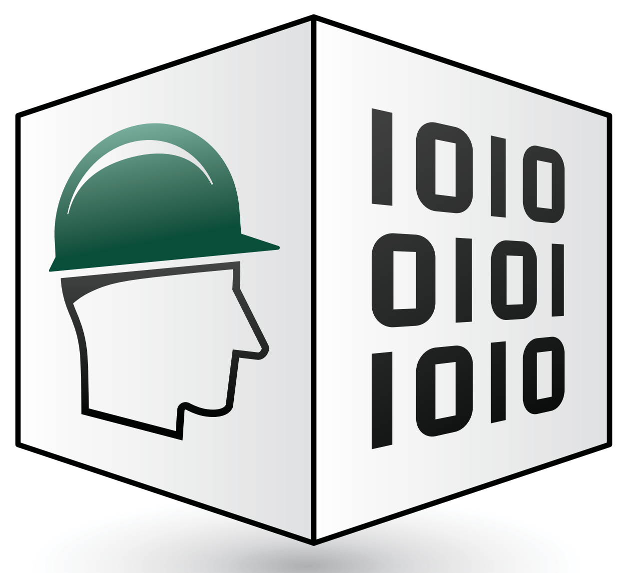
2022 EDITION

PLC TECHNICIAN HANDBOOK

Practical Guide to Programmable Logic Controllers

PLC TECHNICIAN HANDBOOK

Practical Guide to Programmable Logic Controllers



www.gbctechtraining.com

© 2022 George Brown College

160 Kendal Ave, Toronto, ON M5R 1M3

PREFACE

PLCs have become an integral part of manufacturing in the twenty-first century and now dominate industrial automation. To have a meaningful and successful career in the field requires a thorough understanding of the key foundational skills and theoretical concepts of PLCs.

The PLC Technician's Handbook was developed to be a compact collection of fundamental content for practicing automation and PLC technicians to reference through their career. The handbook is broken down into three sections for easy reference: first, a series of articles including basic programming & tips and common practices, second, programming examples, and third, supplemental information such as instruction sets of the main ladder logic programming commands and important schematic symbols.

Special thanks to Dwayne Nehls, Shimona Babb and Surajit Barua for their valuable assistance and contribution to the creation of the handbook.

We hope that this handbook helps you develop a fuller understanding and practice with PLCs and becomes a useful reference to you in the future.

TABLE OF CONTENTS

1. PLC Technician Handbook

GENERAL OVERVIEW OF PLCS

What are the essential elements of a PLC system? 5

How PLCs are applied in various industries 8

The evolution of PLCs 10

BASICS OF PROGRAMMING

Different types of PLC programming languages 13

Basic Instructions and Operation of a PLC 17

Time Driven routine segments 19

Event Driven program segments 21

Counters 22

TIPS AND COMMON PRACTICES

SCADA, what is it and how does it work 24

File based addressing 26

Reducing Scan time 35

2. Program Examples

PLC Timer Instructions 44

Decision Making 49

3. Supplemental Content

Common PLC Field Devices and Schematic Symbols 57

Number Systems and Codes used with PLC 61

Core Instruction Set 76

WHAT ARE THE ESSENTIAL ELEMENTS OF A PLC SYSTEM?

The world of manufacturing has been changing rapidly over the past decade. Processes are largely automated and as a result, the quality of products that are produced and the efficiency of systems that produce them is at an unprecedented level. Despite these changes, programmable logic controllers have remained a critical component of these systems despite being invented in the 1960s.

PLCs come in a variety of sizes and with different capabilities, but all include the following six essential systems:

- Central processing unit
- Rack or mounting
- Input assembly
- Output assembly
- Power supply
- Programming unit

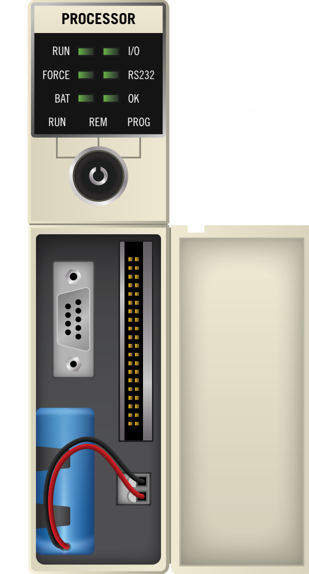
Here's an overview of each and what they do.

Processor or Central Processing Unit (CPU)

Often referred to as the "brains" of the PLC, the processor or central processing unit is responsible for the execution of commands. The industry has made an effort to use a standardized list of programming languages, namely:

- Structured Text
- Ladder Diagram
- Sequential Function Chart
- Instruction List
- Function Block diagram

In spite of aforementioned list of languages being used, it's important to note that each manufacturer has a different method of implementing the code.



Processor

Rack or Mounting

Though not a universal truth, most medium to large PLC systems are assembled so that their individual components (things like the CPU or processor, the I/O, and power supply) are held together within a mounting or rack. Smaller PLC systems often contain all of the components into one compact housing (often referred to as a shoebox, or brick).

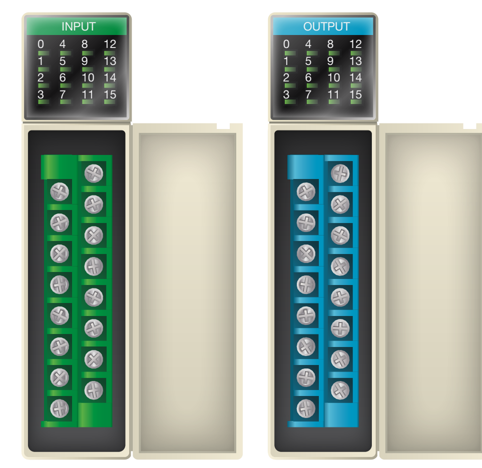
Input Assembly

In terms of functions, the input assembly has two. The first is to receive external signals from field devices and control stations (switches, sensors). The second is to display the input point status.

Output Assembly

Outputs are the pieces of equipment used by the PLC to execute the commands, and are typically used to control a manufacturing process (motors, pumps, actuators, lights, etc.).

There are two types of I/O: Analog I/O and Specialty I/O. The differences between the two are outlined below.



Input/Output Interface

1. Analog I/O refers to inputs and outputs that are responsible for a range (e.g. the running speed of a motor). This type of input operates based on continuous change of variable ranges (temperature, pressure, etc.) In this case, the output would include setting the speed of the motor.
2. As the name implies, Specialty I/O performs specific tasks, controlling things like high speed counters. Digital I/O operates in using a binary change (yes/no or on/off).

The PLC Technician program uses the PLCLogix simulation software to teach students about the I/O assembly. The image above is an example of the software's interface, containing two discrete input modules, two discrete output modules, two TTL modules for BCD control and display, and two analog modules.

Power Supply

If the CPU is the brains of the PLC, then certainly the power supply is the digestive system, commonly supplying the unit with a 24VDC or 120VAC line voltage. The power supply status is continuously monitored and can include a switch for selecting a particular programming mode. Thanks to the power supply's integral lithium battery, in the event of a power failure the contents stored in memory will not change from what they were prior to the loss of power.

Power Supply

Programming Unit, Device, or PC/ Software

The modern PLC is programmed using a programmer or software that is built using a PC or laptop and then loaded into the PLC. PLC software's big advantage is that they can run simulations to see how a PLC system will perform in a virtual environment. RSLogix is one of the most popular simulation tools today.

At George Brown College, our students learn using these same virtual environments. We use PLCLogix software, which emulates RSLogix and provides the

ability to test and practice PLC programs. Students can apply the knowledge acquired from this program in the real world PLC environments.

PLCs Are More Relevant than Ever

In truth, it's difficult to imagine a world in which PLCs did not exist. The manufacturing process would certainly not be what it is today, and would likely rely more heavily on greater quantities of human capital to achieve similar levels of production. Even though the base technology responsible for the typical PLC is decades old, PLCs remain a crucial component to manufacturing, robotic and automated systems.

HOW PLCS ARE APPLIED IN VARIOUS INDUSTRIES

In the most basic terms, a programmable logic controller (PLC) is a computer that's equipped with a microprocessor, but has no keyboard, mouse, or monitor. Though this description might make it seem like PLCs suffer from limited functionality, quite the opposite is true. In fact, PLCs are built to drive the most complicated manufacturing processes and withstand very harsh industrial environments. This article will examine how these sophisticated machines are used in different industries.

Oil Industry

Public opinion may be shifting away from fossil fuels in favor of more ecologically friendly alternative energy sources, but there is still a strong demand for the extraction of oil and natural gas. Unfortunately, many of the world's remaining reserves are difficult to access, meaning that companies are pivoting away from the historically simple, vertically drilled, single well operations to multi-well operations made possible by directional or non-vertical drilling. While this process is more efficient (multi-well operations allow companies to drill more than one well at each site, meaning that operations can be centralized and less impact to the surrounding environment), it is also more complicated, so a more robust system is needed to control the site's equipment.

For accurate readings of inputs and outputs and to control the multitude of valves, pumps, and sensors that allow each well and well pad to operate safely is controlled by its own PLC. PLCs are often paired with a Human Machine Interface, or HMI, that allows users to monitor the health of the system, manually override controls, view system alarms, or get a glimpse of the system in operation in real time.

PLCs allow resource gathering operations to scale up in size with relative ease, since additional PLCs can be added to the system as new well pads are added. PLCs and HMI systems also help maintenance crews quickly identify potential issues on site, resulting in less wasted time performing system diagnostics.

Glass Industry

While PLCs can efficiently control actuators and valves and perform actions based on data gathered by sensors, these are not the only uses for PLCs in an industrial setting. In fact, the glass industry has widely used PLCs for many years to help manage the precise material ratios required by the production process. The manufacturing of glass is a surprisingly complicated endeavor and is largely made possible in part by the robust data gathering capabilities and advanced quality control afforded by PLC technology when paired with a Distributed Control System (DCS). At one point, glass manufacturers relied exclusively on their DCS for their operations, but the associated high cost of these systems propagated a need for a less costly alternative.

In recent years, the equipment used in the manufacture of glass has become more sophisticated, further increasing the demand for PLC solutions that are based on a DCS, which in turn is driving the demand for technicians who have completed PLC programming courses.

Cement Industry

Much like the production of glass, the cement industry relies heavily on equipment and software capable of mixing various raw materials with consistency so that the quality of the finished product is at a consistently high

level. The role of the PLC in this industry varies, but in particular, PLCs control ball milling, coal kiln, and shaft kiln operations.

Under Control

Admittedly, the above represents just a few of the industries that rely on PLCs in the manufacture of their products. PLCs can be found in most factories when automation is used including those that mass produce food, vehicles, textiles and more.

THE EVOLUTION OF PLCS

Nowadays, with almost every service imaginable available to us on our smartphone, it's difficult to envision what technology was like before the invention of a single controller. For factories and manufacturing plants, the invention of programmable logic controllers (or PLCs) revolutionized the industrial automation process, as its cutting-edge technology replaced a complicated system of electromagnetic relays with a singular controller.

In industrial workplaces, PLCs are still the preferred choice over other systems. The system has evolved to incorporate modern technologies, while maintaining the durability of the design to withstand the conditions in a factory setting.

In order to gain the best understanding and appreciation of PLCs, it's important to look at the history of the PLC, how it has evolved, and why it still remains one of the most important automation inventions used today.

What Are PLCs?

Programmable logic controllers are small industrial computers. Their design uses modular components in a single device to automate customized control processes. They differ from most other computing devices, as they are intended for and tolerant of severe conditions of factory settings such as dust, moisture, and extreme temperatures.

Industrial automation began long before PLCs. In the early 1900s until their invention, the only way to control machinery was through the use of complicated electromechanical relay circuits. Each motor would need to be turned ON/OFF individually. This resulted in factories needing massive cabinets full of power relays.

As industrial automation continued to grow, modern factories of the time needed dozens of motors with ON/OFF switches to control one machine, and all these relays had to be hardwired in a very specific way. PLCs were developed as a solution to have one solid control as an electronic replacement for hard-wired relay systems.

First Automotive PLCs

In 1968, the invention of the first PLC revolutionized the automation industry. First adopted by the automotive sector, General Motors began to deploy PLCs into their operations in 1969. Today, PLCs have broadly been accepted as the standard automated control system in manufacturing industries.

