigger)
Input R	Reset the off delay time and set the output to 0 via the R (Reset) input. Reset has priority over Trg
Parameter	T: Q=0 on expiration of the delay time T (output transition 1 to 0). Retentivity on = retentive in memory.
Output Q	Q remains on for time T after Trg goes low.



On-Off-Delay

Connection	Description
input Trg	You trigger the on delay with a positive edge (0 to 1 transition) at input Trg (Trigger). You trigger the off delay with a negative edge (1 to 0 transition).
Parameter	T_H is the on delay time for the output (output signal transition 0 to 1). T_L is the off delay time for the output (output signal transition 1 to 0). Retentivity on = the status is retentive in memory.
Output Q	Q is switched on upon expiration of a configured time T _H if Trg is still set. It is switched off again upon expiration of the time T _L and if Trg has not been set again.



For **On-Delay Trg** must remain *high* during the delay period **T**. Once **Trg** goes *low*, output immediately follows.

For **Off-Delay** timing delay **T** starts only after **Trg** goes *low*. This function has an additional input, **Reset**, which forces the output *low*, irrespective of both **Trg** and **T**.

On-Off-Delay combines the above two functions. A *high* output is delayed by time **T_H** after **Trg** goes *high*. After **Trg** goes *low* the output follows only after a delay time **T_L**. There is no **Reset** with this function.

Problem #1: A particular motor has two parallel windings, **A** and **B**. Voltage to winding **B** is to be delayed 5.5 seconds after motor starts. The PLC has a single input, which must be *high* to operate the motor, and two outputs, one for each winding.

Since **A** winding is active as long as the motor is running, output **A** should be directly connected to the single input, **I1**. The 5.5 second delay for **B** should be *level* triggered, not *edge* triggered, since the motor is to stop when **I1** goes *low*, so an **On-Delay** is suitable here, with parameter **T** set to 5.5 seconds. A solution is shown in Fig. 6.3. As soon as the input **I1** goes *low* the currents in *both* windings stop.

Problem #2: The same type of motor is to be tested on a production line with the same starting sequence, and turned off **automatically** after a fixed time interval, **T**.

Once you hear the word *automatic*, think *edge-triggering*. The input starts the process, but it is up to the program parameters to carry it on from there and to bring it to an end.

Perhaps before attacking this problem, it will be helpful to examine two *edge-triggered* circuits that offer a fixed *starting delay* and a fixed *running time*,

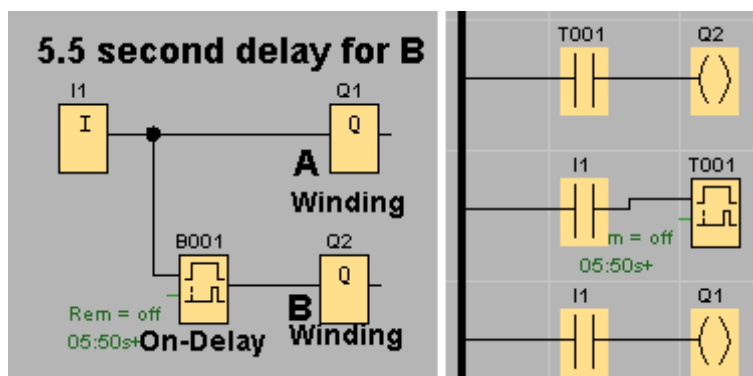


Fig. 6.3 One solution for Problem #1

6.4 Edge-triggered, fixed delay, fixed duration

For the three delay functions discussed above the parameters, **T**, **T_H**, **T_L** controlled delays at the *start* and *end* of a time interval, but **Trg** determined the *duration* of the interval itself. There are two other delay functions for which the duration interval between on and off is not determined by **Trg** but by the parameters of the function. The action *starts* with a *low-to-high* transition of **Trg** but then continues independently of **Trg**. This is called **edge triggering**.

The first of these is the **Retentive On-Delay** described in Fig. 6.4

Connection	Description
Input Trg	Trigger the on delay time via the Trg (Trigger) input.
Input R	Reset the time on delay time and reset the output to 0 via input R (Reset). Reset takes priority over Trg.
Parameter	T is the on delay time for the output (output signal transition 0 to 1). Retentivity on = the status is retentive in memory.
Output Q	Q is switched on upon expiration of the time T.

Retentive On-Delay

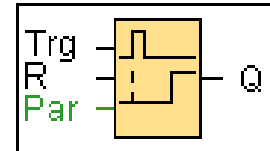


Fig. 6.4 **Retentive On-Delay** Special Function

A parameter sets the delay time, **T**, which is measured starting from a 0 to 1 transition (edge triggering) at input **Trg**. When and if **Trg** goes back down to 0 has no further effect. A *high* signal on the **R** (**Reset**) returns the output to low (or keeps it from ever going high) This function is *edge-triggered* and does provide a programmable *on-delay* but lacks a programmable duration. But a second identical unit can take care of the duration, as shown in Fig. 6.5.

Unit **A** goes *high* after time T_L .
Unit **B** goes *high* after time $T_L + T_H$ and this *high* **Resets** **A** , bringing its output *low*.

What we have just done by using the **Retentive On-Delay** twice can be accomplished by a single **Edge-triggered wiping relay** Then why bother with the two-unit approach? Our goal is not just to tabulate these functions, but to develop a facility in analyzing and applying them.

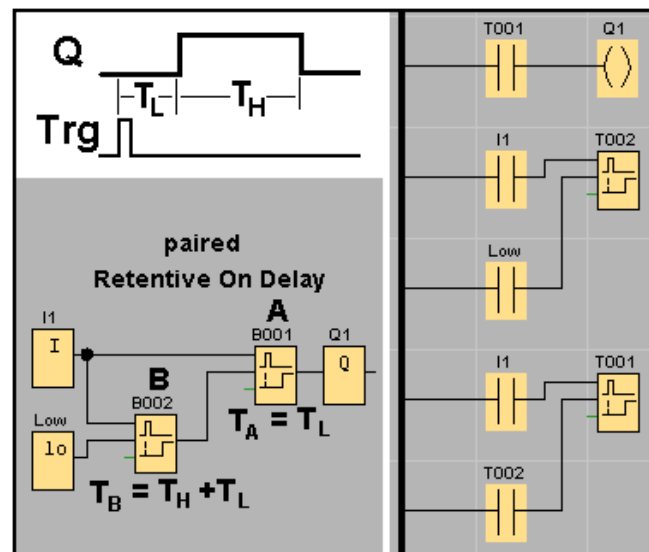


Fig. 6.5 Two **Retentive On-Delay** together

Connection	Description
Input Trg	You trigger the times for the Edge-triggered wiping relay with a signal at input Trg (Trigger).
Input R	The output and the current time Ta are reset to 0 with a signal at input R.
Parameter	TL, TH : The interpulse period T_L and the pulse period T_H . N : number of pulse/pause cycles T_L / T_H : Value range: 1...9. Retentivity set (on) = the status is retentive in memory.
Output Q	Output Q is set when the time T_L has expired and is reset when T_H has expired.

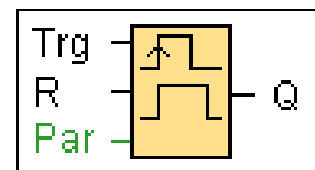


Fig.. 6.6 **Edge-triggered wiping relay**.

The **Edge-triggered wiping relay** has one added attraction; the function can automatically repeat its T_L and T_H transitions up to nine times by setting the N parameter to the desired value. (Remember how to set function parameters: right-click on the icon, and select *Block Properties*) The T_L parameter of either of these two methods may be set to zero, which provides an edge-triggered high output with duration T_L and no delay. Can you think of a situation in which T_H might be set to zero?

As an example of a possible application, if the PLC realizes that a piece of hardware is not working properly. it can signal the operator through a pre-determined series of beeps, generated by this function. (On *boot-up* how does an IBM PC signal the user that the display monitor is not connected?)

Now, back to our motor testing in Problem #2. The circuit shown in Fig. 6.3 took care of the starting sequence for the motor but did not stop it. Therefore, for Problem #2 insert either of the *edge-trigger* circuits ($T_L = 0$, T_L = total testing time, $N = 1$) at the start of the Problem #1 and the problem is solved. (An experienced programmer knows the art of cutting and pasting fragment of previously tested code to construct larger code segments or entire programs)

Task 6.2: Starting Motors

- 1: Create a circuit to solve *Problem #1*, and verify its performance by simulation
- 2: Set up a **Retentive On-Delay pair** with output to **Q1**, and an **Edge-triggered wiping relay** with output to **Q2**. Use **I1** as a common input to both circuits. Use different values for T_L and T_H (same values for both circuits) and by simulation check if the two circuits always have identical outputs.
- 3: Solve *Problem #2*, for a *total test time* of 20.0 seconds.
- 4 Use the **Edge-triggered wiping relay** to produce a series of 7 pulses, each *high* for 4.0 seconds and *low* for 1.0 sec. If possible run the program in hardware and connect the output to a beeper or speaker.

6.5 One Full Process

Problem #3 : A certain process involves three operations, **A**, **B** and **C**. Each is to be run for 4.0 seconds. The sequence of operations is to be **A-B-A-B-C-A-B**

To clarify our thinking, a timing diagram as in Fig. 6.7 can be helpful . Let **Q1** be the control for operation **A**., which must be high for 4.0 sec either at $T=0$ or $T=8$ or $T=20$. This suggests that **Q1** is at the output of a 3-input OR gate Operation A is to be active ($Q1=high$) which suggests three **Edge-triggered wiping relays**.

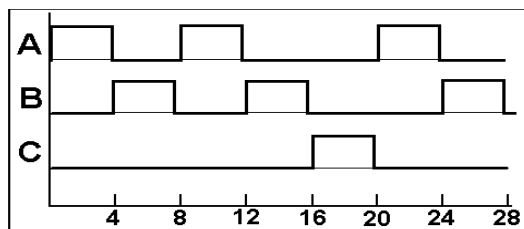


Fig. 6.7 Timing for Problem #3

#1{ $T_L=0$, $T_H=4$ }, #2{ $T_L=8$, $T_H=4$ }, #2{ $T_L=20$, $T_H=4$ }. A similar approach can be used for **B** and **C**. Fig 6.8 shows a [FBD] diagram to solve Problem #3.

There are certain points to be noted in Fig. 6.8. The values for T_H and T_L do not appear on the diagram, as in previous figures. The display of parameters may be turned on or off by clicking: **Tools** \Rightarrow **Options** \Rightarrow **Screen** and check or clear ☐ **Parameters**. You might be interested in exploring some of the other display options. The corresponding [LAD] has fourteen rungs. Ask the program to make the translation for you

Notice also the **Reset** terminals of all the timers appear to have no connections. This was done intentionally to keep the diagram simple. If the **Reset** terminal is shown as unconnected, it defaults to low.

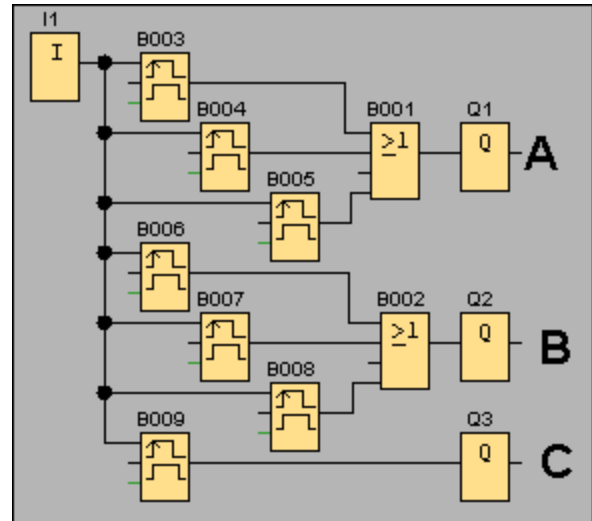


Fig. 6.8 Circuit for Problem #3

Task 6.3: Processes

- 1: Set up, simulate and verify the circuit of Fig. 6.6, a solution for *Problem #3*.
- 2: Have the computer translate your circuit to a [LAD], and identify the timing functions in each version. Notice **OR** gates transform into parallel rows or *rungs*.
- 3: Change the *duration* values of each operation ($A=5$, $B=2$, $C=8$) but keep the same sequence. Create a [FBD] for this modified problem.
- 4: To the circuit just above add an additional output, **Q4**, which is to flash a warning light when the process is finished (9 flashes, 0.7 sec on, 0.3 sec off)

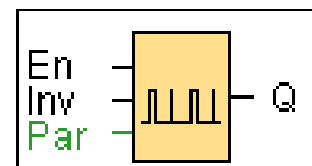
6.6 Turn Indicator

Problem #4 A 10-wheeler trailer truck has pairs of turn lights on either side, which alternately blink ($\frac{1}{2}$ sec on, $\frac{1}{2}$ sec off) to signal a turn. Both sides are not to be blinking at the same time.

Connection	Description
Input En	You enable/disable the asynchronous pulse generator with the signal at input En.
Input Inv	The Inv input can be used to invert the output signal of the active asynchronous pulse generator..
Parameter	TL,TH: You can customize the pulse (TL)/ pause (TH) ratio. Retentivity set (on) = the status is retentive in memory.
Output Q	Q is toggled on and off cyclically with the pulse times T_H, T_L .

Fig. 6.9 Asynchronous pulse generator

Asynchronous Pulse Generator



In the [FBD] circuit for Problem #4 a single **Asynchronous pulse generator** (defined in Fig. 6.9) is used for the blinking of lights on either side, which is **Enabled** by either input through the OR gate. The **INV** input is left unconnected so the default low value is used. T_H and T_L are both set to 0.5 sec to give symmetric flashing,

Both inputs feed the pair of AND gates. Since each has one inverted input, both cannot be open at the same time, which provides control signals for the *left* and *right* turn lights. Each of the outputs, **Q1**,...,**Q4** has its own controlling AND gate, so output can be high only one at a time. Fig.6.10B shows the equivalent circuit in [LAD] . For a number of problems both [FBD] and [LAD] diagrams are presented, to help you develop your ability to think in terms of both languages

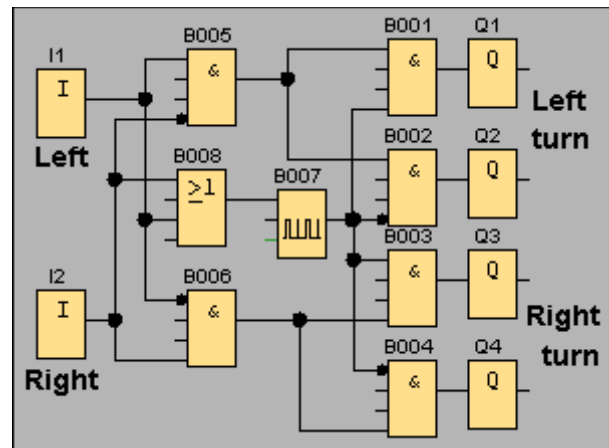


Fig. 6.10A Problem #4 in [FBD]

Task 6.4: Turn indicators

- 1: Set up, simulate and verify the circuit of Fig. 6.10A, a solution for *Problem #4*.
- 2: To the circuit for Problem #4, add an additional input for *roadside parking*, which permits lights on both sides to blink at the same time,

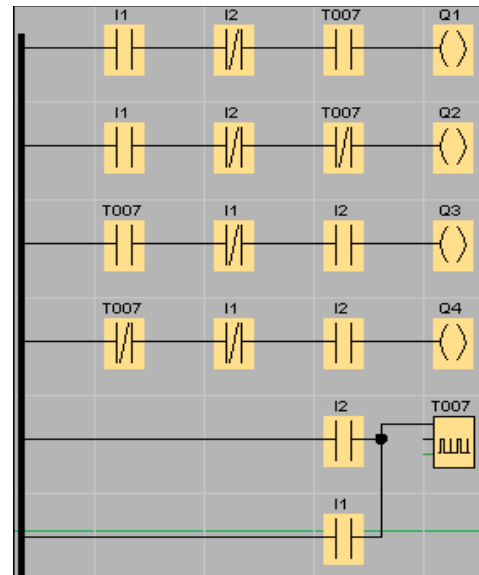


Fig 6.10B Problem #4 in [LAD]

6.7 Motor overload protection

Problem #5 *If a given large motor has not attained full speed within 5.0 sec of starting, it generates an error pulse. In such a case power should be cut off, a flashing warning should be given and no re-try allowed for 10.0 seconds.*

The first step in creating a suitable circuit is to identify *outputs* and *inputs*. Here we have two *outputs*, **Motor** and **Warning** and three *inputs*, **START**, **STOP** and **OVERLOAD**. In past work we started with the [FBD], and then let the program do the translation to [LAD] since [LAD] to [FBD] often results in a messy display.. *Block IDs* are maintained in both translations, but *labels* are never included.