Known as "The Father of the PLC," Dick Morely first came up with the vision of a programmable controller which could work for every job. He put the proposal together on January 1, 1968. Along with the team at his company (Bedford and Associate) they created a design for a unit which would be modular and rugged while using no interrupts. They called it the 084, which was named after their 84th project.

At the same time as the 084, Bill Stone with GM Hydramatic (automatic transmission division of General Motors) was having the same issue: problems with reliability and documentation for the machines in his plant. His solution proposed a solid-state controller as an electronic replacement for hard-wired relay systems.

For this reason, Morely insists he is not the inventor of the PLC. Morley stated: "the programmable controller's time was right. It invented itself because there was a need for it, and other people had that same need."

PLCs were designed so that they could easily be understood and used by plant engineers and maintenance electricians, using a software called Ladder Logic. Widely used in PLCs today, Ladder Logic is a programming language which uses ladder diagrams which resemble the rails and rungs of a traditional relay logic circuit.

The Evolution of PLCs in Industrial Automation

After their initial success with the 084, Bedford and Associates changed its name to Modicon PLC, which stood for Modular Digital Controller. Modicon 084 became the name associated with the very first PLC.

In the next few decades, the PLC evolved in numerous ways to adapt to various environments and integrate the latest modern technologies.

The emergence of competitors who developed similar systems which rivalled Modicon sparked the need for new innovations. As a result, the development of the “Data Highway” by Allen-Bradley and “Modus” by Modicon allowed PLCs to exchange information with each other.

As PLCs became more widely adopted, the need for a vendor-independent standardized programming language for industrial automation led to the introduction of the IEC 61131-3 standard — the international standard held for PLC software made by the International Electrotechnical Commission.

At the start of the 1990s, end users began making special requests. Plant managers wished for the new machinery to have industrial terminals with PLC monitoring software. They wanted machines which could tell the technicians what was amiss rather than spend hours troubleshooting; this resulted in the development of the programmable human-machine interface (HMI).

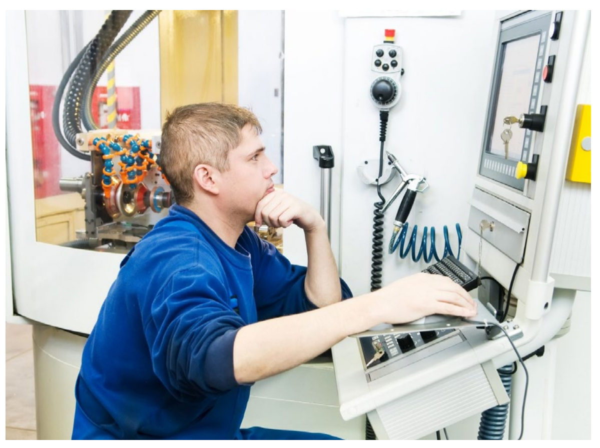
The implementation of HMI with new devices eventually brought internet connection to the factory floor.

PLC Programming in the Workplace Today

In modern industrial factories and manufacturing plants, the PLC is still the preferred system over PCs. As their technology continues to evolve, there is a steady demand for skilled PLC Technicians who are qualified to install, repair and maintain the systems.

Designed to be a more streamlined alternative to relay systems and switch boxes, PLCs have a dedicated OS and limited functionality. This significantly reduces the risk of malware attacks compared to other computing devices. PC devices require top-of-the line antivirus software and constant monitoring- a headache which PLCs, for the most part, avoid.

The design of the PLC has always been able to withstand the extreme temperatures, chemicals, vibrations, and other conditions of an industrial setting. Compared to the original Modicon 084, PLCs today are a fraction of the size, have considerable solid-state memory, and the most prevalent enhancement for the industry, drastically improved speed.



Technician operating a PLC

There continues to be an increase in demand for skilled and qualified technicians who receive PLC training. Many technical colleges and institutions offer PLC training online as a convenient option for those who wish to expand their professional qualifications while still working. Adding a PLC Technician Certificate to your existing resume opens the possibility for new job opportunities, promotions, and higher pay.

PLCs Continue to Evolve

PLCs remain one of the most important innovations in the history of industrial automation. These devices, which replaced complicated electromagnetic relays with a single controller, are still widely used today. The design has evolved to improve the technology, while maintaining its durability for factory settings.

Qualified PLC Technicians are valuable to any manufacturing plant or industrial workplace. As the technology of the PLC evolves, there is a demand for employees with skills in the fundamentals of PLC programming. PLC classes online offer simulation software which enables you to design, run, test, and debug ladder logic programs and simulate the operation of real-world PLC applications.

DIFFERENT TYPES OF PLC PROGRAMMING LANGUAGES

Programmable Logic Controllers (PLCs) are general-purpose control devices often used in plants or manufacturing systems. They provide both a useful and efficient control solution for a variety of applications, and can accept inputs from a variety of devices, such as motion detectors, joysticks and buttons, amongst others. In turn, PLCs are able to produce outputs that control lights, motors and sound effects, amongst others. Programming resources for this input-output system can include timers, counters and other variables.

While PLCs can be very effective in their commercial applications, they come with a steep learning curve. There is some difficulty in learning their functionality as well as inherent difficulty with program maintenance using their traditional Relay Ladder Logic (RLL) programming language. It should also be noted that PLCs from different manufacturers can be programmed in various ways.

If you're an aspiring PLC technician or considering enrolling in an online PLC training course, there are three primary PLC programming languages of which you should be aware.

1. Function Block Diagram

Function Block Diagram (FBD) is the fundamental language for all PLC programmers. It's relatively simple in nature and programs functions together within a PLC program graphically.As its name would suggest, FBD allows a PLC technician to put functions written with lines of code into boxes, or blocks.

You can then connect these boxes to create the larger PLC program. Almost all PLC programs are written, at least partially, with FBD, because it offers the technician the ability to connect various functions together. The function blocks are integral to this programming language, as they delineate the relationship between input and output functions.

Within the FBD language, there are a few standard blocks. Amongst the most important ones include:

- Bit Logic Function Blocks
- Bistable Function Blocks
- Edge Detection
- Timer Function Blocks
- Counter-Function Blocks

There is an infinite number of function blocks provided within a FBD, and oftentimes, one exists for almost every operation that can be performed within PLC programming, including:

- Arithmetic Function Blocks;
- Bit Shift Function Blocks;

- Character String Function Blocks;
- Conversion Function Blocks;
- Communication Function Blocks

Additionally, many PLC technicians and enthusiasts will often build their own function blocks.

2. Ladder Diagram

Also called Ladder Logic, Ladder Diagram (LD) is a visual PLC programming language, which one can learn fairly quickly. Those with experience with electric relay circuits may find LD programs relatively easy to grasp, as the two look very similar. The organization, PLCOpen, has established the standards for LD making it one of the only standardized PLC programming languages. Essentially, each function is coded into a rung and once many rungs join together in a program, they make what looks like a ladder.

LD was created for technicians and electricians who have a background reading and understanding electrical circuit schematics. Rather than using text, LD programming uses graphic elements called symbols, which have been made to look like electrical symbols. An important difference, however, is that while electric circuits are drawn horizontally, LD programs are created vertically.

As you construct the LD program vertically, the PLC will execute one rung (or symbol) at a time, as each symbol in the ladder is an instruction. When you create a new piece of ladder logic, you will notice two vertical lines, and it's in between these two lines where you're ladder logic will live. You'll proceed to draw vertical connections between the original lines, creating the rungs of information. You can then proceed to include any of the aforementioned symbols within these rungs, forming the instructions for the PLC. Executing the program one rung at a time, the PLC will typically scan all of its inputs and then proceed to execute the program to set outputs. A few common symbols, or instructions, include:

Examine If Closed

Output Coil

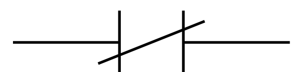
Output Latch

Examine If Closed: looks like two short vertical lines parallel to one another with the name "I0.0" found above the symbol. This is a conditional instruction and is often used to check whether or not something is true, for example, it can check if a bit is turned on. When the PLC checks the state of its inputs, it will assign a boolean value in its memory (either 1 or 0). If an input is low, the bit will be set to 0, alternatively, if an input is high, the bit will be set to 1.

Output Coil: looks like a set of parentheses and is used to turn a bit on and off.

Output Latch: allows you to direct the PLC to perform a continuous output even if the digital input is, let's say, a momentary pushbutton (that is, a device that needs to constantly be pushed down to work). This

is especially handy, for example, when dealing with a fan for a ventilation system where it would be inconvenient for the operator to continuously hold down the fan's button.



Examine If Open

Examine If Open: is another symbol with the memory address "I0.1." This function looks like the Examine If Closed symbol, but with a diagonal line crossed through the two vertical lines, and works in the exact opposite way of Examine If Closed.

There are a number of other symbols involved in LD programs, a few of which include:

Program Control Instructions			
Instruction Mnemonic	**Instruction Name**	**Symbol**	**Description**
JSR	Jump to Subroutine	**JSR** Jump to Subroutine Routine name ? Input par ? Return par ?	This instruction jumps execution to a specific routine and initiates the execution of this routine, called a subroutine.
SBR	Subroutine	**SBR** Subroutine Input par ?	Stores recurring sections of program logic.
RET	Return	**RET** Return Return par ?	Used to return to the instruction following the a JSR operation.
JMP	Jump to Label	? JMP	Skips sections of ladder logic.
LBL	Label	? LBL	Target of the JMP instruction with the same label name.
MCR	Master Cont. Res.	MCR	Used in pairs to create a program zone that can disable all rungs between the MCR instructions.

...

...

NOP	No Operation	—(NOP)—	This instruction functions as a placeholder.
END	End	—(END)—	End rung in ladder logic circuit.
AFI	Always False Instruction	—[AFI]—	Sets the rung condition to False.

3. Structured Text

The Structured Text (ST) programming language is text-based and is often regarded as one of the easiest languages to understand for beginners and for those building programs that will be read by others. While graphics-based programs, such as the aforementioned FBD or LD, may seem easier to decipher, using a text-based language (such as ST) will take up less space and allow users to more easily follow the logic of the program. Another benefit of ST is that it can be combined with different programming languages. For example, you can create function blocks containing functions written in ST, and because ST is a standardized programming language you can proceed to program different PLC brands with it.

Similar to ladder logic, programs written in ST are executed one line at a time. The basic syntax of ST revolves around “Program” and “End_Program” which sandwich your PLC program, as seen below:

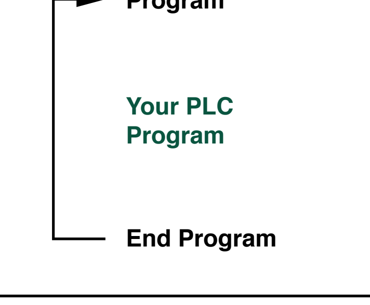
It’s important to note that the “End_Program” command will not end your program definitively, but rather instruct the PLC scan cycle to start over again causing your program to repeat itself.

Your PLC programming software will likely implement the “Program”/”End_Program” construct automatically, prompting you to write the code needed to fill the construct. While there are many syntax details that govern ST, there are some general rules of which you should always be mindful:

All statements are divided by semi-colons;

- ST is not case-sensitive: while it’s good practice to use sentence-case for readability, it’s not necessary
- Spaces have no function: similar to using sentence-case, using spaces improve readability.

Depending on their manufacturer, PLCs use a wide array of different programming languages. In much the same way humans around the world speak in different languages and dialects, so too do PLCs with the same end goal: to communicate with each other and execute functions.



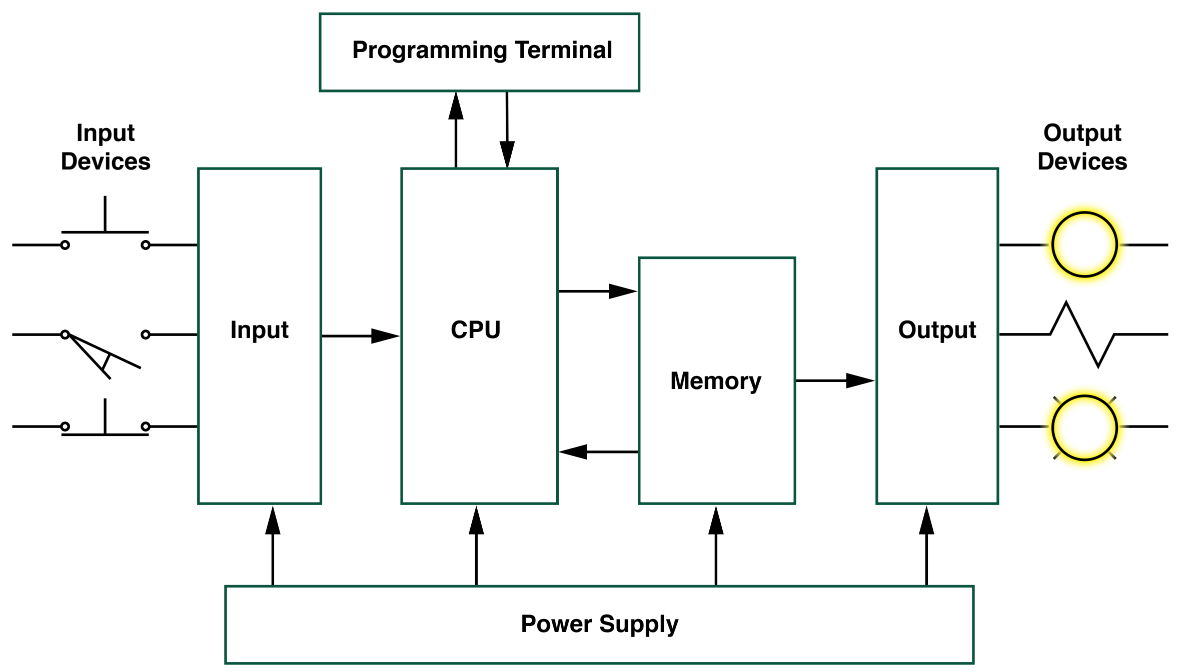
Structured Text

BASIC INSTRUCTIONS AND OPERATION OF A PLC

Let's look at the basic structure of a PLC and how it operates. In general, a PLC takes stimulus from the outside world and brings it into a computing environment. Decision or monitoring instructions within the PLC process the input information and instruct the outputs of the PLC to react to the stimulus in a determined manner. Field devices bring status information to the PLC, the program interprets the information and will then discern the appropriate actions or output to provide in response to the given input conditions. The adoption of PLC's has resulted in the replacement of the majority of Relay Control Logic which has been historically used to control applications.

What Are the Basic Components of a PLC?

In addition to the Input and Output rack locations, which most PLC technicians are familiar with, a PLC contains isolation circuitry, a CPU, data and address busses, memory storage, a power supply, and usually some kind of external MMI or man machine interface or terminal for programming. The number of types of input and output modules will vary according to the needs of the application for which the PLC is being employed.

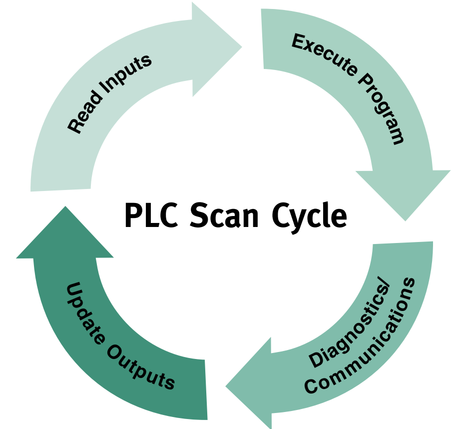


Major Components of a PLC System

Above, we have a functional diagram representing the major components of a PLC system. The input and output devices can allow for a range of voltages or currents, and can be digital or analog in nature. The specifics of the I/O are largely dependent on the type of control application being performed. The actual PLC contains several sections not detailed in the block diagram including facilities for communications.

What is a Scan Cycle?

Generally, PLC's operate by repeating what is called a "Scan Cycle". This cycle consists of four basic functions. First, the inputs are read, next the program instructions are executed after which diagnostics and any required communications occurs, and finally, the outputs are updated. One important aspect of the scan cycle is the "Scan Time". This is the total time it takes for the PLC to perform a complete scan cycle. This time can be critical in applications that require real time monitoring and control.



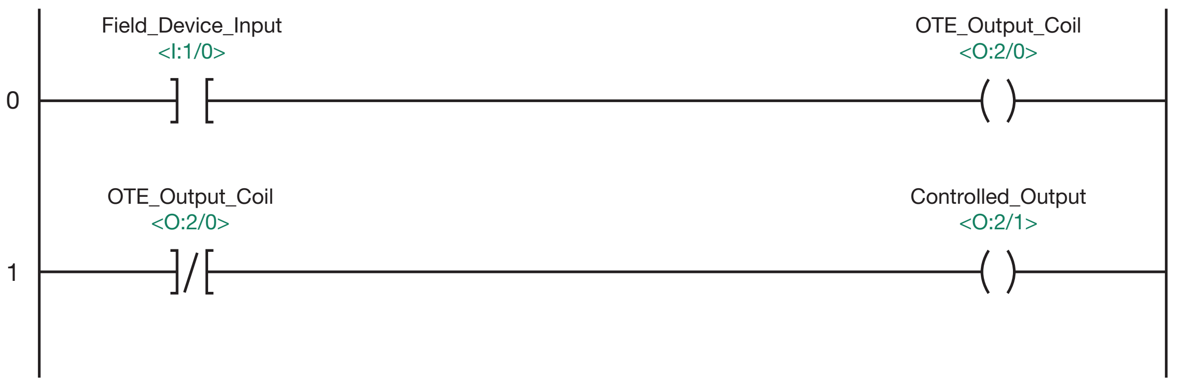
How Are PLC's Programmed?

A programming terminal is used to input or edit a program and allows the program to be uploaded from the PLC to the terminal where it can be edited or downloaded to a PLC where it can be run. These terminals can be PC computers, or simple dedicated HMI interfaces.

There are several types of languages or approaches used for programming PLCs. These include Structured Text, Instruction Lists, Function Block Diagraming, as well as Ladder Logic programming. For the benefit of those technicians who have no programming experience, we will be working with the ladder logic programming approach in our discussions going forward. It uses a format that would be the most familiar for technicians who work with electronics and industrial control equipment. This programming language is based on relay logic as was used traditionally for industrial control applications so many aspects of it will be familiar to those technicians who have been working in the field.

Why Ladder Logic Programming?

The primary reason ladder logic programming exists is that it allows technicians to easily interpret a PLC's program by rendering the program in a format very familiar to technicians who have worked with relay logic controls prior to the advent and wide adoption of PLC's. This programming language is very intuitive for those familiar with electrical wiring and relay control circuits and is easy to learn. Many technicians have worked with relays in the past. The physical device contains a coil, and one or more contacts that will change state depending on whether the coil is energized or not. This type of device is modelled in ladder logic programming by a OTE instruction, (serving as the coil) and one or more NO or NC contacts that are controlled by the coil.



Ladder Logic Example Diagram

TIME DRIVEN ROUTINE SEGMENTS

Application control programs generally consist of 2 types of routine segments; time driven segments and event driven segments that generally either direct a series of timed events or provide a desired response to changing conditions in the environment. In addition to these two basic routine segment types, programs often also include a means of repeating routine segments a desired number of times. Here we examine Time Driven routine segments with a simple practical everyday example.

What's an Example of a Time Driven Routine Segment?

Time driven routines employ timer instructions (TON and TOF as well as RET) to perform timed tasks. Timers can be run concurrently to address different aspects of a single task or can be cascaded to perform a sequence of timed events. Timers are versatile and are widely employed in PLC programming applications. An everyday example of a timed routine segment can be seen in the control of an elevator door. The elevator door opens, the doors remain open for a period, and then the elevator door closes. Each part of this door sequence is being performed for a specified time duration. This is accomplished by using timer instructions in conjunction with the appropriate outputs for door motor control. In addition to the cascaded timer type applications, another common timer application is a reciprocal timer. In that instance 2 timers work together to provide an ON and OFF timing duration.

The ON duration timer is used to start the OFF duration timer. The OFF duration timer is used to reset the ON duration timer and restart the cycle. This type of application can be used for something as simple as flashing lights.

How Do Timer Instructions Work?

The T4 Timer data file is used to store information on timer instructions being used in a given ladder logic control application. The most common timing instruction is the "on-delay" (TON) timer.

As a quick visual review, here is the T4 timer file data structure for a single timing instruction. We recall that this structure consists of 3 Words of 16 bits length each. This allows for the storage of timers bit status, (DN, TT, EN) as well as the "Preset" and "Accumulated" time values.

Addressable Bits in Control Word (Word 0)

Bit 13 Done (DN)	T4:5/13 or T4:5/DN
Bit 14 Timer Timing (TT)	T4:5/14 or T4:5/TT
Bit 15 Enable (EN)	T4:5/15 or T4:5/EN

Addressable Words for Preset and Accumulated Values (Word 1, Word 2)

Preset Value (PRE)	T4:5.1 or T4:5.PRE
Accumulated Value (ACC)	T4:5.2 or T4:5.ACC

The TON delay timer instruction has 3 useful status bits, the enable (EN), timer timing (TT), and done (DN) bits. These are the primary bits used to drive timed routine segments. Below, find the basic operation of this type of timing instruction explained as well as how the status bits respond during operation. This TON instruction can be used to turn on or off an output after a given period of time has expired. The instruction counts time base intervals (0.1s) when the rung condition it is on becomes TRUE. As long as the rung condition remains true, the timer modifies its accumulated value (ACC) time each iteration until it reaches the preset value (PRE). The accumulated value gets reset when the rung condition goes false regardless of whether the timer has timed out or not. The status bit behaviour is as follows:

Setting TON Status Bits		
This Bit	**Is Set When**	**And Remains Set Until One of the Following**
Timer Done Bit DN (bit 13)	accumulated value is equal to or greater than the preset value	rung conditions go false
Timer Timing Bit TT (bit 14)	rung conditions are true and the accumulated value is less than the prset value	rung conditions go false or when the done bit is set
Timer Enable Bit EN (bit 15)	rung conditions are true	rung conditions go false

In addition to the TON delay timer instruction, two other timer instructions are included in the basic instruction set. The TOF delay timer, and the RET or retention timer. The retention timer functions much the same way as the TON delay timer with a couple exceptions. The Accumulated value is not reset when the rung condition goes false, it is kept. This allows the time interval to be interrupted, and then resumed without losing the current time value. The other significant difference with this instruction is that it requires a reset (RES) instruction to clear its accumulated value.

Check out the video using cascading timers to make an elevator door routine in ladder logic. This video provides an excellent example of the instructions we have covered to date, namely, coils, contacts, and timers and provides an illustration of cascading timers to perform a cyclic operation such as opening and closing elevator doors.

EVENT DRIVEN PROGRAM SEGMENTS

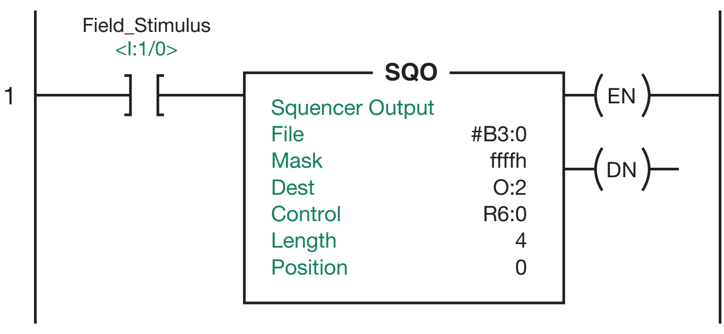
Let's review a simple Event Driven program segment using another staple of relay logic control systems, the drum sequencer. This real-world control device has an associated virtual ladder logic instruction.

What's an example of an Event Driven Routine Segment?

An event driven routine segment is one that is not synchronous meaning that there is no fixed or timed duration associated with input or output changes. In this type of scenario, a sequencer instruction might be utilized to walk/step through a series of actions required for a given "process" much in the way that a drum sequencer could be employed. A drum sequencer rotates and makes/breaks connections providing a pattern of control signals capable of driving field devices. With 500 series PLC programming, the SQO, (sequencer output) instruction performs this same task. The user defines a data table and populates it with the patterns to perform the given actions required for the "process". An external stimulus or "event" can be sensed and used to step through the data table. These external conditions are the "events" that drive the pointer through the lines of the data table containing the desired output states. The behavior mirrors that of the real world "drum sequencer" often employed in relay logic control and would be considered an event driven program segment when employed in a ladder logic program. This is in contrast to a time driven program segment, as there is no specific time interval associated with the occurrence of the stimulus or event.