With [FBD] we usually start with *inputs* and trace *power flow* to *outputs*. In [LAD] it is preferable to start with outputs and work back toward the power buss. For the motor to be running (**Motor** output *high*) **STOP** and **OVERLOAD** must be low, and either **START** or **Motor** must be *high*. From this we know at once that the Motor output must have a path back to the Power Bus through a *series* combination of contacts **STOP** , **OVERLOAD** and the parallel combination of **START** and **Motor** contacts This is seen on in Fig. 6.11.

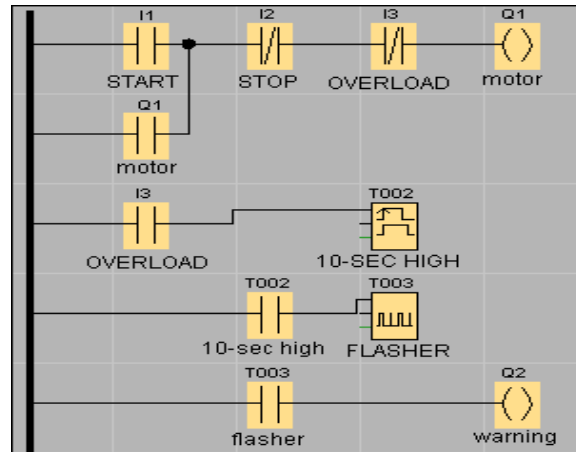


Fig. 6.11 [LAD] for Problem #5

The **Warning** output traces back to the **OVERLOAD** input through the **FLASHER** (*Asynchronous Pulse Generator*) in series with **10-SEC HIGH** (*Edge-triggered wiping relay*). Function icons in [FBD] have both *input* and *output* terminals while in [LAD] only *input* terminals appear (the function *output* is placed in memory, addressable through a *contact*). When used, the function icon must always be the right-most icon on the *rung*. Therefore in [LAD] a logical path from *output* through one or more functions back to *input* must use more than one rung. This is seen in the last three rungs in Fig. 6.11. Fig 6.12 is the program translation from [LAD] to [FBD] while Fig. 6.13 is the same diagram after a bit of editing by the programmer.

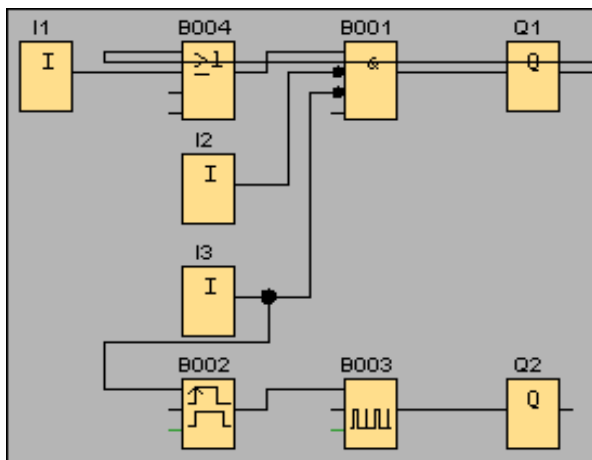


Fig. 6.12 [FBD] Translated by program

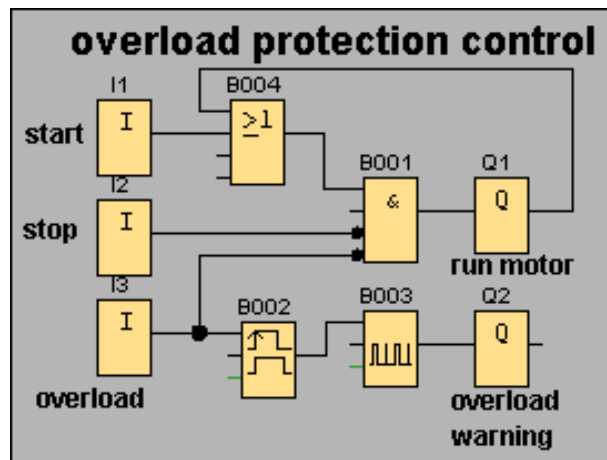


Fig. 6.13 [FBD] created by programmer

For simulation set inputs as **Momentary Pushbutton (make)**. The **motor** may be **started** or **stopped** immediately, But if **overload** is pressed the **motor** stops (if running) and cannot be started again until the 10-sec **warning** has timed out,

Task 6.5: Motor overload protection

- 1: Create directly a [LAD] solution, and then let the program translate it [FBD].
- 2: Simulate the [FBD] version to verify proper operation.. Then rearrange the figure to make it more readable, and add labels. If a printer is available print it.

6.8 A simulated random overload

In real-world applications the PLC may obtain *feedback* from outside at unpredictable moments. The *overload error signal* from the starting motor may or may not come, depending on the load. Recall in Section 5.3 we introduced an *automatic button-pusher* to imitate a feedback signal from outside. In this section we introduce a *random overload* .. It can be used to test more realistically our solution to Problem #5, and provide the occasion to examine a new timing function, **Random generator**,

Connection	Description
Input En	The positive edge (0 to 1 transition) at the enable input En (Enable) triggers the on delay for the random generator. The negative edge (1 to 0 transition) triggers the off delay for the random generator.
Parameter	TH : The on delay is determined at random and lies between 0 s and T_H . TL : The off delay is determined at random and lies between 0 s and T_L .
Output Q	Q is set on expiration of the on delay if En is still set. It is reset when the off delay time has expired and if En has not been set again.

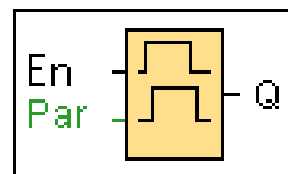


Fig. 6.14 The **Random generator** Special Function

This function is somewhat similar to the level-triggered **On / Off delay**. But now the delay between **Enable** going *high* and output **Q** going *high* is some *random value* between 0 and T_H . Also the delay between **Enable** going *low* and output **Q** going *low* is some other random value between 0 and T_L . We use this function in our simulation.

In the circuit of Fig.6.15 the 5-sec timer is a **Wiping relay (pulse output)** set to remain high for 5.0 sec after **Trg**. The *random timer* is a **Random generator** set to delay going high after **Enable** between 0 and 10.0 sec ($T_H = 10.0$ sec, $T_L = 0$). The 3-input AND goes *high* provided that 1] the *motor* input is *high*, 2] the 5-sec timer is still high and 3] the *random timer* has already gone *high*. This should occur on about 50% of the trials.. A *high* AND output **Sets** the **RS latch** (it was **Reset** when *motor* first went *high*). The flipping of the **RS latch** is converted to a single pulse by the $\&\uparrow$ gate.

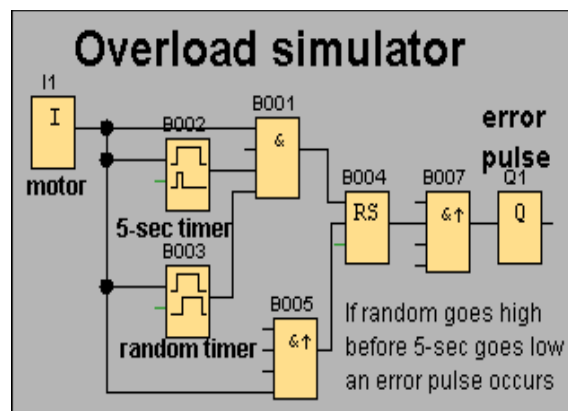


Fig. 6.15 Simulated overload condition

To view the corresponding 8-rung [LAD] click the toolbar icon.



3: Modify the T_H parameter of the **Random Generator** function so that an overload error occurs about 20% of the time. Also try 80%. Is it really random?

interpulse spacing The operation stops when a *high* from **I2**, **stop**, is applied to the **Reset** of B001.

To view the corresponding 12-rung [LAD] click the toolbar icon.



Task 6.7: Random pulse generator


1: Set up the circuit shown in Fig. 6.16, and simulate it. If available, connect the output to both a light and also a beeper, to make the random character more apparent. **Note:** the **ID numbers** of the different blocks are assigned by the program in the order in which you place the icons. Therefore your **ID numbers** may not be the same as in Fig. 6.16, but the same blocks and connectors should appear.

2. Vary the parameters of the two timing gates, to give different ranges of random *pulse width* and fixed *interpulse spacing*

Highlights

An important distinction in timing functions is between **level** and **edge triggering**. Both are started by a 0 to 1 transition at the **Trigger** or **Enable** input. For *level* triggering this initiating pulse must *remain high* for the timer to continue operation; for *edge* triggering the timer operation continues *even if the initial high input is removed*.

Parameters are assigned to particular functions for **duration** and for *on* and *off delays*

A detailed description of each *timer Special Function* may be quickly accessed by selecting the *context-sensitive help* icon, , and then clicking on the function *icon* on a *toolbar*

The help files give the basic action of each function, but how to apply the proper functions to produce a desired result only comes with experience and a detailed analysis of the application.

When using *copy* and *paste* in the Editor windows, the relative positions and connections of copied icons are unchanged, but new *ID numbers* are assigned, since *no two blocks may share the same ID number*.

Programs may be constructed in either [FBD] or [LAD] and program-translated from one format to the other, with identical ID numbers. Either format may be *simulated* in software.

Looking Backwards

1: Add an **Up / down counter** at the output of the **Random Pulse Generator** of Fig. 6.16, so that the *number of pulses* may be counted. Connect the **Reset** of the **Up / down counter** to the **stop** input, so that the count is brought back to zero after each trial.

2: To the modified circuit above add an OR gate so that both the output of the **RS latch** and the **Up / down counter** appear at the **Q1** output. Set the On and Off parameters of the counter to 10 , so the counter output will go high at the 10th count. (Recall Sec. 5.3) At the 10th count the output should remain high until stop is pressed.

3: Create a **betting game** with the circuit as modified above. Use copy and paste to make a second circuit. Call the original **A** and the copy **B**. Remove the **start** and **stop** inputs from **B** and use in place the *start* and *stop* of **A**. so both circuits may be started and reset together.

Next connect the **Up / down counter** output of **B** to the **A Edge-trigger wiping relay Reset** input (use an OR gate to join the two inputs) and connect the **Up / down counter** output of **A** to the **B Edge-trigger wiping relay Reset** input. In this way, whichever circuit is first to count to 10 will lock out the other display and display its own output continuously., declaring it to be the winner!

4: How could you “fix” the *betting game* so **A** has a better chance of winning than **B**, but in a way that the cheating is not obvious?

5: Create a circuit with **Start** and **Reset** inputs and two outputs, **heads** and **tails**, (both may not be *high* at the same time) and **heads** is *high* on approximately 50% of the trials. Describe how you could make this into a “cheating” coin toss.

6: The **Random Pulse generator** of Section 6.9 has constant *interpulse spacing* and *random pulse width*. Modify the program so that *pulse width* is constant and *interpulse spacing* is random.

LOGO! PLC

Chapter 7: Some Odd Functions

The previous chapter was all about time intervals and time delays. In every control process, *time* is a key element, and *LOGO!Soft Comfort*® provides us with a number of additional timing and sequencing tools.

7.1 Pulses per time interval

The standard engineering measurement of frequency is *cycles per second*, or *hertz*. The PLC does not have very good high frequency response, but in many industrial control applications, high frequency measurements are not required. Supermarket checkout lines can be evaluated in terms of customers per five-minute intervals. If this number goes too low, some check-out stations may be shut down, if it goes too high, additional stations should be opened. If each customer checkout generates a single pulse, we are dealing with pulses per time interval. We would hardly wish to measure this quantity in hertz, so it is convenient to select the time interval according to the application. And to do just this, we are provided with the special function, **Threshold trigger**. The formal definition and corresponding icon are shown in Fig. 7.1

Connection	Description
Input Fre	<p>The function counts 0 to 1 transitions at input Fre. 1 to 0 transitions are not counted. Use</p> <ul style="list-style-type: none"> Inputs I5/I6 for fast counts (only available for specific LOGO! devices, see the LOGO! manual): max. 2 kHz. Any other input or circuit element for low frequencies (typical 4 Hz).
Parameter	<p>On: On threshold Range of values: 0000...9999</p> <p>Off: Off threshold Range of values: 0000...9999</p> <p>G_T: Time interval or gate time during which the input pulses are measured. Range of values: 00:05 s...99:99 s</p>
Output Q	Q is set or reset according to the threshold values.

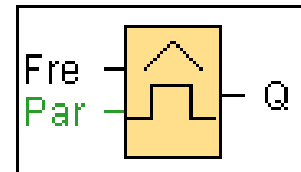


Fig. 7.1 The **Threshold trigger** special function

The pulses to be counter are entered at **Frequency** input, and the time interval, (the time the gate is open) is entered as a parameter **G_T**. Within the function is an internal counter, which counts each pulse during the gate time interval, **G_T**, and then

resets itself to zero for the next interval. The output of the function depends on **fa** (the number of *counts* divided by the *gate time* interval, **G_T**) and the two threshold parameters, **On** and **Off**.

- If the threshold (On) > threshold (Off), then:
 $Q = 1$, if $fa \geq \text{On}$
 $Q = 0$, if $fa < \text{Off}$.
- If the threshold (On) < threshold (Off), then $Q = 1$, if
 $\text{On} \leq fa < \text{Off}$.

These operate in just the same manner as for the **Up / down counter** discussed in Sec. 5.3 and Fig 5.5.

To try out this function we generate a **2.0** hertz square wave using the **Asynchronous Pulse Generator** (set both *pulse width*, T_H , and *interpulse spacing*, T_L , to 0.25 seconds). For the **Threshold trigger** set the *gate time*, **G_T**, to 1.00 seconds and both **On** and **Off** to 1. Fig. 7.2 is a [FBD] for this circuit.

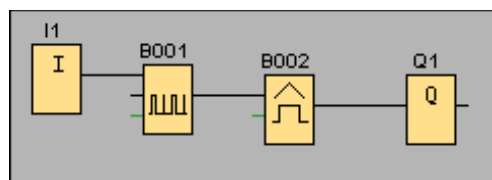


Fig. 7.2 Test of **Threshold trigger**

When this circuit is simulated, we expect the **fa** value to be 2 and the output to be high. But occasionally **fa** is shown as 1. Why? The problem lies in the duration of the Scanning Cycle mentioned in Section 1.8.

Task 7.1: Testing the Threshold trigger

A: Enter the [FBD] circuit shown in Fig. 7.2 and verify its performance by simulation.