How does the SQO output sequencer work?

In the ladder diagram below, we see an SQO sequencer instruction being stimulated by a single NO contact associated with input rack location I:1/0. When this input goes high, the sequencers pointer is moved to the next value in the data table defined for this instruction. This in effect, provides a simple example of an event driven routine segment. The field stimulus is asynchronous and can occur at any time. This stimulus results in changes to output conditions as the data table word that the current pointer location references is sent to the location defined in the Destination field of this SQO instruction. (in this instance, output rack location O:2)



SQO Squencer

EXAMPLE

Click on the link that illustrates the operation of an SQO while explaining how Masking is accomplished when sending a data table to a specified output location. Masking is explained and the operation of the sequencer pointer is illustrated.

The pointer will move from through these locations, and when it reaches B3:4, it will wrap back to location B3:1. The O:2 output will be passed each value stored from B3:1 to B3:4. The maximum number of values that can be stored in any one data table is 255 as we need 1 location to be designated as the origin of the table and as we learned in our earlier discussion of data files, the B3 file can hold 256 words.

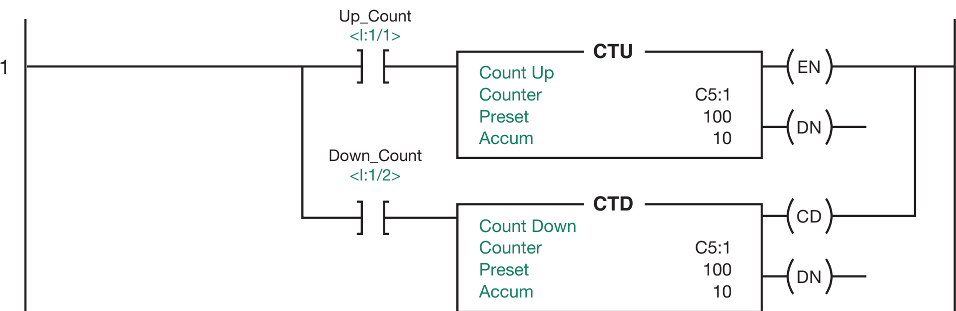
COUNTERS

We will take a brief look at how the virtual version of a counter is implemented when programming modern day PLCs to perform industrial control applications. We will also illustrate how a technician can become familiar with the wide variety of additional instructions used with the AB SLC 500 series of controllers. What is the difference between the CTU and CTD counter instructions?

When programming PLCs, we encounter 2 basic types of counters, an UP counter or CTU instruction, and a DOWN counter or CTD instruction. As you may have intuited, CTU counter instructions start at a user defined value (stored in the ACC or accumulated field of the instruction), and count up or increment the value each time a “false to true transition” occurs on the rung on which the instruction is located. The CTD instruction is used to decrement a user defined (again a value stored in the ACC field for the instruction) value each time the rung on which this instruction is located exhibits a “false to true” transition. In both instances, a field device such as a special purpose sensor, physical wand, or a proximity detector causes an input rack location to change states. Information pertaining to these counter instructions is stored in the C5 (counter) data file location. In addition to the individual counter types, (up or down counter), it is possible to combine these instructions to create a counter bi-directional counter or “counter pair”.

How do you create an UP/DOWN counter pair?

In addition to the preset and accumulated value fields previously mentioned, status bits and the counter reference can be shared by these two counter types. The memory location structure of the C5 file provides storage for user defined values as well as status bits.



The C5 Data File							
	/CU	/CD	/DN	/OV	/UN	.PRE	.ACC
C5:0	0	0	0	0	0	0	0
C5:1	0	0	0	0	0	100	10
C5:2	0	0	0	0	0	0	0
C5:3	0	0	0	0	0	0	0
C5:4	0	0	0	0	0	0	0
C5:5	0	0	0	0	0	0	0
C5:6	0	0	0	0	0	0	0
C5:7	0	0	0	0	0	0	0

Counter Ladder Logic Circuit

In the above illustration, the "Counter" field for both the CTU and CTD instructions reference the same memory storage location, C5:1 (also shown in the figure). Each memory location in that file stores the CU (up count enabled), CD (down count enabled), DN (done), OV (overflow), UN (underflow) status bits as well as the Preset and Accumulated value for counters that share the C5 reference. In this example, the Preset value is set to 100. Notice that value is shared and displayed as the Preset value for both counters. The current Accumulated value is 10 and it is also shared and displayed in each instruction. When the I:1/1 contact closes, the CTU counter is triggered. This will increment the Accumulated value stored in the C5:1 memory location and the modified value will be displayed in both instructions. When the I:1/2 contact closes, the Accumulated value will be decremented and will return to the value of 10. Each of the existing counters is capable of modifying this Accumulated value. They also share bit status. In this way, a running count can be incremented or decremented by field events (state transitions at the I:1 input module referenced by the contacts on the rung.

EXAMPLE

Click on the link to demonstrate the behavior of an up/down counter pair employed to keep a running count value that can be incremented or decremented as required. The video outlines an elevator hoist routine that uses a running count to track the elevator car location.

Is sharing a data file storage address among multiple instructions a common thing?

There are other instructions capable of employing a shared memory location reference. Another common instruction that employs the use of this reference sharing technique would be the SQO sequencer output instruction. It is possible to have several of these sequencer instructions all sharing the same R6 control data file memory location, while referencing different Source file locations. The objective in this instance would be the sharing of the POS (position) value. This would move the file pointer in each of the referenced source files in step with each other. Technicians would use this facility to create a virtual Sequencer Table and send parts of the table data to different output or instruction field locations. A sequencer table could store the output pattern for the lights in a traffic light application.

A separate sequencer table could store the timing information for each step in the light sequence. As the pointers move together, each output light pattern (being sent to the output locations controlling the lights) is stepped through, the corresponding time duration for each step can be sent to the Preset field of a timer instruction.

How do you research the behavior of the wide variety of instructions available with SLC 500 series controllers?

The basic extended instruction set is detailed in the vendors User's Instruction Set Reference Manual. The guide lists the available instructions, and describes the operation and data associated with each instruction. As the instructions are too numerous to be expounded upon in this format, it is important to familiarize yourself with this resource.

SCADA SYSTEM: WHAT IS IT AND HOW IT WORKS

When we think about technologies which have revolutionized industrial automation, we often think about a specific piece of hardware we interact with, rather than the software behind the scenes. For factories and manufacturing plants, the Programmable Logic Controller (PLC) dramatically improves efficiency and control; yet it's the SCADA software which runs all the operations smoothly.

To understand the basics of SCADA systems, it's important to know what they are, their functions and components, and their place in modern industrial workplaces.

What is a SCADA System?

If you've worked in a factory, chances are you've heard the term SCADA. However, if you're not familiar with factory floors, this might be a foreign concept. SCADA systems are an industrial control system at the core of many manufacturing plants' daily operations. SCADA stands for Supervisory Control and Data Acquisition. Simply put, SCADA systems gather and quickly analyze real-time data. In the manufacturing sector, they're used to monitor and automate the control processes of industrial automation.

The basic SCADA architecture begins with PLCs or Remote Terminal Units (RTUs). These microcomputers receive data from the sensors and machinery within a factory or operation, then route the information to computers running the SCADA software. The software processes, distributes, and displays information on a Human Machine Interface (HMI) for a human operator to make decisions based on the information.

SCADA systems were first introduced to the factory floor after the implementation of industrial computers, primarily PLCs. The term "SCADA system" was coined in the early 1970s, as the software which allowed automated communications to transmit data from remote sites to monitoring equipment. Some of the biggest industries that use SCADA include oil and gas, food and beverage, automotive, and chemicals. A basic SCADA system has several key components and functions, which is what we'll explore below.

Functions of a SCADA System

The software and hardware elements of a SCADA system work together to perform the functions which collect, analyze and display real-time data from factory operations. Modern SCADA systems offer the ability to monitor and control various processes from a remote location.

A SCADA system has four primary functions: data acquisition, network data communication, data presentation, and control.

1. Data Acquisition

SCADA systems acquire data from sensors and network devices connected to PLCs. They measure parameters such as speed, temperature, weight, flow rate, gaseous emissions and pressure. This raw data is then sent to a PLC to process, and then on to an HMI for a human operator to analyze and make decisions as required.

2. Network Data Communication

The use of wired or wireless communications technologies is important for SCADA systems when transmitting data between machines and operators. These networks allow multiple systems to be controlled from a central location.

3. Data Presentation

SCADA systems report data to either an HMI or a HCI (Human Computer Interface), where the information is displayed to a human operator. This master station continuously monitors all sensors and alerts the operator when there is an "alarm" or dysfunction - when a control factor is not functioning within normal operational range.

4. Control

SCADA systems can be programmed to perform certain control decisions based on data collected from the sensors. Control functions may include turning power on/off, adjusting temperature, decreasing or increasing speed, and regulating a variety of industrial processes.

Components of a SCADA System

SCADA systems are composed of numerous hardware and software mechanisms working together to perform the functions listed above. The hardware consists of data collection devices such as sensors, relays and switches. SCADA software analyzes and translates the data which is then sent to the operators, and also has the ability to be programmed for control and alarm functions.

Digital or analog inputs and sensors are responsible for measuring and controlling the status and parameters of a machine. Their primary function is data acquisition, which is then sent to the PLCs or RTUs. PLCs and RTUs are small industrial computers which collect data from the inputs and sensors and report the information in a meaningful way. They serve as local collection points for gathering reports and also deliver commands to control relays. Data collected from multiple PLCs is subsequently sent to a central HMI.

HMIs serve as the master and satellite computer stations which allow a human operator to analyze all the collected data from networked devices and sensors. The information is often displayed in graphical pictures and maps representing machines and devices, data charts, and performance reports. Based on the information, the human operator can make informed decisions to optimize the efficiency of the production process. The communications network is what allows data to be sent between the machines, PLCs, and operators. SCADA systems typically use a closed LAN for local geographical areas, or WANs to connect to different regions. Without a correctly designed communication network, a SCADA system would not be able to function.

SCADA Systems in Modern Industrial Automation

In modern industrial workplaces, PLCs are widely used as the device which communicates data from sensors and inputs to an HMI, for operators to make decisions about the manufacturing processes. However, it's the SCADA software which works between all the components to keep operations running efficiently.

FILE BASED ADDRESSING TIPS

This article provides an introductory understanding of the “File Based Addressing” scheme used with 16-bit (500 series) PLC’s from Allen Bradley. Although addressing schemes are proprietary in nature, Allen Bradley’s approach can serve as a valid representation of the addressing schemes generally employed by this generation of controller, and the principals set out in this article can be easily applied to 16-bit controllers offered by a host of other vendor’s. PLC memory allocation consists of 2 general areas: program files, and data files. Each section consists of 256 files, some of which are predefined, while others are flexible and can be used to suit the needs of a given application. The part of the file based addressing scheme that will be discussed in this article, resides within the 256 files contained in the Data Files section of PLC memory. Files 0 through 8 in the data files section are standard and predefined as shown in the figure below. Files 9 through 255 can consist of any of the available file types depending on the user’s needs. The figure below lists the 11 basic data file types, Output, Input, Status, Bit, Timer, Counter, Control, Integer, Floating point, String, and ASCII. These files store data and values pertaining to I/O, instructions, processor status, as well as variables in data tables. All these file types can be referenced using the File Based Addressing scheme.

Memory File Organization for the File Based Addressing System

What is an address?

An “address” is essentially a means of referencing a location in memory. Addresses allow for physical I/O as well as the data or status of instructions/elements to be accessed by the controller. These values are stored in the Data Files portion of the PLCs memory. The Data File section of memory is organized into 11 general file types. Each file type is denoted by a specific letter which is used at the start of all associated addressing.

What is I/O addressing?

The most familiar addresses encountered by technicians would be those addresses pertaining to physical I/O locations. The state of physical connection points on the I/O rack will be stored in the “I” and “O” files in the PLCs memory. The format for an I/O address will start with the file type, (I or O), followed by its slot number in the I/O rack, It will end with a reference to the specific terminal number (0-15) on the module of interest.