B: For the **Threshold trigger** block change the parameters: **G_T = 10.0 seconds**. **On = Off = 19**. Make no change in the **Asynchronous Pulse Generator** block. Now the gate time is ten times longer while the input pulses come at the same rate of 2 pulses per second. We expect **fa** should be 20 (20 pulses per 10 seconds) but when you *simulate* this is still not always what happens,

C: To refresh your skills with ladder diagrams, sketch a [LAD] and then have the program make a translation of your [FBD]. How does your sketch compare with the [LAD] generated by the computer?

7.2 The duration of the Scanning Cycle

The PLC *Scanning Cycle* was first discussed in Section 1.6, but an exact time value was not given. We now have the tools to measure this value. In Section 5.3, *The Button-Pusher*, a simple circuit was displayed in Fig. 5.0 in [FBD] and [LAD]. During each *Scanning Cycle* the NOT gate inverts the value stored in **M1**. Therefore if we take

an output signal from **M1**, we have a symmetric square-wave signal with frequency *one half*, and a period *twice*, that of the *Scanning Cycle*. So all we need do to determine these values is to feed this signal into a **Threshold trigger** with an appropriate *gate-time*, **G_T**.

Fig. 7.3. shows the display during the *Scanning Cycle*. The selected *gate-time* is 20.00 seconds. The **On** and **Off** parameters are not significant since we are not monitoring the output, but only the pulse count value. The figure captured the situation part way through the 20.00 second *gate-time*, and indicates that during the prior *gate-time* cycle of 20.00 seconds, there were **196**

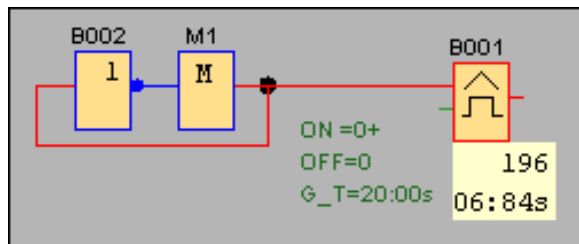


Fig. 7.3 Measuring the *Scanning Cycle*

counts corresponding to $2 \times 196 = 392$ *Scanning Cycles* per 20.00 seconds or a *Scanning frequency* of 19,6 hertz. and a **period** of $1/19.6 = 0.051$ seconds. From such a value for the *Scanning Cycle*, it is easy to see why this PLC cannot deal reliably with time intervals shorter than about 0.05 seconds.

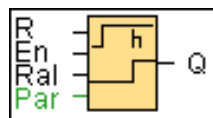
Task 7.2: Measuring the *Scanning Cycle* period

A: Enter the [FBD] circuit shown in Fig. 7.3 and verify its performance by simulation. Make 5 trials and note if the period always has the same value.

B: For increased accuracy, increase the *gate-time* to **99.99** seconds and make an additional measurement of the duration of the *Scanning Cycle*

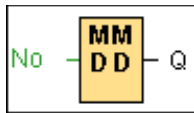
7.3 Longer time intervals

When configuring function *Parameters* of all the timing special functions considered so far, time values shorter than 0.05 seconds really do not apply, given the period of the *Scanning Cycle*. You may have noticed that the longest selectable time interval for many of these is 99.99 hours. But there are several other timing functions that cover much longer periods, which are briefly discussed below.



Hours Counter

Perhaps you may recall that the speedometer of an automobile displays not only speed but also distance traveled. One distance display can be set of zero at any time, the other keeps on counting for the life of the vehicle. The special function **Hours Counter** is somewhat like that, but in terms of *elapsed time* rather than *distance traveled*. This function is useful in setting up maintenance schedules for plant equipment. The time values remain even when the equipment is turned off and are updated as long as the **Enable** input is high. The **Q** output goes high after an *adjustable maintenance interval* or *total elapsed time* has expired. Full details are found in the **Help File**.

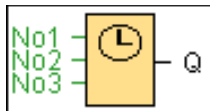


Yearly timer

This function has no **Trigger** or **Enable** input, which means that it is always active. It has two parameters, **On** and **Off**, determining the *month* and *day* of the year for the output to go *high* or *low*. The **On** and **Off** selection panel for the parameters is shown in Fig. 7.4. You may set the *month / day* numerically, or click the calendar *icon* to display a large calendar page. It is possible to select an **Off** date that is earlier than the **On** date since the timing rolls over year by year. If you check the **Every Month** box, you only select the day for **On** and the day for **Off**, and the selection is valid for every month of the year.

Fig. 7.4 Setting the **Yearly Timer** parameters


Suppose, as an enlightened manager, you wanted happy music to played in the factory every pay-day, the 15th and 30th of every month. Use an OR gate to combine the outputs of two **Yearly Timer** special functions, both checked for **Every Month**. One is set for **On=1, Off=2** and the other for **On=15, Off=16**.



Weekly Timer

This special function has no **Trigger** or **Enable** input; it is always active. It may be considered as a set of three independent *On/Off timers* ORed to give a common output, **Q**. The **On** and **Off** times are set in terms of hour and minute and set for one or more day of the week. Fig. 7.5 shows the settings for **Cam 1**. The output will be high every **MWF** from 7:25 to 8:20 AM. Clicking on the appropriate tab may configure the other two **Cams**. The outputs of two or more **Weekly timers** may be ORed in common.

Fig. 7.5 Setting parameters for **Weekly timer**

Programs using these functions may be *simulated*. But it is not convenient to come back every Monday, Wednesday or Friday at 7:25 AM to check on what the program is doing. An easier way is to tell the computer that now it is a different month, day or hour. Both the IBM computer and the LOGO! software maintain *real time* clocks, quite independent of the *Scanning Cycle*. The *Simulation toolbar* shows the software time, and this may be adjusted by clicking on the **clock icon**.  Adjusting software time does not adjust the IBM time.

Task 7.3: Special Timer Functions

A: Configure the **Yearly timer** for a *high* output on the 15th and 30th of each month. Then in *Simulation mode* vary the *real-time date* for the 14th, 15th, or 16th of the month to verify your configuration.

B: Configure the **Yearly timer** to go *high* for June 1 to 15. Verify in simulation mode by changing the real-time date.

C: With one (or more) **Weekly timers**, configure a circuit that gives a high output on MWF at 7:30 to 9:00 AM and also 1:30 to 3:0 PM, and on TTh from 8:00 to 11:00 AM and 3:00 to 5:00 PM. Verify the circuit in *Simulation mode*.

D: Create a circuit to turn on your house lights every evening at 7:30 PM and turn them off again at 5:45 the following morning. Verify the circuit in *Simulation mode*.

7.4 The Shift Register

In most computer languages there is *shift* command, moving each bit of a byte to the left or right. *LOGO!Soft Comfort*[®] provides a somewhat similar special function called **Shift Register** shifting the bits of the register byte left or right. Each register bit is associated with a pre-defined memory locations, **S1**, **S2**, ... , **S8**, which may be read from, but not written to, by the user program. The full description is shown in Fig. 7.6.

Connection	Description
Input In	The function when started reads this input value.
Input Trg	The SFB is started with a positive edge (0 to 1 transition) at input Trg (Trigger). A 1 to 0 transition is irrelevant.
Input Dir	You define the shift direction of the shift register bits S1...S8 at the Dir input: Dir = 0: shift up (S1 >> S8) Dir = 1: shift down (S8 >> S1)
Parameter	Shift register bit that determines the value of output Q. Possible settings: S1 ... S8 Retentivity set (on) = the status is retentive in memory.
Output Q	The output value corresponds with the configured shift register bit.

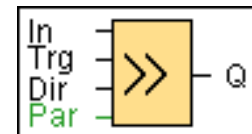


Fig. 7.6 The **Shift Register** special function

As mentioned above the Shift Register function controls the set of eight one-bit memory locations. The single selectable parameter equates output **Q** with one or other of the eight **S** locations. In terms of digital logic gates this function is a *serial in – parallel out shift register*. An easy way to get the feel of this function is to set up the circuit shown in Fig. 7.7. Set the **Trig** input, **I2**, as *Momentary pushbutton (make)*, and the other two inputs as *Switch*. and run the program in *Simulation* mode.

Start with **In** and **Dir** both *low*, and **Shift register bit** = 5. Pressing **Trig** seems to have no effect. Actually it is shifting zero values through the **S** memory locations. Next set **In high**: still no change. Now press **Trig** once; the *high* of **In** moves into **S1** (and the former zero in **S1** moves into **S2** and so on, but this does not change the display.) Notice that the **S1** icon now changes from *blue* to *red*, indicating it has gone *high*. Set **In low** again, and press **Trig** a few more times, and notice how the highlighted **S** shifts to the next higher location. Whenever **S5** is highlighted the **Q1** output also goes *high*. If you change the **Dir** input value the direction of the shift changes.

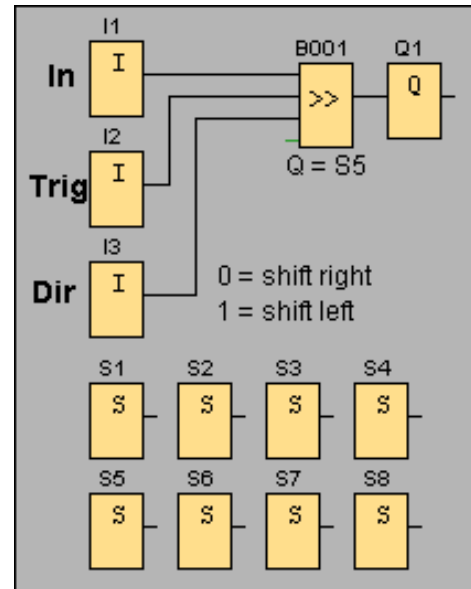


Fig. 7.7 Shift Register test

In Fig. 7.7 notice that the **S** icons have an *output stub* on the right but no *input stub* on the left. This is because the programmer cannot directly write to any **S** location, but can only read from it. Compare this with an **M** icon, with *stubs* on both sides, for the program can both read from and write to any **M** memory location or **flag**. The **Shift Register** function may be used only once in a program



To view the corresponding 12-rung [LAD] click the toolbar icon.

Task 7.4: Playing with the Shift Register.

A: Set up the circuit shown in Fig. 7.7, and go to *Simulation* mode. What happens if **In** is *high* each time **Trig** is pressed? What happens if **In** is *low* and **Trig** is pressed eight or more times? .

B: If **Dir** = 1 and **In** = 1 and **Trig** is pressed, does **S1** go high or is it **S8**?

C: Explain how to set all the **S** locations *high* at the same time. Explain how to reset all the **S** locations *low* at the same time. Explain how to set **S3**, **S4** and **S5** high and the others low.

D: Review the idea of **Retentivity** and **power interruption** explained in Section 4.6 Experiment with **Retentivity** on or off for the Shift Register, and describe the behavior after a *power interruption*,

7.5 The Shift Register as a rotary switch

In a typical *rotary switch* a single input is connected in sequence to one of a given number of outputs as the switch knob is rotated. We can duplicate an eight-position *rotary switch* with a **Shift Register** and a number of AND gates. We have seen how to set *high* in sequence one **S** location at a time by pulsing the **Trig** input. If we connect each **S** location to one input of its own AND gate, this opens the AND gates one at a time, providing the desired sequential switching. The circuit of Fig. 7.7 can be a beginning, but it has to be modified. We wish the shifting direction always to be **up** or **right**, so the **Dir** input may be left un-connected, which defaults to a *low* (shift *up* or *right*.) However we wish the shifting to recycle continuously to avoid the need of the **In** input of Fig. 7.7. So we must find a way to have **S1** set *high* at the start of the program, and for the selected **S** go back to the beginning once it reaches the end. Fig. 7.8 presents one solution.

In this circuit the *switch input* is **M2**, and the *four outputs* are the four AND gates, B004, B005, B006, B007. The **M2** input could be replaced by an **I2** input and the outputs of the four AND gates could be connected to **Q1, Q2, Q3, Q4** in a particular application. The *output selection* is made by pulsing the **I1** control input. The four icons, **S5, S6, S7, S8** are *not* part of the switch configuration. They are included to give additional insight while in *Simulation* mode.

Before we go into the explanation of how the switching is done, something should be said about the icon pairs,



At times the wire connections between icons in a [FBD] can be somewhat confusing, especially if they cross one another. In the diagram each of the four AND gates has one input connected to the **M2** flag. After these connections were made with the wire tool, each was *separated* using the *separation* option. To do this move the selection tool, □, to the wire you wish to separate, right-click, and select **Separate** and the *arrow icons* appear at the end connections of the wire. The *arrow head* icon gives the address of the destination and the *arrow tail* gives the address of the source. Thus the arrow head icon B004/1 references *Block B004*, input 1, and the corresponding arrow tail references **M2**.



To view the corresponding 15-rung [LAD] click the toolbar icon.

Now, back to the *selector switch*. Our *secret weapon* is the **M8** flag. At the start of the *first* Scanning Cycle **M8** has been set *high* by the computer, but is reset to *low* for the second cycle. After that **M8** behaves as any other memory location. Referring to Fig. 7.8, during the first Scanning Cycle **M8** is high, making both the **Trig** and **In** signals to

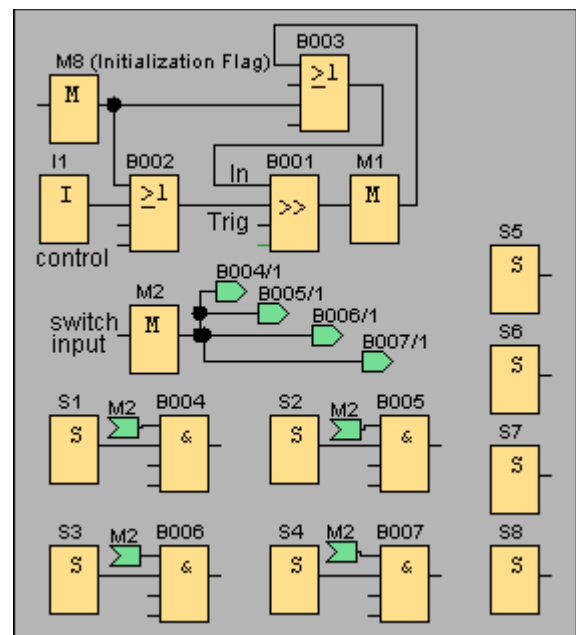


Fig. 7.8 A 4-pole rotary switch circuit

the **Shift Register** block *high*, so **S1** is set *high*. During all following cycles **M8** is *low* and plays no further role in the program. Any further **In** signals must come from the **I1** control input. **Q=S4** had been set for this circuit so on the third **I1** impulse **S4**, the output of the **Shift Register**, **M2** and **In** all go high. On the next **I1** impulse **S1** again goes *high*. It is helpful to configure and simulate this circuit and watch it in operation.

Task 7.5: A Rotary Switch

A: Set up the circuit shown in Fig. 7.8. For the **Shift Register Parameters** let **Q=S4**, **Retentivity=Off**. Go to *Simulation* mode. Notice the icons on the Simulation Taskbar. On-going simulation is stopped by clicking the *red square*, and restarted by clicking the *green triangle*. Since *Retentivity* has been turned off, clicking the *power interruption* icon re-initializes the system,

B: After *starting* or *re-initializing*, is **S1** always *high*? (The first Scanning Cycle is so fast that we cannot observe its action!).

C: Pulse the **I1** control input several times and observe what happens. Does a high continuously cycle through **S1** to **S4**? Do the states of **S5** to **S8** have any effect on the action of our 4-position switch?

D: Modify this circuit to become a **6-position** switch.

Problem #1: A mixing tank is filled for 5.00 seconds from pipe **A** and for 2.00 seconds from pipe **B**. The contents is then stirred for 10.00 seconds and then drained for 8.00 seconds. The cycle is to be continuous until a “last cycle” signal is set.

As usual we start by identifying outputs and inputs. We need four outputs: **Q1** = open pipe **A**, **Q2** = open pipe **B**, **Q3** = stir liquid, **Q4** = drain tank. We also need one input, **I1**, to set the “last cycle”. Each cycle ends with the draining of the tank. One solution is presented in Fig. 7.9.