An input address related to terminal connection 12 on an input module situated in slot 3 of the I/O rack would have the address I:3/12. This address contains 3 alpha numeric characters and 2 delimiters. The address starts with the memory file type “I” denoting its location in the input file. The next item in the address is the delimiter “:”. This delimiter is used to separate the file type from the slot number containing the input module that is being referenced, in this instance slot 3. After the slot number is specified, a second delimiter (a backslash), separates the slot number from the actual terminal connection point (0-15) on the input module. The state of this terminal connection point, (high or low in the case of discrete I/O) will be stored in the input file. A 16-bit word, stored in the input file, is used to represent each one of the input modules present in the I/O rack. Each terminal, on a given module, is represented by a single bit in that 16-bit word. Bits 0 through 15 in the stored word correspond to terminals 0 through 15 on the input module. The values appearing at these terminals (high or low for discrete modules) will be stored to the corresponding memory location in the “I” (input) File.

An output address follows the same format as an input address. It consists of 3 alpha numeric values and 2 delimiters. Data pertaining to output addresses are stored in the “O” File. A discrete output can be turned on or off by storing a high or low value to a memory location associated with a specific terminal in a specific I/O

rack slot location. As an example, if you wish to turn on a device connected to terminal 7 of the output module located in slot 2 of the I/O rack, you would store a 1 value to the address O:2/7. Once again, the "O" (output) File is organized into a data table of 16-bit words. Each terminal on the output module corresponds to a single bit in a given word. Each output module in the rack will correspond to a separate word in the data table contained in the file. Module terminals will present the state (high or low for discrete I/O) that has been stored to the corresponding "O" (output) File in memory.

In addition to addresses referencing physical terminals in the I/O rack, it is common to reference "Virtual Outputs" using this addressing scheme. A virtual output is an output that does not physically exist in the I/O rack. For instance, if 8 slots in the rack are populated with modules of various types, a virtual output can be referenced by specifying a slot number that is not present or in use. The address O:10/0 is an example. Since there is no slot 10 in the rack, there is no module associated with this address. This does not mean, however, that there is not a word "10" in the data table in the "O" (output) File. Virtual outputs can be used as internal memory locations to store status or data without referencing an actual terminal on an output module.

What other things can be referenced by this addressing scheme?

In addition to I/O addressing, data tables containing variables/values can also be referenced using addresses. There are 3 types of values that can be stored in the file based addressing system, Integers, floating point values, and single bit values. These values are stored in the "N" (integer) File, "F" (floating point) File and the "B" (binary/bit) File types. Each of these files are organized into a table of "words". Each word and/or bit in these tables can be referenced individually using this addressing scheme. We have obtained a solid basic knowledge of the general file organization that should serve to assist in understanding the File Based Addressing system as we move forward.

Now let's examine the addressing format for these 3 files, as well as examine some specialty files that store custom data structures used to hold information pertaining to timers, counters, and control elements. These addresses as well as the structure of the associated timer (T), counter (C) and control (R) files are a bit more involved than the straightforward I/O addressing we have examined so far.

Here is a quick review of the "I" (input) and "O" (output) addressing scheme.

Input and Output Addressing Format:

Input File Address I : e . s / b

Output File Address O : e . s / b

Where:

I or O	indicates the input or output file is to be used (file 0 or file 1 of the 256 available data files)
:	is the element delimiter
e	is the slot number of the I/O module
.	is the word delimiter
s	is the word number (only used when more than 16 inputs or outputs contained in a single slot)
/	is the bit delimiter
b	is the terminal number on the I/O module (0-15)

In addition to addressing physical I/O locations, virtual outputs can be addressed by specifying a slot number that is beyond the range of the number of slots that have defined modules located in the rack. As an example, the address O:10/0 would denote a virtual output whose data is stored in the 11th element in the output data file. This is a virtual output if no output module was defined to exist in slot 10 of the I/O rack.

How do you store/read variables using this addressing scheme?

In addition to I/O addressing, data files containing variables/values are also referenced using file-based addressing. The B3 bit (or binary) file is used to store single bit status values (0 or 1). The N7 integer file is used to store integer values, and the F8 float file is used to store floating point values. Variables can be stored to or read from these data file locations for use in PLC programs. Each of these 3 default file types (B3, N7, and F8) can employ files 9 through 255 should additional storage of a given type be required. For example, you can use B22, N37, and F145 as user specified data files should your application demand additional storage for these variable types. Remember, only the first 9 files of the available 256 data files are pre-defined and considered to be default. The remaining files can be specified to be any of the available file types.

The B3 Bit File:

The B3 file is organised into a table containing 256 elements, each of which is 16 bits in length giving the user the ability to reference a total of 4096 bits per data file of the "B" type. Each bit in an element (16-bit word) is accessible using the file based addressing scheme. This file is ideal for setting and monitoring user defined status flags when desired application conditions exist. There are two ways in which the status of a given bit in this file can be specified. You can reference the element (word 0 - 255) in the table and then specify the bit within that element (bit 0 – 15). Alternately, you can also leave the element designation out of the address, and directly specify the bit number (bit 0 – 4096) in the data file table.

B3 Bit File Addressing Format:

Using Elements B3 : e / b

Where:

B3	indicates the bit file is to be used (B3 is the default, B9 to B255 can be used if additional storage is required)
:	is the element delimiter
e	is the element number (16 bit word). The value can range from 0 to 255
/	is the bit delimiter
b	is the bit number within the element specified. The value can range from 0-15.

Using only bits B3 / b

Where:

B3	indicates the bit file is to be used (B3 is the default, B9 to B255 can be used if additional storage is required)
/	is the bit delimiter
b	is the bit number within the entire data file table (can range from 0-4096)

In summary, a status bit in the B3 data file table can be referenced in either one of two ways. If we want to address bit 3 in the 2nd element in the B3 data file, we can specify this memory location as B3:1/3 or we can alternately specify this memory location as B3/19. For ease of use, it is generally recommended that you use the element designation format as it allows for an additional layer of categorization when organizing status flag bits.

The N7 Integer File:

The N7 integer file is also organized into a table of 256 elements, each of 16 bit length. This file table allows for specifying elements as well as bits within an element, however, values stored to or read from this data file are typically values as opposed to single bit states. The addressing format is similar to the one used with the B3 file addressing that includes the element number.

N7 Integer File Addressing Format:

Integer File Addressing N7 : e / b

Where:

N7	indicates the integer file is to be used (N7 is the default, N9 to N255 can be used if additional storage is required)
:	is the element delimiter
e	is the element number (16 bit word). The value can range from 0 to 255
/	is the bit delimiter
b	is the bit number within the entire data file table (can range from 0-15)

As an example, to store or read a value from the 4th element in this data file table, you would use the address N7:3 to identify the specific memory location for that integer value. Again, in general usage, you would not often have the need to specify a particular bit within the stored integer value.

The F8 Float File:

The F8 file is used to store or read floating point variables/values in memory. This file organization differs slightly from the ones described above for the B3 and N7 data files. Each file of this type contains 256 elements, however, in this case, each element is comprised of 2 words each being 16 bits in length. This means that consequently, this file type can be 2 times the size of a “B” or “N” type file. Another difference with this file type is that individual bits within elements cannot be individually referenced. Only elements inside this file type can be specified. This type of data file is ideal for storing and retrieving values used with math instructions and calculations. The basic format for the file based addressing scheme associated with this file type is similar to the ones previously outlined with the above noted exceptions.

F8 Float File Addressing Format:

Float File Addressing F8 : e

Where:

F8	indicates the integer file is to be used (N7 is the default, N9 to N255 can be used if additional storage is required)
:	is the element delimiter
e	is the element number. Each element consists of two 16 bit words. The value can range from 0 to 255

Floating point information can only be used by instructions that accept the REAL data type. Not all basic instructions can be used with floating point values stored in this data file table.

How do you reference information contained in data files specifically intended for use with instruction set items?

So far, we have covered addressing schemes that deal with data files that store information regarding physical I/O as well as variables that are used with PLC program. In addition to these types of data files, we will now take a look at some of the data file types used with the instructions that are available in the instruction set for the 500 series of AB controllers. There are 3 data file types which are intended for use with specific instructions. The T4 data file is used for storing timer variables and status bits. The C5 data file is used to store values relating to counter variables and status bits, and lastly the R6 data file is used to store information and status bits for "control items. (this last category would include sequencers, the FIFO and LIFO data stack instructions, as well as bit shift instructions among others. Each of these 3 data file types consists of 256 elements, each of which is comprised of three 16 bit words making these the largest of the data file types. We will move on now, to take a brief look at the addressing format for each of these data file types.

The T4 Timer File

This file consists of 256 elements used to track the preset and accumulated values as well as the state of various status bits associated with timer operation. Each element in this file contains three 16-bit words. Word 0 in an element is referred to as the control word. It contains the status bits associated with the timer instruction (i.e. EN, TT, DN bits). Word 1 in an element is used to store the desired PRESET value for the timer. The last word in each element, Word 2 contains the ACCUMULATED value. (Note, the timers "time base" value is not addressable and cannot be specified using the file based addressing scheme).

T4 Timer File Addressing Format:

Timer Addressing T4 : e .s / b

Where:

T4	indicates the timer file is to be used (T4 is the default, T9 to T255 can be used if required)
:	is the element delimiter
e	is the element number (each element contains three 16 bit words)
.	is the word delimiter
s	is the word number (word 0 is the control word, word 1 is the preset value, word 2 is the accumulated value)
/	is the bit delimiter
b	is the bit number in the specified word (0-15)

Addressable Bits in the Control Word (word 0) of a Timing Element:

When addressing word 0, you can leave out the word designation and simply reference the bit number for the desired status bit. It is also significant to note that you can use letters to denote specific bits in the control word. (ie EN, TT, and DN. These can be used to replace the numeric values denoting specific bits in the address. This

means there are two ways to reference items in the control word (word 0) of a timer element. In the examples and explanation below, we are assuming that we are addressing the 6th element in the timer data file. (Timer T4:5 would be the general reference for this timer in a program)

Bit 13 Done (DN)	T4:5/13 or T4:5/DN
Bit 14 Timer Timing (TT)	T4:5/14 or T4:5/TT
Bit 15 Enable (EN)	T4:5/15 or T4:5/EN

Addressable Words in a Timing Element (Word 1 and Word 2)

Word 1 of a timer element holds the preset value. This value can be referenced in one of two ways. You can include the word number, 1, or you can use the letters PRE to denote the storage location of the preset value, both of which are interpreted as equivalent references to word 1 of the timer element. This same approach can be used to designate word 2 in the element. This word contains the accumulated value. You can specify word 2, or you can use the letters ACC to denote the location of the accumulated value in the element.

Preset Value (PRE)	T4:5.1 or T4:5.PRE
Accumulated Value (ACC)	T4:5.2 or T4:5.ACC

In either instance of the above, (for both PRE and ACC addresses) you can add a designation specifying the particular bit inside either of these words in the element. For example, if we want to reference the least significant bit (bit 0) of the accumulated value of the 6th timer element in the table, we can do so as follows:

Accumulated Value, Bit 0	T4:5.2/0 or T4:5.ACC/0

In summary, the status as well as preset and accumulated values can all be referenced down to the single bit level. The timer elements each contain three 16-bit words that are addressable using the file based addressing scheme. Individual bits inside each of these words can be referenced as well as then entire word in the case of the Preset and Accumulated values (word 1 and 2 in the element). In addition, there are several standard letter combinations that can be used in place of specific bit or word numbers. These letters provide easily identifiable references, and it is recommended that they be used in place of the more obscure number only approach.

The C5 Counter File

This file consists of 256 elements used to track the preset and accumulated values as well as the state of various status bits associated with counter operation. The address formatting for this data file is very similar to that employed for timers. The first word in the element, word 0 is the control word that contains the status bits associated with the counter function. The second word, word 1 is used to hold the preset value for the counter, and the last word in the element, word 2, is used to hold the accumulated value. As with the timer addressing scheme, certain letters can be used in place of bit or word numbers. The counter control word has several more addressable bits than the timer control word.