Four separate processes are involved: **A-fill**, **B-fill**, **Stir**, **Drain**. A **Shift Register** schedules the processes, each of which uses for timing its own **Wiping relay (pulse output)** triggered by its own **S** flag. When its output goes low, a NAND (edge) gate produces a pulse, *high* for a single Scanning Cycle, which goes back to the 4-input OR gate. This gate takes the

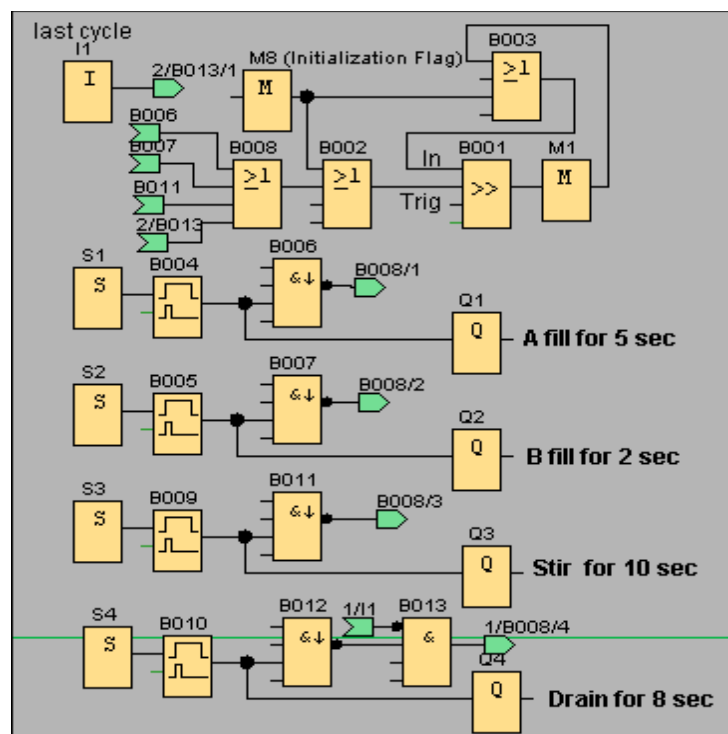


Fig. 7.9 Circuit for Problem #1

place of the **M2 flag** in Fig. 7.8. As each of the processes ends it advances the *rotary switch* by one step. Notice how the AND gate in the final *drain* process can prevent the *rotary switch* from advancing, if the *last cycle I1* input is held *high*.

To view the corresponding 21-rung [LAD] click the toolbar icon.



Task 7.6: A continuous process

A: Set up the circuit shown in Fig. 7.9. For the **Shift Register Parameters** let **Q=S4**, **Retentivity=Off**. For each of the four **Wiping relay (pulse output)** functions set the appropriate **T** value for the process. Then *simulate* the program.

B: Does the operation begin with pipe **A**, as soon as the simulation begins? Does the process repeat over and over again? If **I1** is held high will the process stop after the tank has drained? After “last cycle” is the drain switch closed or left open?

C: Experiment with the *power interruption* icon. Does the process stop completely on power interruption and restart from the beginning when power returns? With frequent power interruptions could pipe **A** fill several times before the tank is drained? Change the **Shift Register** parameter to **Retentivity=On**. In this case is it possible for the tank to overflow due to *power interruptions*?

D: With this circuit, pipe **B** opens only after pipe **A** has closed. Modify the circuit so **A** and **B** start filling together (each is to retain its original filling time). Will it be necessary to change the original **Q=S4** of the **Shift Register**?

Highlights

The special function **Threshold trigger** counts incident pulses per unit time. Pulses may be periodic or random and the programmer sets the unit time or gate time. Function output is determined by the current *count* and the *On* and *Off* parameters. With this function we can determine the period of the Scanning Cycle, which is approximately $\frac{5}{100}$ seconds.

The special functions **Hours counter**, **Yearly timer** and **Weekly timer** control longer duration time intervals.

The special function **Shift register** may be configured to simulate a *rotary selector switch*.

Long traces between [FBD] blocks may be conveniently cut, to improve readability. Identifying information is placed at both connections of the original trace.

Looking Backwards



1: In Section 7.1 we used the special function, **Threshold trigger** to measure the output frequency of the **Asynchronous pulse generator** configured as a 2.00 hertz square wave. It is also possible to use a *counter* and *timer* function to make the same measurement. Connect the output of an AND gate to the **Cnt** input of an **Up/down counter** (connect the **Reset** input of the counter to input **I1**, *Pushbutton momentary high*). To one *input* of the AND gate connect the output of the **Asynchronous pulse generator** (connect its **Enable** input to an **[hi]** block so that it continuously generates the 2.00 hertz square wave.) To the other *input* of the AND gate connect the output of an **Edge-triggered wiping relay** configured to produce a single 10.00 second *high* output ($T_L=0$, $T_H=10.00$ seconds.) Connect its **Trg** input to **I2** *Pushbutton momentary high*. With this configuration the 2.00 hertz square wave can reach the counter via the AND gate for exactly 10.0 seconds. In theory the Up/down counter should record exactly 20 pulses. In practice it is often less. Make at least five trials and compare the count for each (briefly press **I1** and **I2** before each trial.)

2: In #1 above, no mention was made of connecting the **Up/down counter** output to an output **Q1**, for we were only interested in the counter value in *simulation* mode. Modify the program so that the output of the **Asynchronous pulse generator** is connected to **Q1**. Then download the program to a hardware LOGO! unit, and measure the output **Q1** with a *frequency counter* reading to a fraction of a hertz, or to an *oscilloscope* (remember LOGO! output is a relay.) In this way you can verify the accuracy of the 2.00-second square wave.

3: For increased accuracy of the count in #1 above, increase the T_H value of the **Edge-triggered wiping relay** to **99.99 seconds**, its maximum value. Theoretically, the final count should be 200. Determine its actual value.

4: In the circuit of #1 remove the **Asynchronous pulse generator**, and in its place use an oscillator circuit made from a NOT gate and **M1**, which should have a frequency just one half that of the *Scanning Cycle*. Use a T_H value of **99.99** seconds for best accuracy to the period of the *Scanning Cycle*.

5: Describe what happens if you try to insert into your circuit diagram more than eight **S** icons or more than one **Shift Register** icon.

6: Experiment with **Cut / Join**. Create a new [FBD] with only two blocks, **I1** and **Q1**. Join them with the wire tool. Then move the *select* tool, , to any part of the connecting wire, right-click, and select **Separate**. To restore the original wire, move the *select* tool to either *arrow* icon, left click, and select **Join**, and the wire returns. Experiment with the *select* tool on either *arrow* icon, and select **Go to Partner**. Can you move about either of the arrow icons the same way as you move any block? Click the **Cut / Join** icon  on the *Tools* toolbar, which gives you a scissors tool to cut wires. Experiment with it.

LOGO! PLC

Chapter 8: Digital Control Circuits

Before moving on to analog inputs and functions it may be helpful to apply in concrete situations some of the concepts already considered.

8:1 2-station Lamp Control

Problem #1 Design a household stairway light which may be turned on or off by switches at the head or foot of the stairs.

A common circuit using two *single-pole double-throw (SPDT)* switches is shown in Fig. 8.1 Although one would hardly consider using a PLC in such a situation, just for exercise we try.

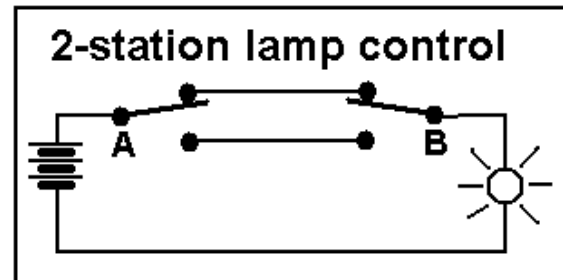


Fig. 8.1 2-station lamp control

No **SPDT** switches are among the LOGO! icons. However we can use AND and OR gates as shown in Fig. 8.2. In the [FBD] it can be seen that if both inputs are *high* or both are low, Q1 is also low (the light is off.)

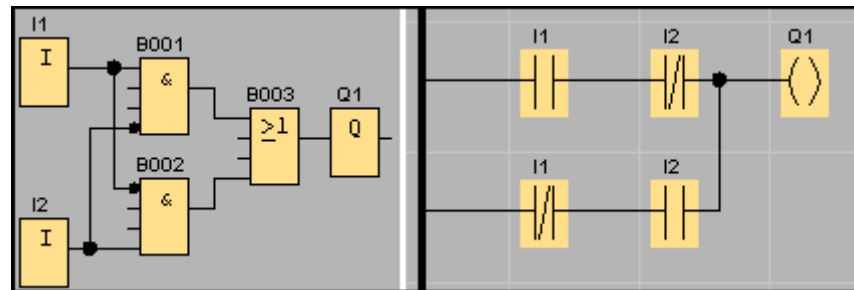


Fig. 8.2 One approach to 2-station lamp control

Recall that the **X-OR** (*exclusive OR*) gate has just this property, so an alternate approach using a single X-OR gate is shown in Fig. 8.3

Problem #2: *Three bedroom doors open onto a corridor in which there is a single light. By the door of each room is a switch that can turn on or off the corridor light.*

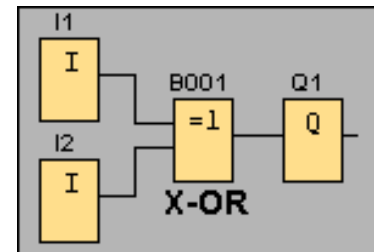


Fig. 8.3 An X-OR version

A first attempt to solve this problem might be extending the [LAD] of Fig.8.2 using three rungs and three contacts on each rung. But this does not seem to work out and the program provides no 3-input X-OR. Basically we wish the control by each room to be quite independent of the control by other rooms. Look back at Section 6.2 for the *push-on push-off* switch, in which the input is a *momentary pushbutton (make)* and the output of the included **R S** circuit switches between *high* or *low* on each contact closure. So for Problem #2 we may use the circuit of Fig. 6.2 but instead of a single input, use three inputs feeding an OR gate and then on to the *edge-triggered AND* gate, as shown in Fig. 8.4

The **RS** special function has two inputs, **Set** and **Reset**, and the additional blocks surrounding it make the circuit single-input to the edge-triggered AND block. However *LOGO!Soft Comfort* offers the special function **Pulse Relay**, a pre-packaged version of the *RS flip-flop* with a single input.

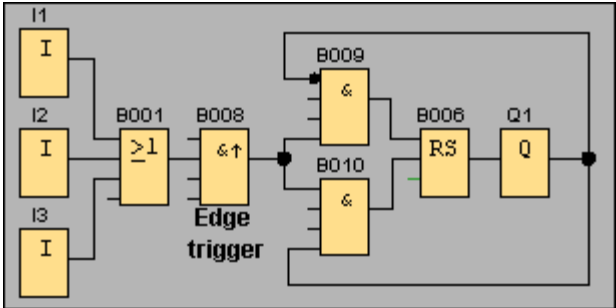


Fig. 8.4 One solution to Problem #2

Connection	Description
Input Trg	You switch output Q on or off with a signal at input Trg (Trigger) input.
Input S	A one-shot at input S (Set) sets the output to logical 1.
Input R	A one-shot at input R (Reset) resets the output to logical 0
Parameter	Selection: RS (input R priority), or SR (input S priority) Retentivity set (on) = the status is retentive in memory.
Output Q	Q is switched on with a signal at Trg and is reset again at the next Trg pulse, if both S and R = 0.

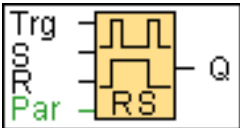


Fig. 8.5 The special function **Pulse Relay**

This **Pulse Relay** function conveniently provides the service of an RS circuit with the choice of either single or double input. It will also provide us an alternate and more compact solution to Problem #2

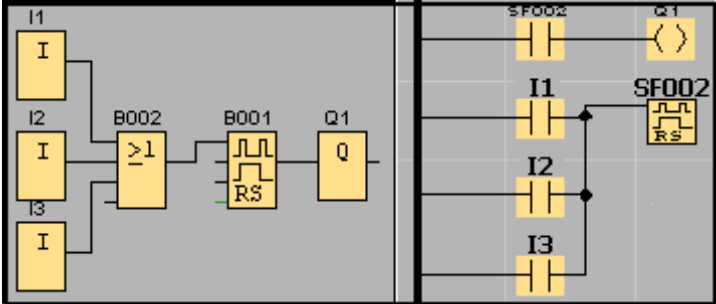


Fig. 8.6 Alternate solution

Task 8.1 Corridor lights

Solve Problem #2 if there are *eight* rooms along the corridor and only one light. **Hint:** can you design the equivalent of an *8-input OR* gate?

8.2 Paging system

Problem #3: Design a paging system for a large hospital to call any one of 8 doctors to the emergency room, using chimes. One chime for Doctor A, two chimes for Dr. B, and so on up to 8 chimes for Dr. H.

It is natural to start with one *output*, **Q1**, providing the chime signal, a ¼ second high signal, and eight *inputs*, **I1** to **I8**, one for each doctor. For simulation, these may be

configured as *momentary pushbutton (make)*. Fig. 6.6 of Section 6.4 shows that an **Edge-triggered wiping relay** can provide up to 9 pulses, of duration T_H and spacing T_L , which seems ideal for this application. Set parameters $T_H = T_L = 0.25$ seconds, and the N value corresponding to the doctor's assigned number. Connect the *output* of each timing block to **Q1** through OR gates. The details are left to the reader.

Problem #4: In the hospital of Problem # 3 doctors F, G and H were complaining that it is distracting to have to count so many chimes, so the administration asked for a $\frac{3}{4}$ second pause after the fourth pulse.

For Drs. A to D no change is needed. For the others, the first four pulses are finished in 2.00 seconds and the next set should start $\frac{3}{4}$ seconds later, or 2.75 seconds after the input pulse. Therefore for the last four doctors we use two additional **Edge-triggered wiping relays**, as shown in Fig. 8.7, **B11** to provide the delay and **B14** to produce the next set of pulses. Recall from Fig. 6.1 that with the **Edge-triggered wiping relay** the T_L time precedes the T_H time. Note that **B11** produces no sound, since it is not directly connected to the OR gate **B12**; its only function is to delay the starting of the second group of pulses. The creation of an [LAD] is left as an exercise for the reader.

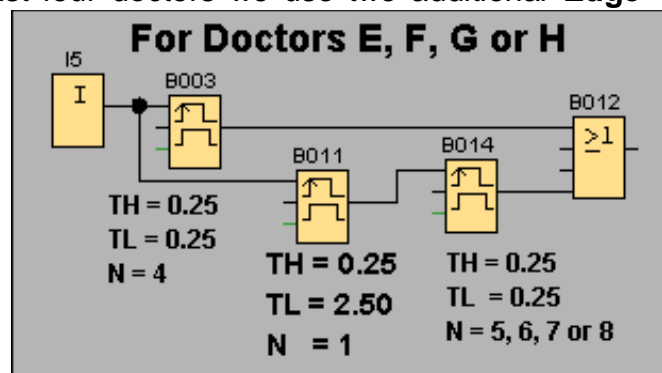


Fig. 8.7 For the latter 4 doctors of Problem #4

Problem #5 This time it's the nurses in the hospital who are complaining that there is too much noise, so the administration decided to use a pattern of no more than three musical tones; short tones for $\frac{1}{4}$ second, long tones for $\frac{3}{4}$ seconds, with a $\frac{1}{4}$ second pause in between. The calling codes are to be:

$A = \bullet$, $B = \text{—}$, $C = \bullet \bullet$, $D = \text{—} \text{—}$,
 $E = \bullet \bullet \bullet$, $F = \text{—} \text{—} \text{—}$, $G = \bullet \text{—}$, $H = \text{—} \bullet$

For Drs. **A** to **F** a single **Edge-triggered wiping relay** is sufficient, with a corresponding adjustment of parameters T_L , T_H and N . For Drs. **G** and **H** two such relays are needed and the T_H will be different for each.