C5 Counter File Addressing Format:

Counter Addressing C5 : e .s / b

Where:

C5	indicates the counter file is to be used (C5 is the default counter data file, C9 to C255 can be used if required)
:	is the element delimiter
e	is the element number (each element contains three 16 bit words)
.	is the word delimiter
s	is the word number (word 0 is the control word, word 1 is the preset value, word 2 is the accumulated value)
/	is the bit delimiter
b	is the bit number in the specified word (0-15)

Addressable Bits in the Control Word (Word 0) of a Counter Element

As with timer data files, counter data files use word 0 as the control word which contains the counter status bits. Here again, because we are addressing the first word in the element, the word designation part of the address can be omitted as it is not required. In addition, as with the timer data file, the counter data file addressing scheme recognizes specific letters that are used to denote bit numbers for particular status bits. In the examples and explanations below, we will be assuming that we are addressing the 3rd element (C5:2 would be the reference to this counter in a program) for the purpose of illustration.

Bit 11 Underflow (UN)	C5:2/11 or C5:2/UN
Bit 12 Overflow (OV)	C5:2/12 or C5:2/OV
Bit 13 Done (DN)	C5:2/13 or C5:2/DN
Bit 14 Count Down Enable (CD)	C5:2/14 or C5:2/CD
Bit 15 Count Up Enable (CU)	C5:2/15 or C5:2/CU

Addressable Words in Counter Element (Word 1 and Word 2)

Word 1 of a counter element holds the preset value. The same approach (using PRE instead of the word number) can be used with counters as was used with timers. This also applies to the use of ACC to denote word 2 of a counter element.

Preset Value (PRE)	C5:2.1 or C5:2.PRE
Accumulated Value (ACC)	C5:2.2 or C5:2.ACC

In either instance of the above, (for both PRE and ACC addresses) you can add a designation specifying the particular bit inside either of these words in the element. For example, if we want to reference the least significant bit (bit 0) of the accumulated value of the 3rd counter element in the table, we can do so as follows:

Accumulated Value, Bit 0	C5:2.2/0 or C5:2.ACC/0

In summary, the counter data file type contains 256 elements, each of which is made up of three 16-bit words. These elements and the subsequent addressing scheme follow the same format laid out for the timer (T4) data file type. The only significant difference pertains to the control word bits that are addressable, and the letters used to denote them.

The R6 Control Data File

This file consists of 256 elements used to track the status, length of a bit array or file, and position pointer within an array information relating to instructions that employ this file type. Unlike the T4 and C5 data file types, this file type is used with several types of instructions. For this reason, there are many more status bits that are addressable in the control word (word 0) of each element in this data file. Various instructions use specific combinations of these control status bits. Not all status bits in the control word are used in each instruction type that employs this file to store information pertaining to the instruction's operation.

R6 Control Data File Addressing Format:

Control Data Addressing R6: e .s / b

Where:

R6	indicates the control file is to be used (R6 is the default counter data file, R9 to R255 can be used if required)
:	is the element delimiter
e	is the element number (each element contains three 16 bit words)
.	is the word delimiter
s	is the word number (word 0 is the control word, word 1 is the length value, word 2 is the accumulated value)
/	is the bit delimiter
b	is the bit number in the specified word (0-15)

Addressable Bits in the Control Word (Word 0) of a Control Data Element

As with timer and counter data files, control data files use word 0 as the control word which contains the control data status bits. As with our other multi word elements, because we are addressing the first word in the element for the status bits, the word designation part of the address can be omitted as it is not required. In addition, as with the timer and counter data file, the control data file addressing scheme recognizes specific letters that are used to denote bit numbers for particular status bits. In the examples and explanations below, we will be assuming that we are addressing the 1st element for the purpose of illustration. (R6:0 would be the reference used with an instruction employing the Control Data file type)

Bit 8 Found (FD)	R6:0/8 or R6:0/FD
Bit 9 Inhibit (IN)	R6:0/9 or R6:0/IN
Bit 10 Unload (UL)	R6:0/10 or R6:0/UL
Bit 11 Error (ER)	R6:0/11 or R6:0/ER
Bit 12 Stack Empty (EM)	R6:0/12 or R6:0/EM
Bit 13 Done (DN)	R6:0/13 or R6:0/DN
Bit 14 Unload Enable (EU)	R6:0/14 or R6:0/EU
Bit 15 Enable (EN)	R6:0/15 or R6:0/EN

Addressable Words in Control Data Elements (Word 1 and Word 2)

Word 1 of a control data element holds the length of the bit array or file. Word 2 holds the position pointer location within that array. As with our previous multi word elements, we can reference these locations using the word number, (1 or 2) and also using LEN for length, and POS for position.

Length Value (LEN)	R6:0.1 or R6:0.LEN
Position Value (POS)	R6:0.2 or R6:0.POS

Once again, in either instance of the above, (for both LEN and POS addresses) you can add a designation specifying the particular bit inside either of these words in the element. For example, if we want to reference the least significant bit (bit 0) of the length value or the position value of the 1st control data element in the table, we can do so as follows:

Length Value, Bit 0	R6:0.1/0 or R6:0.LEN/0
Position Value, Bit 0	R6:0.2/0 or R6:0.POS/0

To summarize, Control Data files are used with several instructions. This data file type uses elements consisting of three 16-bit words. As with the timer and counter data files, the first word, word 0, is used to store status bits used for the specific instructions that are employing the data file

Wrapping Up the Basics of File Based Addressing

The file based addressing scheme can be used to store, reference, and retrieve information pertaining to physical I/O locations, variables and values, as well as parameters relating to specific instruction set items used in PLC programs. The data file section of memory consists of 256 files. The first 9 files are of pre-determined types. The balance of the data files available (files 9 through 255) can be defined and used as any of the allowable data file types. Each data file type contains 256 elements that can be addressed individually. Depending on the data file type, and element can consist of 1, 2, or 3 distinct 16-bit words. In most cases, each word, as well as each bit in the word can be selectively addressed depending on the file type, and/or instruction it is being used with. There is a general consistency employed throughout the various data file types used by the file based addressing scheme and most aspects are common among the various data file types. We have examined 8 of the 9 default data files provided with the 500 series AB controllers. The data files that employ three-word elements all allow for pneumonic referencing in lui of specifying bit or word numbers. The letters used for these are specific to the element and function they are being used with.

5 TIPS ON HOW TO REDUCE SCAN TIME USING LADDER LOGIC

Let's examine scan time and its impact on everyday PLC applications as well as examine how it can be reduced. Scan time is an important metric to be considered in many high-speed PLC applications. The time it takes to perform a single scan cycle can have significant impact on the input stimulus and/or output control signals present or required for high-speed applications. Specialty I/O modules are often employed in these situations and offer significant advantages.

That said, it is important to note that scan cycle time can also be of significance for general applications as well. Although they may not have the extreme constraints and dependencies of high-speed applications, general applications, such as a simple multiplexing program segment, can have practical limitations that can relate directly to scan time. The impact scan time has on the operation of timers can be significant. It is generally accepted that an understanding of the impact of scan time along with some basic knowledge of how it can be reduced is important when learning how to program a PLC with ladder logic.

We will examine a simple example of each of the following 5 tips over the next two blogs:

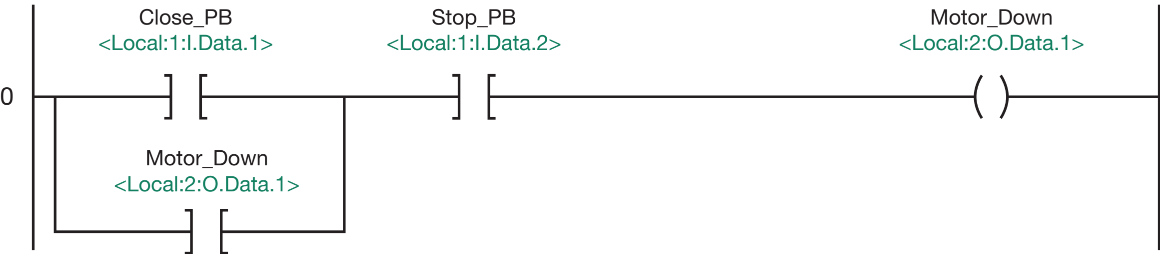
1. Place instructions/conditions that are most likely to be false at the start of a rung to reduce the number of instructions seen during the scan.
2. Avoid duplicating unique tag/instruction combinations when creating ladder logic programs whenever possible. A change in architecture can often reduce the total number of instructions used in a program reducing memory usage as well as scan time.
3. Program flow control can be key to significant reductions in scan time. Use the JMP and LBL instructions to reduce the active segments of a running program.
4. Compartmentalizing tasks (making modular processes) and organizing them can have significant impact on scan time. Passing variables can allow for program segments (subroutines) to be used in multiple instances.
5. Avoid floating point arithmetic and try to use integers wherever possible. If you need better than integer precision, consider multiplying all your floats by 10, 100, or 1000 to get integers.

A Basic Example of Each Scan Cycle Time Reduction Tip:

Let's review a simple example of each the first two tips.

1. The instruction most likely to be false should be at the start of a rung.

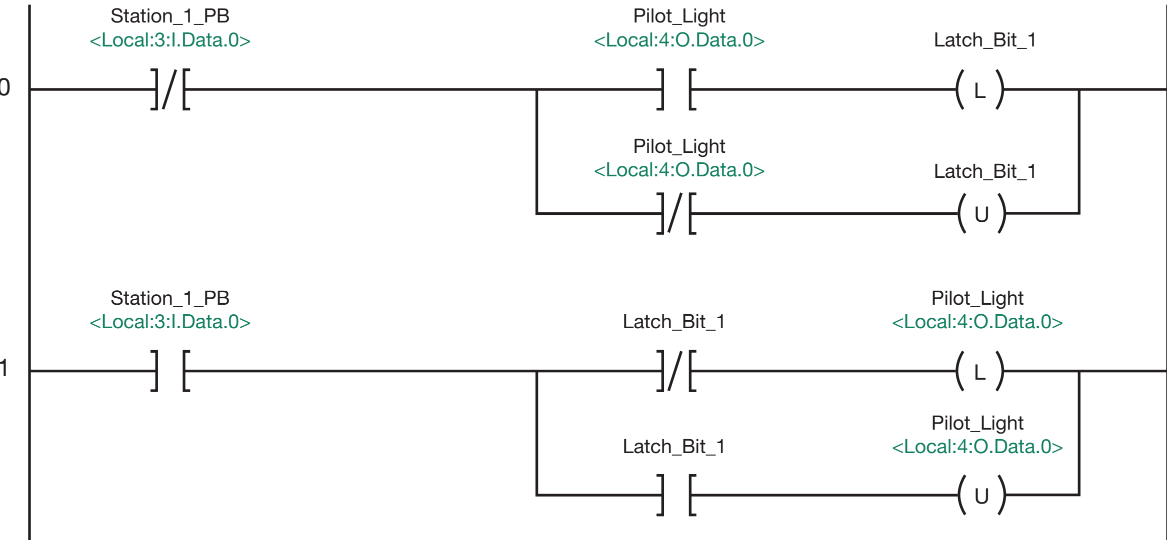
When possible, arranging instructions from left to right with the likelihood of them being false determining their position is a good practice that should be adopted. Consider the simple motor seal-in rung below:



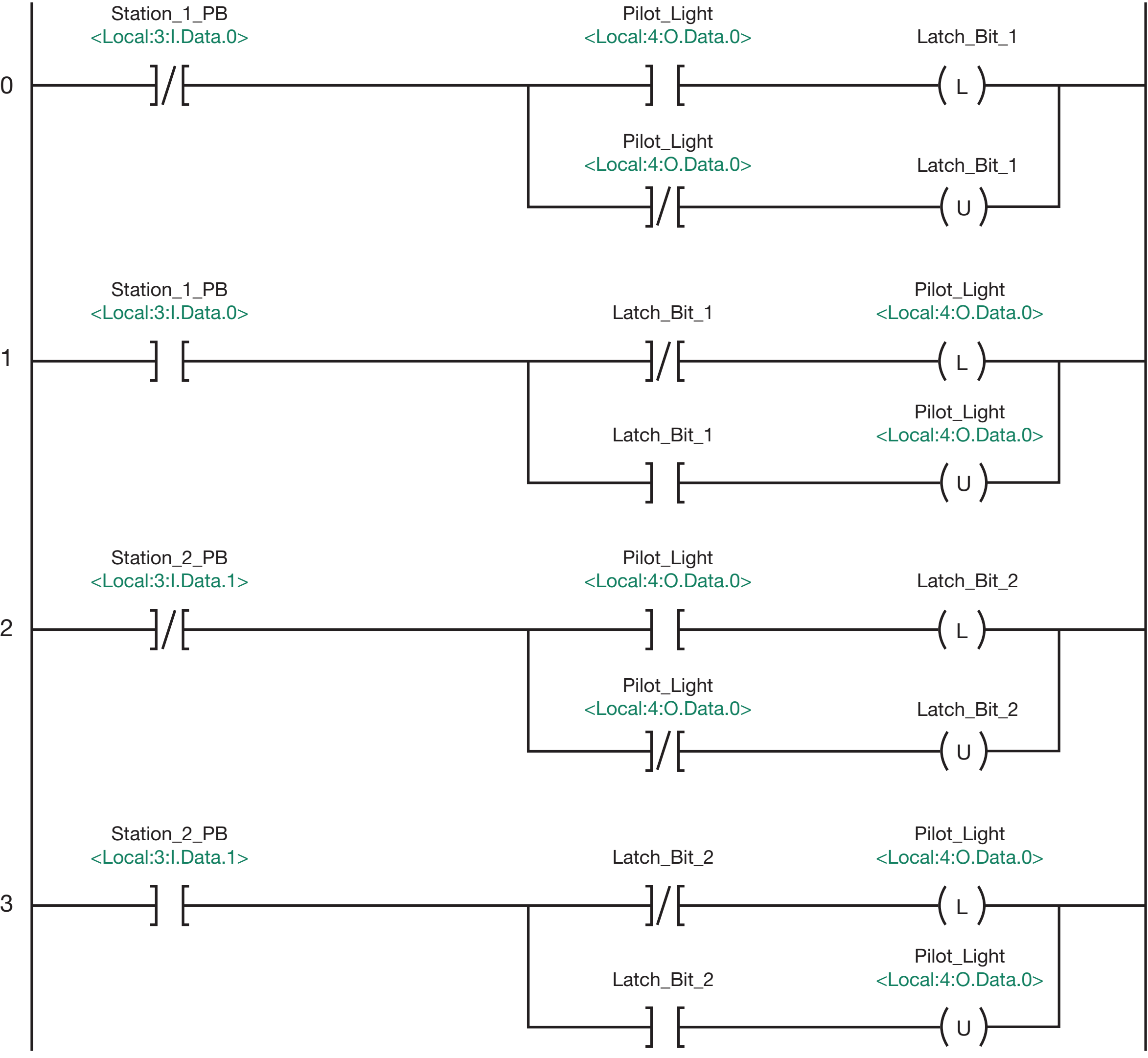
The “Stop PB” pushbutton field device is a normally closed momentary contact switch. When the application is running, the normally open contact associated with it closes due to the state of the pushbutton in the field. For this reason, the branch part of the seal in, should appear to the left of the “Stop PB” pushbutton contact on the rung. The Stop PB contact will be closed most of the time during program execution. It should not be the first instruction scanned on the rung. This small effect can have a significant impact on scan cycle time cumulatively.

2. Avoiding the duplication of unique tag/instruction combinations.

Consider this program segment that is used to provide a single start/stop station using a momentary contact pushbutton.

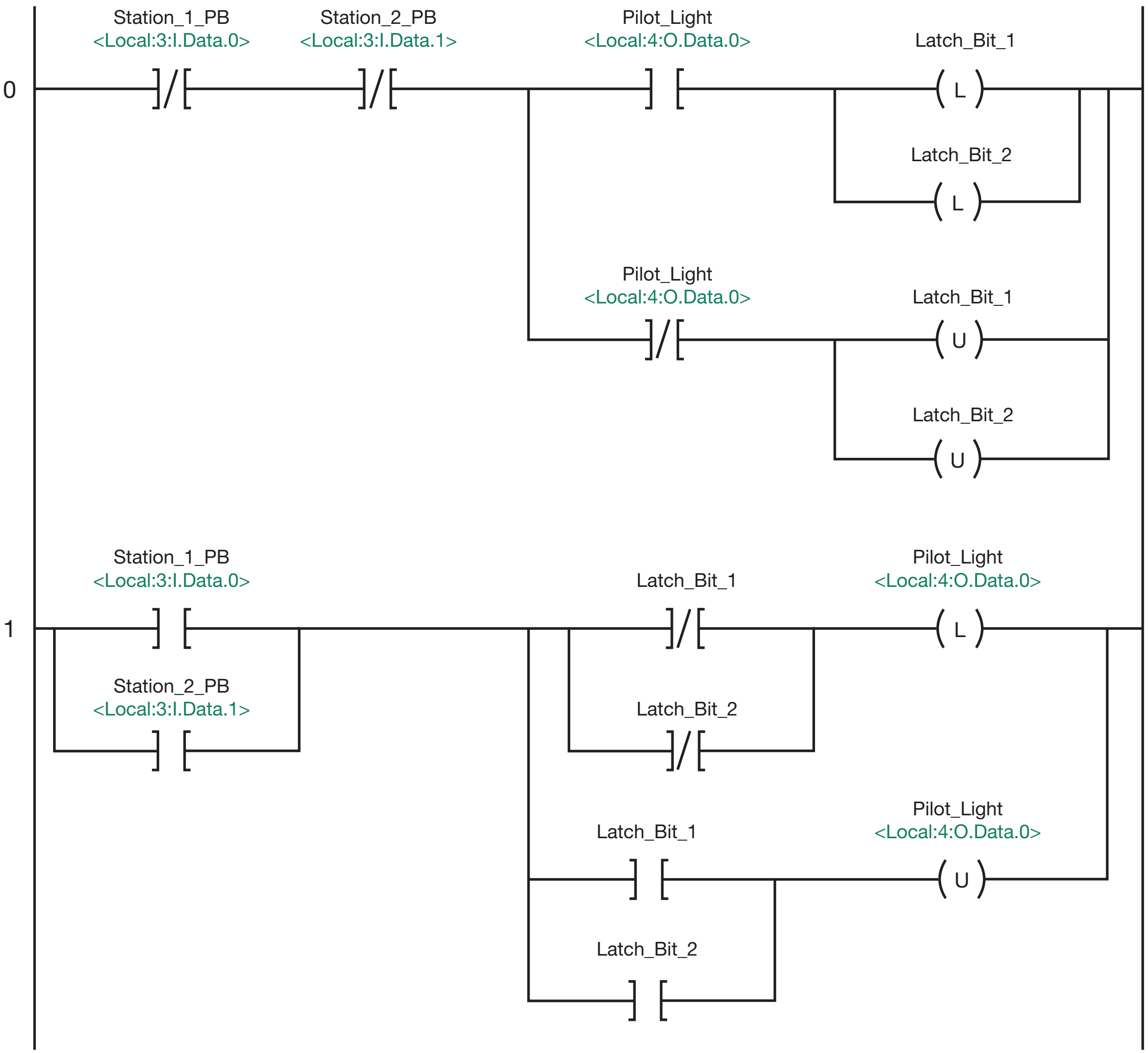


The program segment above is optimized and does not use any unique tag/instruction combinations more than once. If the application called for the use of a second station, one could copy this structure and modify the tags to create the program segment shown below:



Although this was an easy way to accomplish our objective, we have introduced a duplication of 4 tag/instruction combinations. The XIO, XIC, OTL and OTU instructions are appearing twice with the “Pilot_Light” alias tag in the version above. The impact of this redundancy would increase for each additional station that may be required. An alternate topology can be the way to resolve this redundancy issue.

The program segment shown below will function identically to the version having 2 stop/start stations we examined previously.



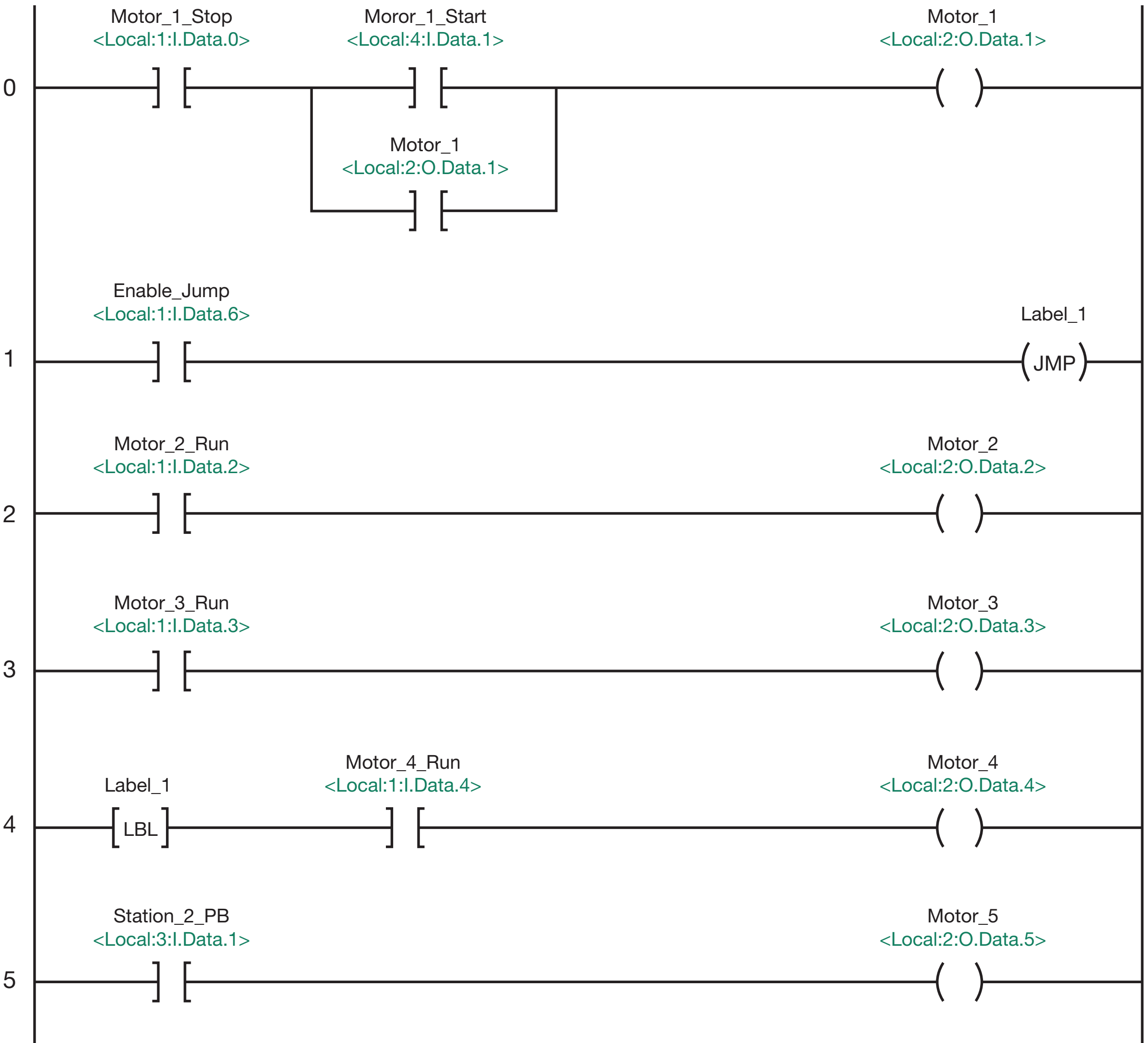
This version of the program segment uses 4 instructions less than the previous version. It contains no duplicate tag/instruction combinations and this new topology is easy to expand upon when additional stations may be required. It was not immediately apparent that there was another way to organize the required logic for this example, but time should be taken for the attempt as it is always worth the effort when successful and can have a significant impact on scan cycle time as well as memory usage.