Task 8.2 Paging Systems

- A:** Create and simulate an [FBD] and [LAD] for Problem #3
- B:** Create and simulate an [FBD] and [LAD] for Problem #4
- C:** Create and simulate an [FBD] and [LAD] for Problem #5

8.3 Linear motion

Many processes involve back and forth motion over a limited range; the opening and closing of an elevator door, the positioning of a drill bit over a printed circuit board, the windshield wiper on an automobile. The simplest case is *linear motion at constant speed*. For this two control outputs are needed, **Move–Stop** and **Left–Right**. And because the motion range is *limited*, two *feedback inputs* are needed, **Right–limit** and **Left–limit**. In addition there may be additional inputs for things like *Process–start* or *Emergency–stop*. Look back at Section 5.1 for a simpler version of *linear motion*. In studying such processes software simulation is not quite adequate; we need some physical device controlled by the PLC outputs, which in turn provides meaningful feedback. For this reason, with the packaged LOGO! unit described in this manual a simple hardware module is supplied, as shown in Fig. 8.8.

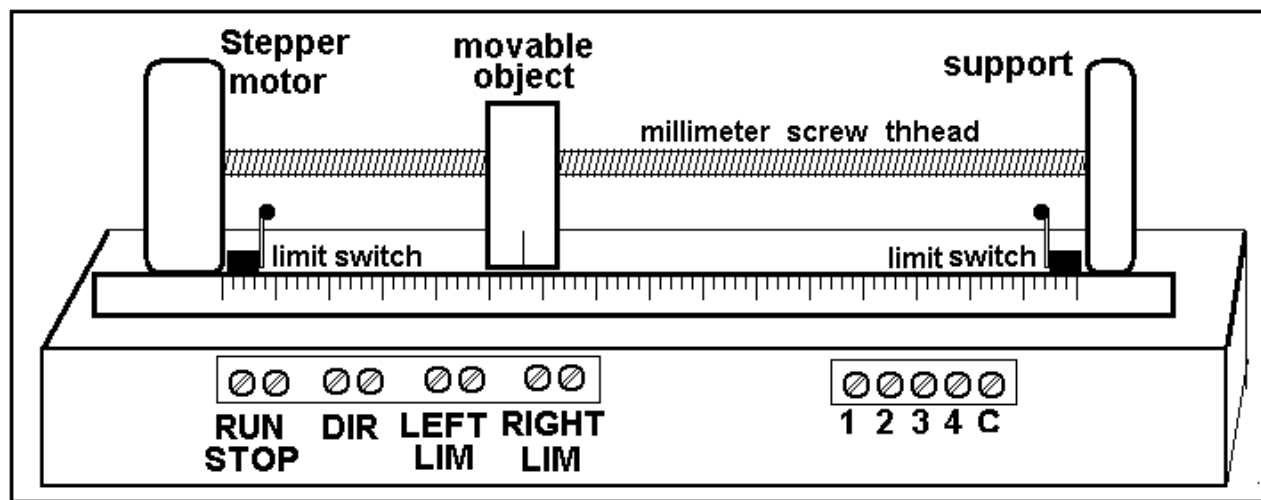


Fig. 8.8 Linear motion module

The linear motion module pictured in Fig. 8.8 contains its own power source, so the relay outputs of the LOGO! unit may be connected directly to **RUN STOP** or **DIR**. The two *limit switches* are **N.O.** It is also possible to run the stepper motor in single step mode as will be explained later (connect to **C** in succession each of the numbered terminals **1, 2, 3, 4**)

Problem #6 Create a program to move continuously the movable object (module block) back and forth between the two limit switches.

Our program should have three inputs, **I1** to start the process, **I2** and **I3** connected respectively to the **LEFT LIM** and **RIGHT LIM** terminals on the *linear motion module*. Connect output **Q1** to **RUN STOP** and **Q2** to **DIR**. The linear motion module is designed so that if the **DIR** terminals are *open* the motion is toward the *left*. Each time the moving block briefly touches a limit switch, the block direction should change and continue in this

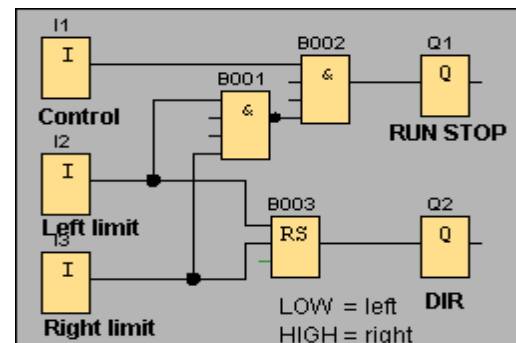


Fig 8.9 Circuit for Problem #6

new direction until the opposite limit switch is touched. Since direction is to be remembered even after a limit switch goes *low*, this suggests an **RS** function. Of course the *movable block* could never touch both *left* and *right limit switches* at the same time. However curious fingers can fool around with these switches, so it is a good idea to stop the motion at once if both limit switches are high at the same time. The circuit shown in Fig. 8.9 is one solution to Problem #6. The creation of an [LAD] solution is left to the reader as an exercise.

Problem #7 *Modify Problem #6 to that after touching either limit switch, the module block pauses for five second flashing a warning light before continuing.*

The *inputs* remain the same but now *output Q3* is to control the *flashing light*. Since the new action is to take place when either *limit switch* goes *high*, connect an **OR** gate to these limit inputs. Since the flashing is to run for 5.0 sec, we use an **Edge-triggered wiping relay** to provide a 5.0 second interval. During this interval *RUN STOP Q1* should be *low* and an **Asynchronous pulse generator** should be enabled to produce alternate *high* and *low* flashing at *Q2*. (Look back at Fig. 6.9 to refresh your memory on this function) An acceptable solution is shown in Fig. 8.10

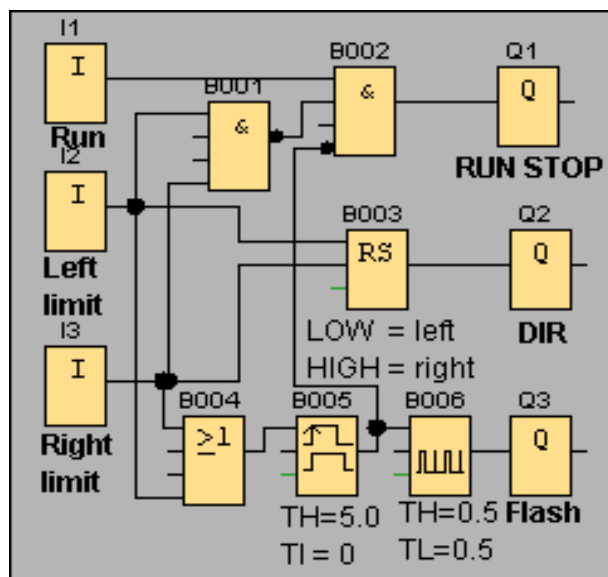


Fig. 8.10 Circuit for Problem #7



To view the corresponding 9-rung [LAD] click the toolbar icon.

Problem #8 *Create a program for the linear motion module that first moves the block to the left end, then flashes a light for 5.0 seconds, next moves the block back and forth twice, and ends with another 5.0 seconds of flashing*

There are four basic operations involved: *Start*, *Flash*, *Move Left* and *Move Right*, but several of these are repeated more than once. Counting repetitions, there are eight steps following initialization. One approach is the use a **Shift Register** to move through the eight steps. As each step ends, it signals the **Shift Register** to call the following step.


Initialization begins when **I1** goes high to **start** the process. This produces a brief pulse at both the **In** and **Trig** ports which sets **S1** high. The **Dir** port of the **Shift Register** (not the **Dir** input on the *linear motion* module) is un-connected, so all shifting is toward the *right* or *upwards*

The *Move Left* operation occurs in steps #1, #4 and #6. Therefore **S1, S4** and **S6** must feed an OR gate that starts the action. It must do four things; reset **Q2** to low for motion toward the *left*, set **Q1 high** to **Run** the motion, **stop** the motion when **I2**, the *Left limit*, goes *high*, and finally send a pulse to the **Trig** port of the **Shift Register** to advance to the *next step*.

The *Move Right* operation occurs in steps #3 and #5, so either **S3** or **S5** can start the operation which sets the block **Dir** toward the *right*, starts the motion and continues it until the *Right limit* goes *high* and then sends a pulse to **Trig** to move to the next step.

The Flash operation occurs in steps #2 and #7. An **Edge-triggered wiping relay** gives a *high* output for 5.0 seconds, **Enabling** an **Asynchronous pulse generator**, which produce the *flashing* signal at **Q3**.

Notice that these three operations end with an output going *low*. It is therefore convenient to use a *negative edge-triggered AND gate*, **&•↓**, discussed in Section 4.6, to produce the **Trig** signal for the **Shift Register**. The full circuit is shown in Fig. 8.11.

To view the corresponding 25-rung [LAD] click the toolbar icon. 

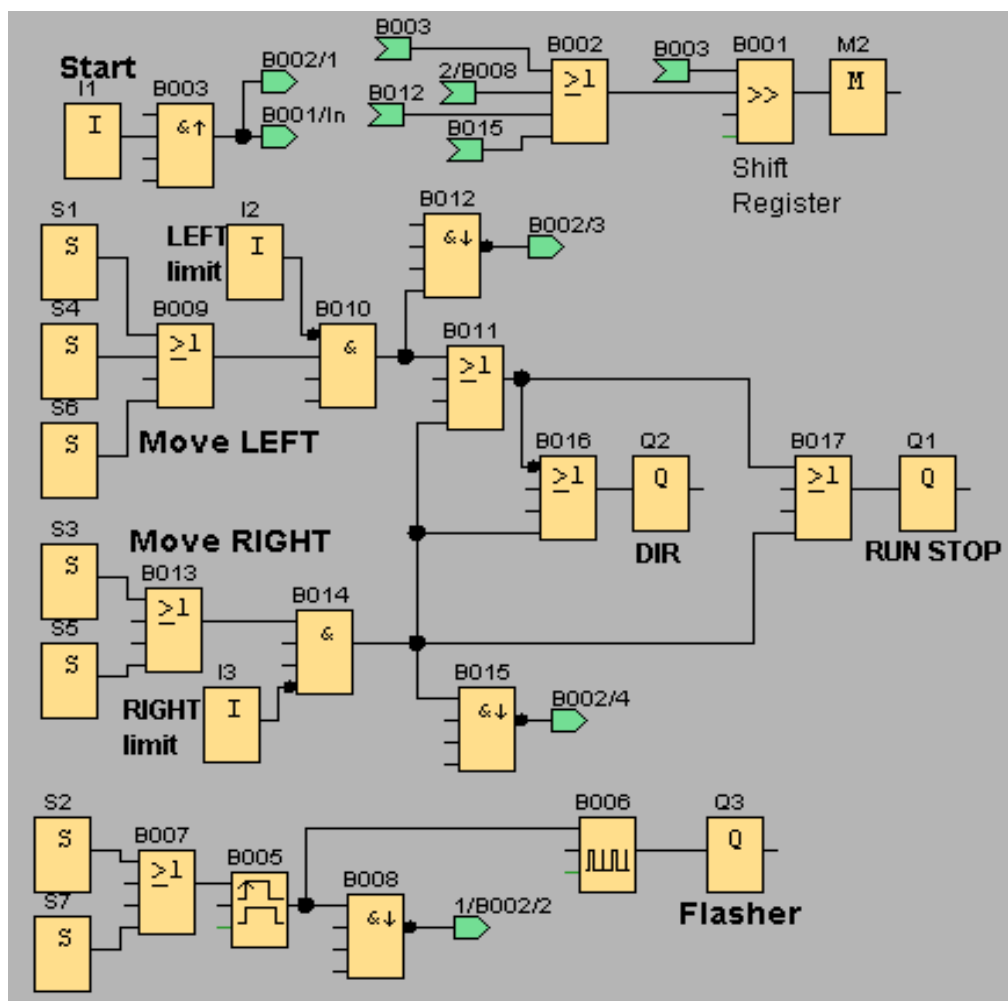


Fig. 8.11 Circuit for Problem #8

Recall that in a [FBD] two or more block *outputs* cannot be connected to a single input (if one went *high* and the other *low*, what would be the result?) This is the reason for the many OR gates in the circuit shown in Fig. 8.11. The **M2 flag** at the *output* of the **Shift Register** is not needed for *simulation*, but the **LOGO!** unit does not like to accept a *downloaded program* in which any function block is not connected to an *output, flag* or another function *block input*. The positive edge-triggered AND, **&↑**, is placed after **I1**, just in case the operator happens to hold this input *high*.

Task 8.3: Linear Motion

A: Construct and simulate an [FBD] solution for Problem #6. If available download your program to the LOGO! unit and run it with the *linear motion* module connected. Recall **Section 3.5 Running in Hardware**

B: View your solution as a [LAD] and compare the placement of all inputs, outputs and function blocks.

C: Construct and simulate an [FBD] solution for Problem #7. If available download your program to the LOGO! unit and run it with the *linear motion* module connected

D: Modify Problem #7 so that the program will stop itself after the motion has reversed itself for 12 times. Simulate your solution and also run it in hardware if available.

E: Construct and simulate an [FBD] solution for Problem #8. If possible, run it in hardware. Examine carefully the Parameter values of all the Special Functions

F: Modify Problem #8 so that the moving block makes only one round trip, the flashing is for 10.0 seconds and has a pattern as **● —**. Construct and simulate your solution and also run it in hardware, if available.

8.4 Stepper motors

Depending on their load *conventional electric motors* usually do not attain full speed until a short time after power is switched on, and do not stop turning the moment power is switched off. However **stepper motors** have a number of parallel inputs and the motor advances a fraction of a full turn each time power is briefly applied to one or other input. The motor on the *Linear motion module* pictured in Fig. 8.8 is actually a *stepper motor*, which can act as a *conventional motor* (a source on the module applies a rapid series of pulses whenever the **Run Stop** terminals are shorted). In Section 8.3 the module was connected as a *conventional motor*, but in this section we control it as a *stepper motor*, by briefly shorting to terminal **C** any of the four terminals **1, 2, 3, 4**.

A simple test program is shown in Fig. 8.12. The **Edge-triggered wiping relays** are configured to give a 0.04 second high output. For the simulation mode, inputs **I1** to **I4** should be configured as *Momentary pushbutton (make)*. Of course in *Simulation* mode you cannot see any motion of the motor shaft.

To use the *stepper motor* on the *Linear motion module*, first transfer or *download* the program to the LOGO! unit. Select from the Standard Toolbar Tools \Rightarrow *Transfer* \Rightarrow *PC \rightarrow LOGO!* or the corresponding icon. Connect one terminal of **Q1**, ..., **Q4** to the corresponding numbered input on the *Linear motion module*, and the other terminal to the **C (Common)** input on the *Linear motion module*.

In *simulation* mode we click the input icons with the mouse. In *hardware* mode we press in input keys on the LOGO! Module board. When a program has been successfully downloaded to the LOGO! unit, the computer may be disconnected, and the LOGO! module stands by itself. In practical applications the LOGO! unit does not remain connected to the computer; the computer is used only to create, de-bug and download the program.

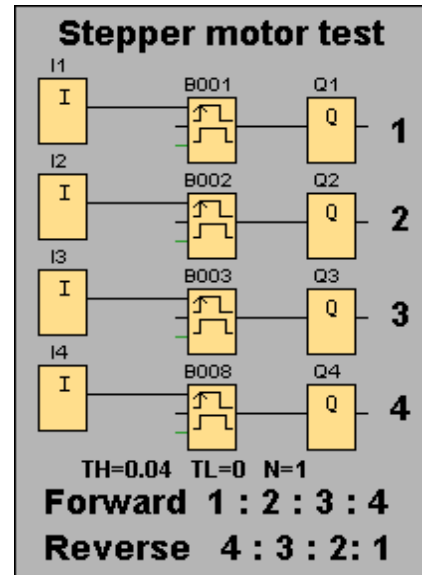


Fig. 8.12 Stepper motor test

As mentioned in Section 3.5 it is possible to monitor the hardware operation through the computer. To do this, on the Standard Toolbar click Tools \Rightarrow Online Test or the associated icon. In this mode the state (*high* or *low*) of all inputs, outputs and memory flags used in the program are displayed on the computer. You may also display the variable values of selected function blocks (by clicking on them), to display their actual hardware values.

Task 8.4: Testing the stepper motor

A: Create the [FBD] shown in Fig. 8.12, and simulate it in software.

B: If available, connect the *Linear motion module* to the *LOGO! module* and download the program into the LOGO! unit. Verify that the direction of motion is in accord with the key sequence indicated in Fig. 8.12. Describe what happens if the input keys are pressed in random sequence.

C: Determine how many steps are needed for the stepper motor shaft to make one full rotation (360°). From this data, calculate the *number of degrees of rotation for each step*

While the circuit in Fig. 8.12 allows us to test the basic operation of a stepper

motor, it is neither convenient nor practical. We need a control circuit with at least three inputs, for **Run**, **Single step** and **Direction**. One approach to maintain the correct sequence of pulses for forward or reverse motion is the use of a *Shift Register*. In Section 7.4 we developed a circuit that would cyclically shift a single high bit *in one direction*, but here we need a cyclic shift in either direction.

The **Shift Register** is basically **linear**, that is, a bit shifted out at either end is lost. We need to make the action **cyclic**, that is, a bit shifted out at either end comes back in at the opposite end. Although the *Shift Register* controls 8 memory locations and the stepper motor has only four input coils, it is possible with OR gates for both **S1** and **S5** to control coil **1**, both **S2** and **S6** to control coil **2**, and so on. A complete control circuit is shown in Fig. 8.13.

Fig. 8.13 A stepper motor control circuit

The right portion of Fig. 8.13 shows how pairs of **S** locations trigger the **Edge-triggered wiping relays** to produce the required 0.04-second coil pulses. The **Direction** input, **I2**, controls not only the **Dir** input of the *Shift Register* but also the pair of AND gates, **B8** and **B9**, both of which cannot be open at the same time. The other input to these AND gates is from either **S1** or **S8**. The **In** input of the Shift Register may come from the initializing **M8 flag** (as explained in Section 7.4) or from **S1** or **S8** depending on **Direction**. The *Shift Register Trigger* input comes from the initializing high of **M8**, or from the **Step** input, **I1**, or from a series of pulses from the **Asynchronous pulse generator**, **B18**, enabled by the **I3 Run** input.



To view the corresponding 25-rung [LAD] click the toolbar icon.

Task 8.5 Running the Stepper Motor

A: Create the [FBD] shown in Fig. 8.13, and simulate it in software.

B: If available, connect the *Linear motion module* to the *LOGO! module* and download the program into the LOGO! unit and verify correct operation.

C: Convert the [FBD] into a [LAD] and compare inputs, outputs and function blocks. How many rungs are required by the [LAD]?

D: Add **left-** and **right-limit** switches the your control program.

E: Modify the **Run** input, **I3**, so that the moving block advances exactly 1.00 cm each time **Run** is pressed. *Hint:* From the number of steps per 360° and the pitch of the drive screw, calculate the number of pulses needed. Use an **Up/Down Counter** (see Section 5.3) to count this number of pulses. Also include a latching circuit so that the movement completes the 1.00 cm even if **I3** is released.

Highlights

Simulation in software is a great convenience to the programmer, but actual testing on the LOGO! unit with its hardware connections is the final test of every application program.

Generally there are a number of different approaches to solve a PLC problem. Start your solution with the approach that seems most natural to you, and select [FBD] or [LAD] according to your own background and taste.

The **Shift Register** Special Function at first may seem not too interesting, but the experienced programmer can put it to many uses.

Looking Backwards

1: Discuss the similarities and differences between the *Multiplex–DeMultiplex* circuits of Sec. 3.6 and the *Corridor Lights* circuits of Section 8.1

2: *Three-input X–OR* circuits are not in common use. Design a 3-input circuit that has a *high* output provided one and only one input is *high*.

3: Discuss the similarities and differences between the *Pulse Relay Special Function* and the *Push On – Push Off* circuit of Section 6.2

4: If both **R** and **S** inputs of the **Latching Relay** go *high* at the same time, the output is *high*. The **Pulse Relay** offers all that the **Latching Relay** offers and more besides. The **Pulse Relay** has the same dual **R** and **S** inputs as the **Latching Relay**, a **Trigger** input similar to the *Push On – Push Off* circuit, and also, by parameter selection, can determine the output if **R** and **S** are high at the same time. Set up the circuit shown in Fig. 8.14 and in *simulation* mode verify all these properties of the **Pulse Relay**.

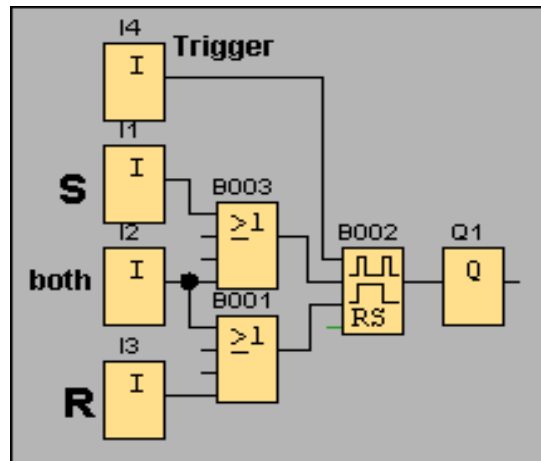


Fig. 8.14 Pulse Relay Test

5: Design and de-bug a circuit that on command will output the international distress signal, SOS, (• • • — — — • • •)

6: In the light of the PLC *Scanning Cycle*, what is the theoretical upper limit to the speed (in revolutions per second) of the module Stepper Motor when driven by the PLC?

7: The stepper motor circuit shown in Fig. 8.13 uses the **S flags** in *pairs*. Would it be practical to use only the *first four flags*, and have the *Shift Register* circle only through the first four bits? What problems might you encounter?

8: Explain the difference, if any, between the 8 **S flags** and the 24 **M flags**.

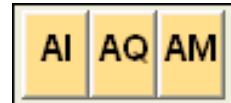
LOGO! PLC

Chapter 9: Analog

The Siemens PLC LOGO! series offers 8 different models all with 8 digital inputs and 4 relay outputs. Four of these models have a small 6-key keypad on the module face. Also four models (12/24 RC, 12/24RCo, 24, 24o) permit two of the eight inputs (**I7** and **I8**) to be used for either *digital* or *analog input*. None of the basic modules have an *analog output*. Various *Expansion Modules* are also available, providing up to 24 digital inputs, 16 digital outputs, 8 analog inputs and 2 analog outputs. Details may be viewed by selecting from the Standard Toolbar **Tools** \Rightarrow **Determine LOGO!** or **Select Hardware**. In *Simulation* mode all these inputs / outputs are available, since *Simulation* mode can work without any hardware present

9.1 Analog Ins and Outs

Analog inputs accept a voltage between 0.00 and 10.00 volts (higher voltage truncated to 10.00 volts) which is converted by an internal *Analog-Digital Converter (ADC)* to an *integer value* between 0 and 1000, providing an input resolution of 10 millivolts. *Analog outputs* use a *Digital-Analog-Converter (DAC)* to provide a similar voltage between zero and ten volts (recall that *digital* outputs are relays.). Just as LOGO!Soft provides 24 **M flags**, each holding a single-bit value of 0 or 1, so also there are **6 AM analog flags**, holding integer values (–32768 to +32,767, the range of a 16-bit signed integer). Shown here are the icons, **AI**, **AQ**, **AM**, from the **Constants** display used to select these blocks.



We start with a comparison of the simplest possible analog and digital circuits, shown, in Simulation mode, in Fig. 9.1. Notice the difference in the thickness of the wire connecting digital and analog blocks. The state (*high* or *low*) of a digital block is shown by its color, while an analog value is displayed in numerical form below the block. The simulation status bar contains both input, flag and output icons, but the analog icons are more elaborate. You can change the **AI1**

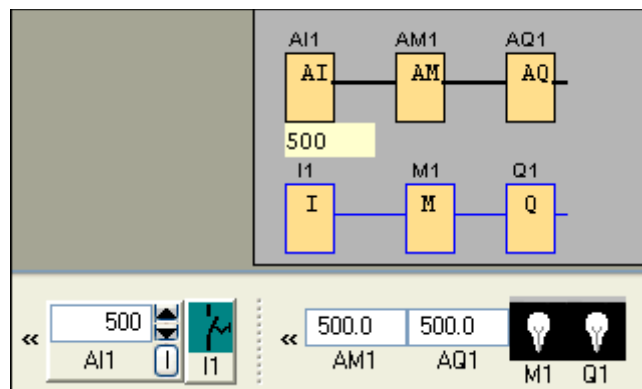


Fig. 9.1 Two simple circuits in Simulation mode

input value by clicking the **up** or **down arrowheads**, by clicking and dragging the **[I]** slider, or directly typing a value in the *display area*. You can flip the state of a *digital* input icon by *left-clicking* its icon; likewise you can change in value of an *analog* input icon by *left-clicking* the icon and dragging the slider that appears.

Why does an *analog input*, **AI**, multiply by a factor of 100 the voltage at the hardware input terminal? The typical sensor voltages applied to analog inputs amount to

only a few volts. By multiplying the input voltage by 100, the internal integer steps are 1/100 of a volt. Of course at the *analog output*. **AQ**, the internal integral representation has to be divided by 100 to obtain a value in volts. The analog output voltage is always within the range from zero to ten volts. In *Simulation* mode the numerical value displayed below the **AI** icon is the *internal integer representation*, that is, the applied voltage multiplied by 100.

Task 9.1 Analog Simulation

A: Set up the circuit of Fig. 9.1 and run in *Simulation* mode. If your LOGO! unit is one of the models mentioned above with *Analog input*, replace **AQ1** by **AM1** (unless you have an Analog expansion module) and download your program. Connect the variable voltage source as input to **I7** (alias as **AI1**). Monitor the LOGO! operation with *Online Test*.

9.2 Sensors, Digital or Analog

We have already used digital sensors; the float switches of Section 5.2 or the limit switches of the *Linear motion module* shown in Fig. 8.8. The *digital* sensor is basically an *open / closed switch*, with state determined by the condition that is sensed, such as position, pressure, temperature, humidity, etc.. The *analog* sensor produces a voltage (or current) that is *proportional* to the condition that is sensed. For example, in a large air-conditioning system a *flow switch* is a *digital sensor*, that is *closed* if the cooling water *rate of flow* is above some minimum. A *flow meter* in the same system is an *analog* sensor, with voltage output proportional to the actual *rate of flow* of the cooling water. A *digital* sensor is usually directly connected to the PLC digital input, while the signal from an *analog* sensor often needs some *pre-conditioning* before connection.

The advantage of *pre-conditioning* the input signal may be seen in the case of a thermocouple with linear voltage output from 0.077V to 0.193V as temperature varies from 0.0° to 100.0° C. If directly connected, the input **ADC** converts this to an integer range from 8 to 19, providing a sensitivity of about *12° per step*. However if the signal is first passed through a linear pre-amplifier with fixed gain of 40.0, the **ADC** internal integer range is from 308 to 772 providing a sensitivity of about *1/4° per step*.

Pre-conditioning is done outside the PLC. Also available is a form of *post-conditioning* within the PLC operation which works on the internal integer values. Why might this be needed? With additional hardware the LOGO! unit can send text messages and decimal values to a remote display so the whole control process may be monitored. Since most of the analog Special Functions we will consider later make provision for such a transformation, it might be good to take a closer look on how it works.

9.3 The Analog Amplifier Special Function

The **Analog Amplifier Special Function** is not an external voltage amplifier but rather an internal function that performs a linear transformation on a integer value. The formal description is shown in Fig. 9.2.

Connection	Description
Input Ax	Input the analog signal to be amplified at input Ax. Use the analog inputs AI1...AI8, the analog flags AM1...AM6, the block number of a function with analog output, or the analog outputs AQ1 and AQ2. AI1...AI8: 0 - 10 V corresponds with 0 - 1000 (internal value).
Parameter	A : Gain Range of values: ± 10.00 B : Zero offset Range of values: ± 10000 p : Number of decimals Range of values: 0, 1, 2, 3
Output AQ	Analog output AQ: -32768...+32767

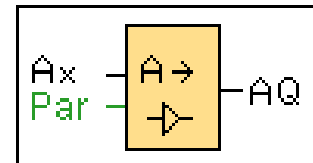


Fig. 9.2 Analog Amplifier Special Function

Notice that both the input **Ax** and output **AQ** are integer values, not a *high* or *low* bit as for the functions we have already considered. Fig. 9.3 presents in *Simulation* a test circuit for this function. The gain and offset parameters are chosen so that the output of the *Analog Amplifier* equals the value of the temperature that is sensed. Recall the pre-conditioned thermocouple example in which 0° corresponded to 3.08 volts, (digital equivalent of **308**) and 100° corresponded to 7.72 volts (digital equivalent of **772**). The

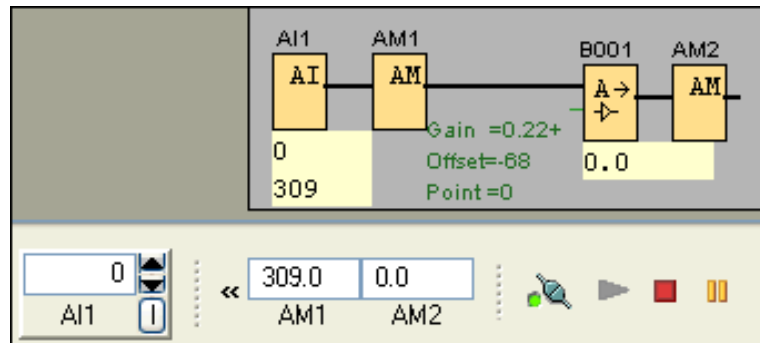


Fig. 9.3 Analog Amplifier test circuit

AM1 flag placed after **AI1** gives the digital equivalent of the input voltage, which is also the input to the *Analog Amplifier*. **AM2** flag shows the output of the *Analog Amplifier*. The linear transformation is given as:

$$\text{Input} \times \text{Gain} + \text{Offset} = \text{Output}$$

The *Analog Amplifier* parameters have been set as **Gain = 0.22** and **Offset = - 68** so with an input voltage of 3.08 volts, and a post **ADC** integer value of 308 we obtain zero for the *Analog Amplifier* output, which is the actual temperature that is sensed:

$$308 \times 0.22 - 68 = - 0.24 \approx 0$$

Ideal values are *Gain*=0.216 and *Offset*=-66.38 , but the software permits only *two decimal places* for **gain** and *integer values* for **offset**.

The box below the **AI1** icon displays two numbers ; the *lower* number, 309, is the digital output of the **ADC**, and the upper number, 0, is the result of the *Analog Amplifier*

transformation. When you simulate *input*, using the *up* or *down arrowheads* or the *slider* on the *Simulation* bar, you step through the *integer values* transformed by the *Analog Amplifier* (contents of the **AM2 flag**) and not the **ADC output** (contents of the **AM1 flag**.)

Why is there no **AQ1** analog output icon appearing in the test circuit of Fig. 9.3? We can simulate the circuit in software with or without the **AQ1**, but if our hardware configuration does not include an **Analog expansion module** (which is somewhat expensive) our LOGO! cannot run the program. Incidentally the **M2 flag** is optional in software simulation but is required for hardware operation in place of **AQ1**, for *the output of every Special Function must go to some designated memory location..*

Task 9.2 The Analog Amplifier

A: Set up the circuit of Fig. 9.3 and run in *Simulation* mode. Try to increase the simulated input signal one step at a time. Notice how “one step at a time” is in terms of the output of the Analog Amplifier (**AM2**) rather than its input (**AM1**)

B: What is the input digital value that makes **AM2 = 100.0** ? Explain any slight errors that may appear.

C: Transform the circuit from [FBD] to [LAD] . Notice the new icons used for an **analog contact**, $\neg | \neg^\circ$ and an **analog coil**, $\neg ()^\circ$

D: If your LOGO! unit is one of the models with Analog Input mentioned above, download your program. Recall that **I7** is an alias for **AI1**. Connect the variable voltage source as input to monitor the LOGO! operation with *Online Test*.

E: In the [FBD] remove just one connecting wire, Can you still view this incomplete circuit in Simulation mode? Can you download it to hardware?.

9.4 Analog Limits

Problem #1 *The incubator temperature in a chicken hatchery is to be maintained by heaters between 39° and 42°, while in temperature in the manager's office is to be cooled by an air-conditioner within the range of 19° to 21°. Assume the sensor is preconditioned so that at 0° the voltage is 0.00V and at 100° the it is 10.00V.*

There is something similar between this problem and the case of maintaining the water level in a tank between certain limits, as we saw in Section 5.2. At the *lower limit* the pump was turned **on**, to raise the water level; at the *upper limit* the pump was turned **off** to keep the level from rising further. For levels *in between* the pump could be **on** or **off** depending on whether we are pumping more liquid in or just allowing it to slowly drain out..

In Section 5.3 we met the **Up / Down counter** function. Counting **Up** raised the counter *tally*, counting **Down** decreased the *tally*, somewhat similar to the rising and falling liquid level in the tank. The function also has **On** and **Off** settings; For **On>Off** if

tally is above the **On** setting the output goes *high*, for *tally below* the **Off** setting the output goes *low*. For a *tally in-between* the output is either *high* or *low*, depending on the last change.

In problem #1, the temperature goes up and down, almost like the counter tally. Fortunately LOGO!Soft Comfort® gives us an *analog Special Function*,, **Analog threshold trigger**, similar to the **Up /Down counter**. Fig. 9.4 presents details.

Connection	Description
Input Ax	Input the analog signal to be evaluated at input Ax. Use the analog inputs AI1...AI8, the analog flags AM1...AM6, the block number of a function with analog output, or the analog outputs AQ1 and AQ2. 0 - 10 V is proportional to 0 - 1000 (internal value).
Parameter	A: Gain Range of values: +- 10,00 B: Zero offset Range of values: +- 10,000 On: On threshold Range of values: +- 20,000 Off: Off threshold Range of values: +- 20,000 p: Number of decimals Range of values: 0, 1, 2, 3
Output Q	Q is set or reset depending on the set thresholds. If On >= Off , then: Q = 1, if digital value Ax > On Q = 0, if digital value Ax <= Off If On < Off , then Q = 1, if On <= digital value Ax < Off

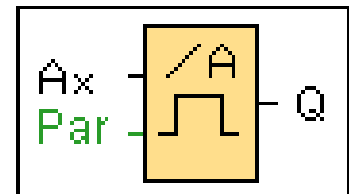


Fig. 9.4 **Analog threshold trigger** Special function

Notice that the output of this function is **Q**, a *binary* value, **0** or **1**, not **AQ**, a *analog* value. The Parameters are entered in the usual manner; left-click on the icon and select **Block Properties...** . It seems like this function is made to order for the *manager's* air-con. Set **Gain=1**, **Offset=0**, **On=210**, **Off=190** and ignore **p**. The function output controls the air-con motor. If office temperature is *above* 21, turn *on* the air-con, and let it run until the temperature drops *below* 19 .

But things are a little different for the incubator heater. Unlike the air-con motor, the heater should go **On** at the *lower* temperature and **Off** at the higher temperature.. But if **On** < **Off** the function is *high* only in the range between **Off** and **On**, and *low* for temperatures above or below this range. (This would cook the eggs, not hatch them!) The solution is simple; just add a NOT gate between the analog function block and the **Q2** output. The circuit is shown, in Simulation mode, in Fig. 9.5.

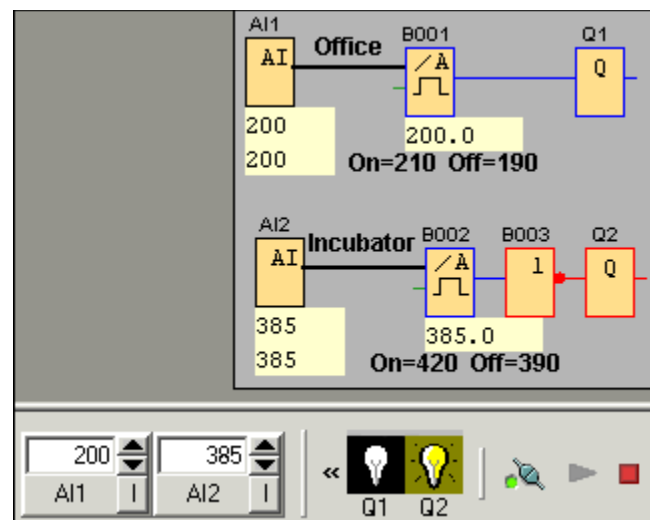


Fig 9.5 Circuit for Problem #1

Task 9.3 The Egg Farm

A: Set up the circuit of Fig. 9.5 (or design your own version) and operate it in Simulation mode. Verify that it really does solve Problem #1..

B: The circuit in Fig. 9.5 presumes that at 30°, the voltage applied to **AI1** is 3.00 volts. Change the **Gain** and **Offset** settings of the *Analog threshold trigger* and the **On / Off** levels so that the system works with the *thermocouple* and *pre-processor* discussed in Section 9.3. Note that the **Gain / Offset** setting here do the work of the *Analog Amplifier* discussed there.

C: Convert the circuit of Fig. 9.5 from [FBD] to [LAD]. A NOT gate appears In the [FBD] . What is its counterpart in the [LAD]

9.5 More of the same

As already mentioned, the diagram shown in Fig. 5.5 applies to the *tally value* of the **Up – Down counter** as well as to the *integer value* of the **Analog threshold trigger**.

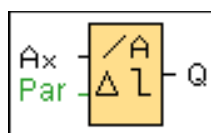
If **On** \geq **Off**, then

Q=1 if the monitored value is *greater than On*, **Q=0** if the monitored value is *less than Off*, and in the *in-between region* there is no change in **Q**; it could be *high* or *low* depending on its last transition.

If **On** < **Off**, then

Q=1 if the monitored value is in the *in-between region*, and **Q=0** is *above* or *below* it

In each case the programmer sets the **On** and **Off** parameters. For some reason *LOGO!Soft Comfort*® offers us another analog function, **Analog Differential Trigger**, that does almost the same thing as the **Analog Threshold Trigger**, with the exception in the way the **Off** parameter is defined:



**Analog
Differential
Trigger**

In place of directly defining the **Off** parameter the programmer defines the quantity **Delta**, which may be positive or negative. Then it defines **Off = On + Delta**. If **Delta** is *negative* then **On** > **Off**; if **Delta** is *positive* then **On** < **Off**. So the end result is the same for both these analog functions. For a full description of the **Analog Differential Trigger** (and all other functions as well), left-click on *Help* \Rightarrow *Contents*

Task 9.4 Analog Differential Trigger Special Function

A: Create a circuit to solve Problem #1, but use the **Analog Differential Trigger** Special Function rather than **Analog Threshold Trigger**. Run this in Simulation mode and verify its behavior

9.6 Monitoring a value

A common control task is to monitor a particular physical value, say, temperature of a furnace or air pressure of a compressor, and provide a warning signal if the value moves outside a pre-assigned range of values. For example:

Problem #2 *It is desired to maintain the system pressure in the range of 65.0 to 70.0 psi, absolute. The pressure gauge response from 15 to 90 psi is from 0.00 to 10.00 volts. If the pressure is outside the desired range, a warning signal should be given.*

Here the input is *analog*, the output *boolean*, that is, *high* or *low*. An **Analog Threshold Trigger** seems to be the first choice. If **On** < **Off** then the output is *high* when the digital value is *within the range*. Since we need a high signal when *outside the range*, simply add a NOT gate after the function. To give numerical values to **On** and **Off** we must find the relation between the external pressure, **P**, and the internal PLC integer expression, **I**.

We are told that the sensor is *linear*, *voltage* directly proportional to *temperature*. We are given two pairs of values, and from these we can draw a straight-line graph, relating **ADC** value to *temperature*. From this relationship we find the corresponding **On** and **Off** values. Notice that we need not change the default **Gain** and **Offset** values. Fig. 9.7 shows an appropriate [FBD].

The slope of the linear response graph shown in Fig. 9.6 is 13.33, approximately 13 integer steps per degree, giving a response sensitivity of 1/13 of a degree.

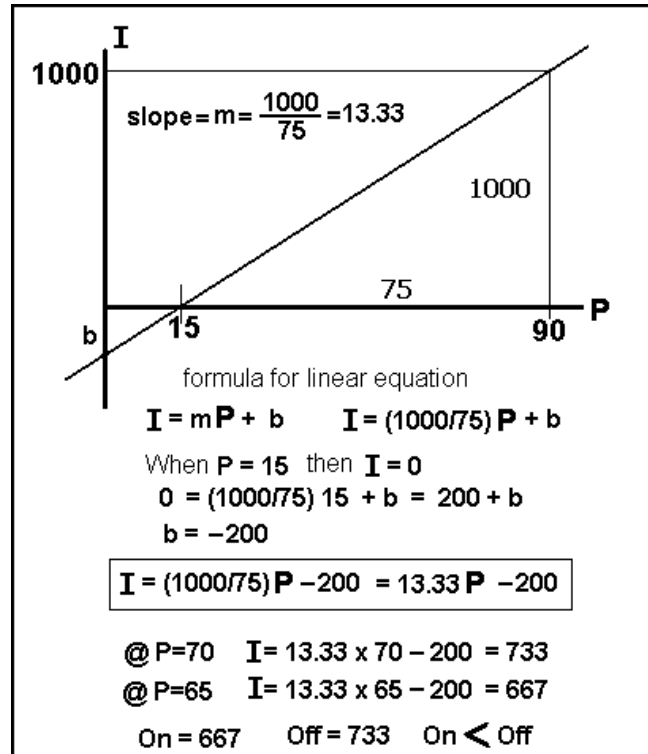


Fig. 9.6 Configuring for this sensor

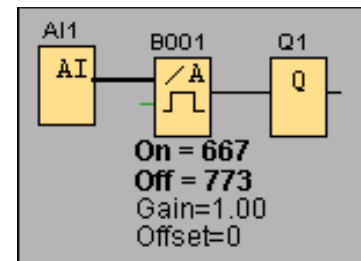


Fig. 9.7 For Problem #2

Task 9.5 Monitoring Temperature

A: Set up the circuit shown in Fig. 9.7 and in software Simulation verify that is a valid solution for Problem #2

B: Would it be possible to use an **Analog Differential Trigger** ? If so, what would be the values of **On** and **Delta**?

C: Suppose the sensor of Problem #2 has been replaced by a different model, also linear, with the characteristics : 0.00 volts at -10.0° and 10.00 volts at 120° . Find the new **On** and **Off** settings, and in *Simulation* verify your values.. With this new sensor, what is the system *response sensitivity*

9.7 Analog watchdog

In the previous section we considered monitoring a value, and warning if this value moved outside a pre-determined range. In Problem #2 the range is 5° ($=70^{\circ}$ – 65°) with the corresponding integer range of 66 ($=733$ – 667) . By setting definite **On** and **Off** values we not only determine the extent or *width* of the range, but also where this range is located. LOGO!Soft Comfort[®] also provides an interesting *analog monitoring function* that looks at the *width* of the allowed range, **|Off–On|** but is not concerned with the *actual value of either On or Off*. This special function is **Analog value monitoring**. In the *Help* file it goes by this name, but if you click its icon or *Block properties* it is called **Analog watchdog** (Note: it is quite different in purpose and operation from the *Watchdog Timer* common to **PIC** chips.) The formal definition is shown in Fig. 9.8.

Connection	Description
Input En	A positive edge (0 to 1 transition) at input En saves the analog value at input Ax ("Aen") to memory and starts monitoring of the analog range Aen +/- Delta.
Input Ax	You apply the analog signal to be monitored at input Ax. Use the analog inputs AI1...AI8, the analog flags AM1...AM6, the block number of a function with analog output, or the analog outputs AQ1 and AQ2. 0 - 10 V is proportional to 0 - 1000 (internal value).
Parameter	A: Gain Range of values: +/- 10.00 B: Zero offset Range of values: +/- 10,000 Delta: Difference value for the Aen on/off threshold Range of values: +/- 20,000 p: Number of decimals Range of values: 0, 1, 2, 3
Output Q	Q =1 if Ax out-of-range and En =1

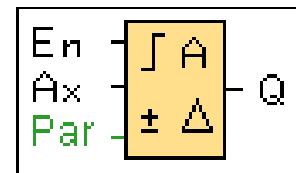


Fig. 9.8 Analog Value Monitoring (Watchdog)

If we know the range, but not where the range is located, how can the function work? The *midpoint* of the range is set as the integer value of **Ax** the moment **Enable** goes *high*. While **Enable** is *low*, the function does not operate. Like a true watchdog that barks only when things get out-of-hand, the output goes *high* only when the current **Ax** value differs by $\pm\Delta$ from the initial **Ax** value, that is, goes *outside* the range. (The **Q** of the circuit in Fig. 9.7 remains *high* as long as the input is *within* the range. A NOT gate may be added if desired.) Fig. 9.9 shows the circuit in *Simulation* mode. Notice the two values shown below the **AI** input block are the same since **Gain=1**, **Offset=0**. The upper of the two numbers shown beneath the Watchdog function block is the *current Ax*, the lower number is the *initial Ax* the moment **Enable** went *high*. If **Enable** is *low*, the lower number is zero.

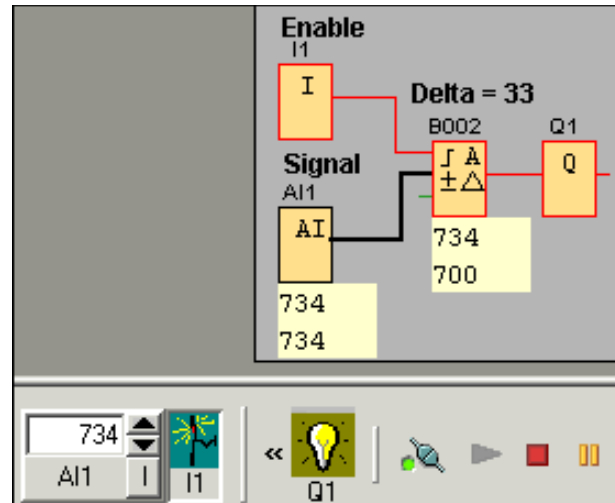


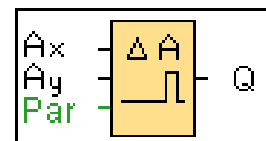
Fig. 9.9 The Watchdog in action

Task 9.5 The Analog Watchdog

- A:** Set up the circuit shown in Fig. 9.9 with **Delta=33** ($\frac{1}{2}$ the range in Problem #2) and in software *Simulation* verify its supposed behavior.
- B:** Experiment with various values for **Delta**. Is it possible to have **Delta=0**?
- C:** If suitable hardware is available, download the program to a LOGO! unit, and connect a potentiometer to **AI1** (alias of **I7** or **I8**) and operate in *Online Test* mode.

9.8 Analog Comparator

The analog functions considered so far have all had a single analog input, **Ax**. The **Analog Comparator** which we now examine has *two* analog inputs, **Ax** and **Ay**, and a single digital output. The function behaves exactly the same as the **Analog Threshold Trigger** provided we replace the **Ax** term there with the signed difference (**Ax-Ay**). The **Analog Comparator** icon is shown here and the official description may be found by clicking its icon with the ☐? context-sensitive help cursor, or Fig. 9.4. with (**Ax-Ay**) in place of **Ax** alone.



Problem #3 A pressure reducer in a compressed-air line is to reduce the pressure by 20.0 ± 1.0 pounds per square inch (psi). The linear pressure sensors before and after the reducer have the following characteristics: 0.00 volts at 0.0 psi and 10.00 volts at 100.0 psi. A warning is required if the difference is outside the 19 to 21 psi range.

The obvious function to use here is the **Analog Comparator**. From the character of the sensors we know that a pressure change of 1.0 psi corresponds to an integer change of 10 within the PLC. Since we wish to monitor *within* a range we select **On < Off** with **On=190** and **Off=211**. Since the warning is for differences outside the allowed range, the NOT gate follows the Comparator icon. A suitable circuit is shown in Fig. 9.10

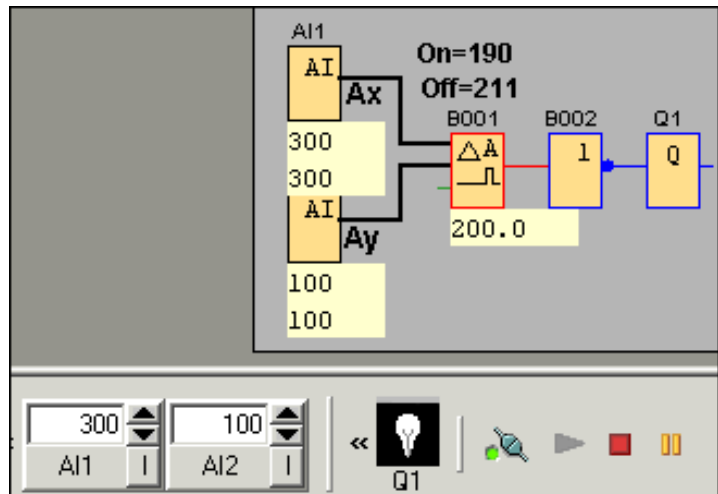


Fig. 9.10 Circuit for Problem #3

Task 9.6 Analog Comparator

A: Set up the circuit shown in Fig. 9.10. In Simulation mode. Notice how the warning signal goes high, depending on the *difference* between the two inputs.

B: If equipment is available, download the program to the LOGO! module. Follow the instructions provided with your hardware on how to connect the analog voltage to inputs **I7** and **I8**. (See the Appendix for instructions on the hardware provided with this manual.) . Check the input voltages with a digital voltmeter and compare these with the values on the screen while running in *Online Test* mode

9.9 Analog multiplexer

Problem #4 To cure an epoxy sample it must be maintained at 60° for 10 minutes, then at 70°, 80° and 90° each for 10 minutes, then allowed to cool down for an additional 15 minutes, after which a flashing signal occurs for five seconds, indicating the process is finished. The characteristics of the temperature sensor are 0.00V @ 0°C, 10.00V @ 100°C.

This problem involves using an **Analog Comparator** to maintain a temperature at four different levels (then turning off the heater and warning). One approach is to use five separate comparators; an alternate approach is to use an **Analog Multiplexer**. Back in Section 3.6 we considered **digital** multiplexers; here the idea is somewhat the same. Up to four analog values, **V1**, ..., **V4**, are stored as parameters within the function, and one at a time may be selected as the function output. Unless **Enable** is high, analog output, **AQ**, is zero. Digital inputs **S₁** and **S₂** select which of the four values appear at **AQ**. Note that the **S₁** and **S₂** symbols here are quite different from the dedicated **S1** and **S2** memory locations controlled by the **Shift Register**. The formal definition is presented in Fig. 9.11

Connection	Description																		
Input En	1 on Enable AQ =selected output 0 on Enable AQ =0																		
Inputs S1 and S2	S1, S2 output selectors <table><tr><th>S1</th><th>S2</th><th>V</th></tr><tr><td>0</td><td>0</td><td>V1</td></tr><tr><td>0</td><td>1</td><td>V2</td></tr><tr><td>1</td><td>0</td><td>V3</td></tr><tr><td>1</td><td>1</td><td>V4</td></tr></table>				S1	S2	V	0	0	V1	0	1	V2	1	0	V3	1	1	V4
S1	S2	V																	
0	0	V1																	
0	1	V2																	
1	0	V3																	
1	1	V4																	
Parameter	V1...V4 : Analog values -32768...+32767 p : Number of decimal places:0,1,2,3																		
Output AQ	V1...V4 or zero																		

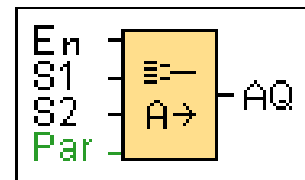


Fig. 9.11 The **Analog Multiplexer** Special Function

Notice that this is the first function we have considered whose output is not *digital*, but *analog*. So, instead of four **Analog Comparators**, we may use a *multiplexer* and a single *comparator*. Since there are six sequential operations (4 *heatings*, a *cooling* and a *warning*) each of known time duration we can use **Edge-triggered wiping relays** with selected T_L and T_H values.

In the solution circuit shown in Fig. 9.12 notice that the first heating operation does not have its own timer; it operates until the second operation begins. The first four operations simply adjust the S_1 / S_2 values of the **Analog Multiplexer** for the desired temperature. The last two operations force *low* the heater output **Q1**, to allow the sample to cool. The T_L values for the five timers are 10, 20, 30, 40, 55 minutes respectively. The corresponding T_H values are 10, 10, 10, 10 minutes and 15 seconds.

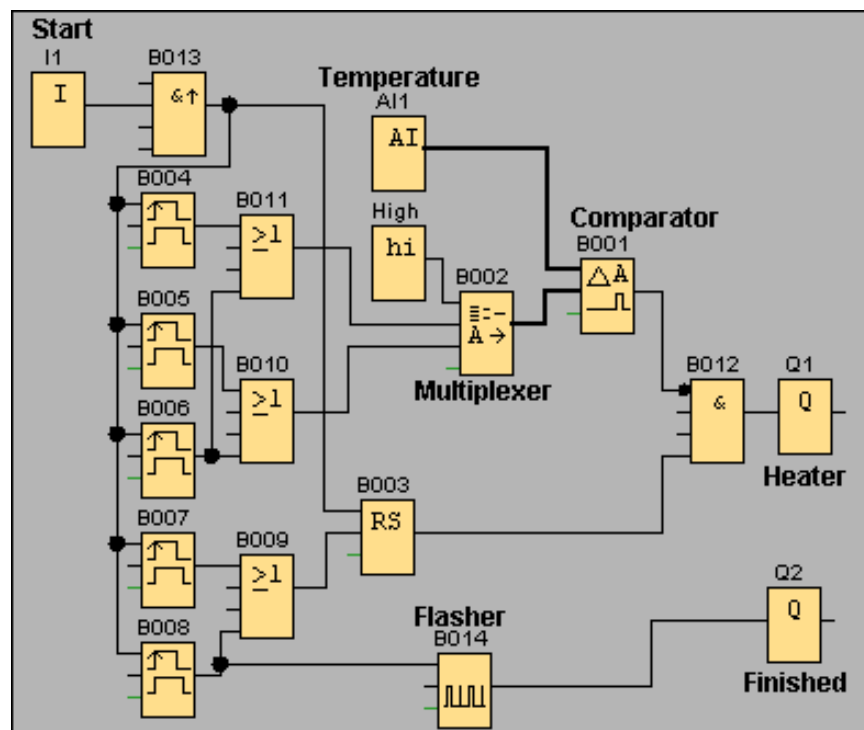


Fig. 9.12 Solution circuit for Problem #4

Because of the first inverted input of the AND gate, B12, the heater is on during the first four operations as long as $Ax < Ay$.

Task 9.7 Simulating Problem #4

A: Set up the circuit shown in Fig. 9.12. As it stands, 55 minutes is needed to simulate one full cycle. Therefore change the units of T_H and T_L from *minutes* to *seconds*. At any time during the simulation you can click the **pause icon** (the yellow parallel bars) and adjust the temperature analog input, **AI1**, and then resume the simulation. Verify that the circuit really behaves as specified.

B: Modify the circuit for the case where the 10-minute heating at 90° is omitted.

C: When working with integrated circuits, the more chips we use, the greater the production cost. With the PLC additional gates and function blocks have no effect on cost. Therefore solve Problem #4 without using the **Analog Multiplexer**. Verify your solution in software.

Highlights

Each **analog input, AI**, is an **ADC** (*Analog to Digital Converter*), with an *analog* input range from 0.00 to 10.00 volts and an *integer* output range from 0 to 1000. Each *analog output, AQ*, is a **DAC** (*Digital to Analog Converter*) with input *integer* range from 0 to 1000, and *analog* output range from 0.00 to 10.00 volts.

For convenience, *within the PLC* digital, values may be transformed to a range from -32768 to 32767 , the range of a 16-bit signed integer. Unless special display hardware is included in the system, it is usually not necessary to change the Gain / Offset default values of the various analog special functions. However external signal pre-conditioning is often required.

The **Analog threshold trigger** and the **Analog comparator** are the most used functions.

Only certain basic LOGO! modules provide two analog inputs (sharing the digital inputs **I7** and **I8**.) A special analog expansion module is required for each analog output, **AQ**

Looking Backwards

- 1: What is the maximum temperature that can be measured by the thermocouple circuit described in Section 9.2 . Explain the reason for your answer.
- 2: Set up a circuit similar to that of Fig. 9.2 but insert an additional **Analog Amplifier** just before **M2**. Set the *gains* at 5.00 and 0.20 respectively and the *offsets* of both as 0. Simulate the circuit and compare the circuit input and output digital values (**M1** and **M2**). What is the largest possible input *voltage* for this circuit?
- 3: A linear flow meter provides zero output voltage for any flow less than $0.20 \text{ m}^3/\text{sec}$, and a 10.00 volt output at a flow rate of $8.00 \text{ m}^3/\text{sec}$. Design and simulate a circuit that warns the operator if the flow rate departs from the optimum of $3.50 \text{ m}^3/\text{sec}$ by more than $\pm 0.20 \text{ m}^3/\text{sec}$.
- 4: An modern eskimo wishes an electric heater to turn on, **Q1**, whenever the temperature inside his igloo drops below 15° C and an ice-block window to be open, **Q2**, whenever the inside temperature goes above 29° C . Using the temperature sensor of Problem #4, design and simulate a suitable PLC system for this igloo.

- accumulator, 30
- AI, 81
- AM, 81
- analog amplifier, 81
- analog comparator, 85
- analog differential trigger, 83
- analog digital converter, 79
- analog multiplexer, 86
- analog threshold trigger, 83
- analog value monitoring, 85
- analog watchdog, 84
- AQ, 81
- arrowheads, up or down, 79
- assembly language*, 8, 30
- asynchronous pulse generator, 55
- automatic button pusher, 47
- automatic control, 44
- Banluta, iii
- basic functions, 14
- Bedford Associates, 3
- block name, 38
- block number, 17
- block properties, 37
- button pusher, automatic, 47
- clock icon, 64
- closing, 15
- coil, inverted, 36
- coils*, 7
- comparator, analog, 85
- Computer Engineering*, iii
- connecting cable*, ii
- constants, 13
- contact, break, 17, 26
- contact, make, 17, 26
- contacts*, 7
- counter, up/down, 45
- Data memory*, 4
- de-multiplex, 31
- Digital computers, 3
- download, 31
- drawing pictures, 12
- duration, 50
- edge-triggered AND, 39
- edge-triggered wiping relay, 53
- Electrical Engineering*, iii

- Electronic & Communication Engineering.*, iii
- every month, 63
- exclusive OR, 28
- expansion modules*, 4
- fan motor, 1
- flags, 35
- flags, IM, 43
- flags, internal, 43
- Flash memory* chip, 4
- flasher, 56
- float switches, 44
- function, 41
- function blocks diagram, 6
- functions, timing, 50
- gain, 81
- go to partner, 68
- grid lines, 14
- help, context sensitive, 16
- hours counter, 63
- icon, AI, 81
- icon, AM, 81
- icon, AQ, 81
- instruction list, 8, 18, 27
- invert connector, 29
- ladder diagram, 6, 16
- ladder rung, 32
- language conversion, 20
- latch, 34
- latching relay, 37
- limits, 43
- linear motion module, 71
- LOGO! 12/24RCo**, 12
- LOGO! 24 RCo**, 18
- LOGO!Soft Comfort, 9
- LOGO!Soft Comfort[®], ii
- M8 flag, 66
- machine language, 6
- memory cell, 38
- memory flags, 35
- memory location, 36
- memory usage, 40
- memory, data, 25
- memory, input, 25
- memory, output, 25

- memory, parameter, 25
- menu bar, 13
- moving window, 12
- multiplex, 31
- multiplexer, analog, 86
- multivibrators, 35
- off-delay, 51
- offset, 81
- On > Off, 46
- on-delay, 51
- on-line test, 31
- on-off-delay, 52
- page layout, 48
- parallel contacts, 31
- parameter, 50
- parameters, screen, 54
- parameters, set, 46
- PIC, iii, 5
- pixxa, 45
- pizza, 47
- PLC, 3
- power bus, 17, 19
- pre-conditioning, 80
- printing, 20
- Program memory*, 4
- pulse relay, 69
- pulses per time, 61
- pump, 44
- push button, 24
- push button, momentary, 56
- push on - push off, 51
- random generator, 56
- random pulse generator, 57
- relay, 9, 24
- remembering, 34
- reset, 35
- resizing window, 12
- retentive on-delay, 53
- retentivity, 38
- rotary switch, 65
- rotating drum, 2
- RS, 37
- RS flipflop, 35
- RS gate, 35
- rules, programming, 25
- run in hardware, 30
- rung, 32
- S** memory locations, 65
- saving, 15
- scan cycle, 9
- scan time, 5
- Scan time, iii
- scanning cycle*, 4, 18
- scanning cycle, duration, 62
- Schneider, ii
- Scratch-pad memory*, 4
- select hardware, 30, 79
- selection, 14
- sensors, 80
- separate, 66
- set, 35
- shift register, 64
- Siemen, ii
- Simantic*, ii
- simulation, 20
- special functions, 14
- stack, 30
- stepper motor, 74
- structured text, 8, 18
- switch, 24
- switch, rotary, 65
- threshold trigger, 61
- timing functions, 50
- toolbar, simulation, 20
- toolbar, standard, 13
- toolbar, tools, 36
- transfer, 31
- trigger, edge, 50
- trigger, level, 50
- truth table, 27
- Twido*, ii
- TwidoSoft, 8
- up / down counter, 45
- watchdog, 84
- weekly timer, 63
- wiping relay (pulse generator), 57
- wire, 15
- wire technology, 2
- www.siemens.com, ii
- XOR**, 28
- yearly timer, 63
- Zamora, iii

zoom, 13