3. Use the JMP and LBL instructions to reduce the volume of program code being executed during each scan cycle.

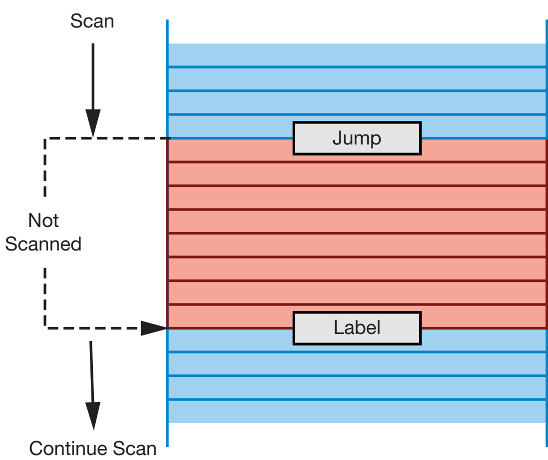
Program flow is an area in which significant reduction in scan time is possible.

In the example below, the motor 2 and motor 3 rungs are skipped over when the “Enable_Jump” contact is closed on rung 1. At that time, the JMP instruction is encountered and program execution jumps to the

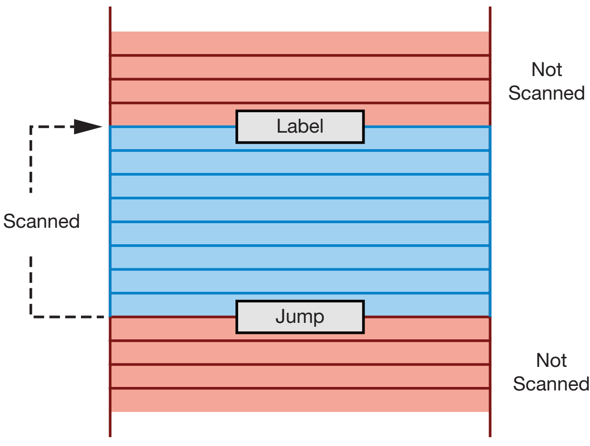
"Label_1" LBL instruction location on rung 4 skipping over rungs 2 and 3. This concept can be used to jump over large portions of a program that are not being used at any given time in a production cycle. This has the potential to greatly reduce the number of rungs being scanned and subsequently can significantly reduce an applications scan cycle time.



The ability to skip over rungs is not the only way that the JMP and LBL instructions can be utilized to reduce scan cycle time. Suppose you are monitoring a series of inputs for specific changes in state, and that the nature of these state changes will determine what series of actions will consequently occur. In a situation such as this, creating a program loop to execute only the rungs needed to monitor the inputs will ensure that scan cycle time is not a deterrent to providing rapid responses to any input state changes.



Skipping over sections



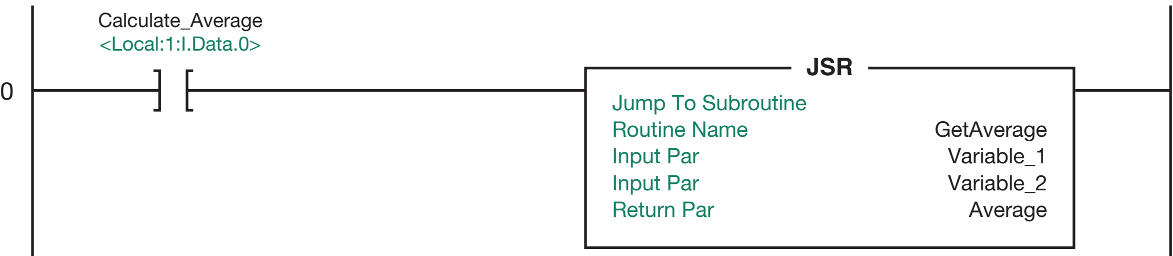
Creating program loops

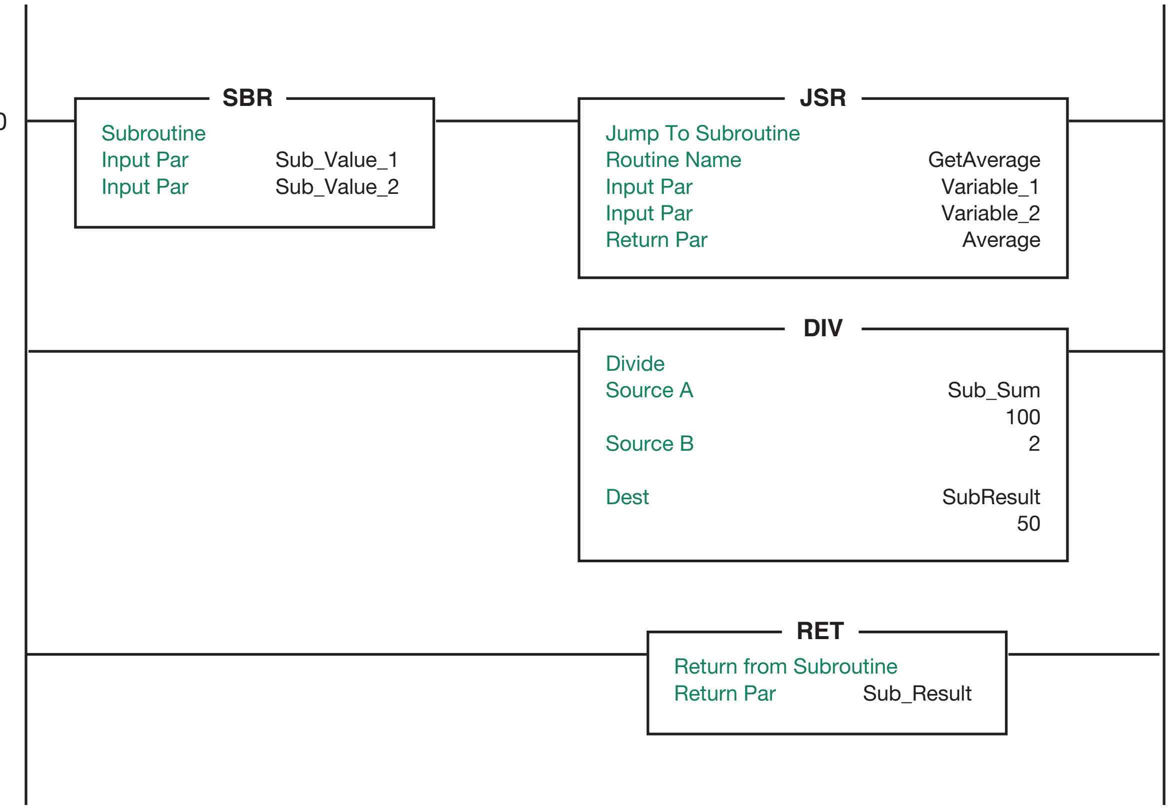
This is often accomplished by creating a "zone" that is defined by the location of the JMP and LBL instructions in your program. This zone is scanned from top to bottom and then program execution loops back to the start of the defined zone again. This will continue until an exit condition allows program execution to resume outside the loop. It is obvious that these techniques can have significant impact on the scan cycle and that program flow management using the JMP and LBL instructions can be a powerful tool providing a degree of control over scan cycle times.

4. Compartmentalizing tasks using subroutines to allow for program segments to be used in multiple instances.

In keeping with the concept that scanning less rungs will reduce overall scan cycle time, the use of the JSR, SBR and RET program flow instructions and their impact on scan cycle times always merits consideration. Creating reusable atomic program segments for functions that get repeated at different times in a process allows these program segments to get called only as needed and will significantly reduce the number of rungs being scanned as this avoids needless duplication. As a simple example, consider a routine that requires the average

of two values in several calculations being made. Creating a subroutine that can be called in more than one circumstance avoids the need to duplicate any of the instructions needed to find an average value. Values are passed into the routine and a result is passed out as shown below.



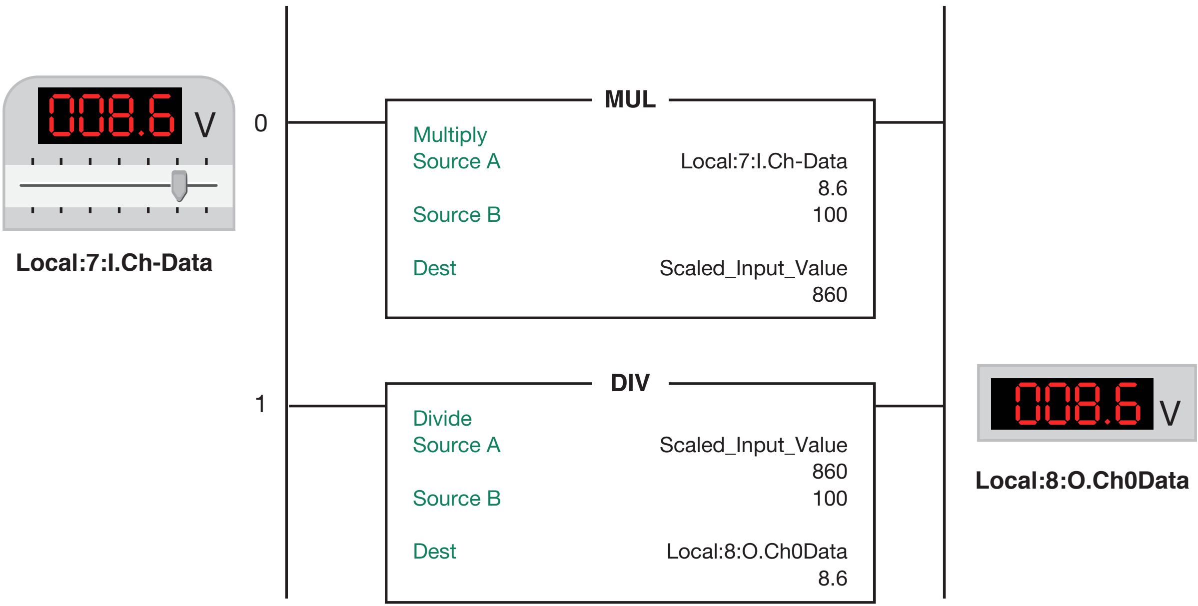


Mainline Program Subroutine Call and Subroutine

The JMP and LBL instructions allow you to focus on a segment or skip over segments. The JSR and RET subroutine instructions allow program segments to be used multiple times instead of creating multiple instances of them. The program flow instructions are great tools for managing scan cycle times.

5. Avoid floating point arithmetic and try to use integers wherever possible.

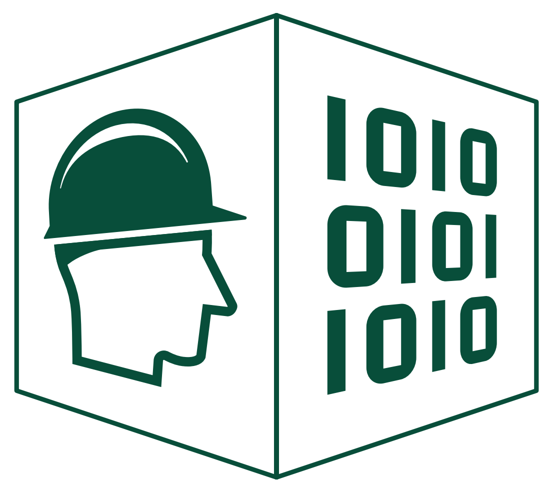
This general practice shortens scan cycle time as the instructions that use the REAL data type take up more memory, use more resources to function and require more cycles than the INT data type variables. As an example, suppose we have an analog input signal ranging from 0 to 10 volts and the sensor is providing 2 decimal places of accuracy. A value such as 8.34 V would be an example of the type of input coming to the analog input channel. Multiplying the value by 100 will remove the decimal portion of the value and create a whole number that can be used with the INT data type. It can be said that in general it is beneficial to work with INT or DINT data types when doing math operations by scaling values at the input of output point of a process. An analog output control signal should be generated using INT or DINT data types and converted to a REAL value, within output range, just prior to being sent out when at all possible.



This program segment takes an analog input value between 0.00 and 10.00 and converts it to a whole number value ranging from 0 to 100. This scaled value can be operated on using DINT data type variables at that point. The value is being scaled back to the 0 to 10 range before being sent to an output module location.

Scan cycle time is an application parameter that can have varying significance to a process. In instances where scan time is critical, knowing ways to reduce its impact on an application can be extremely helpful. Following these 5 tips on reducing scan cycle time and they will help you to manage this parameter in cases where it may be critical to normal operation.

2 PROGRAM EXAMPLES

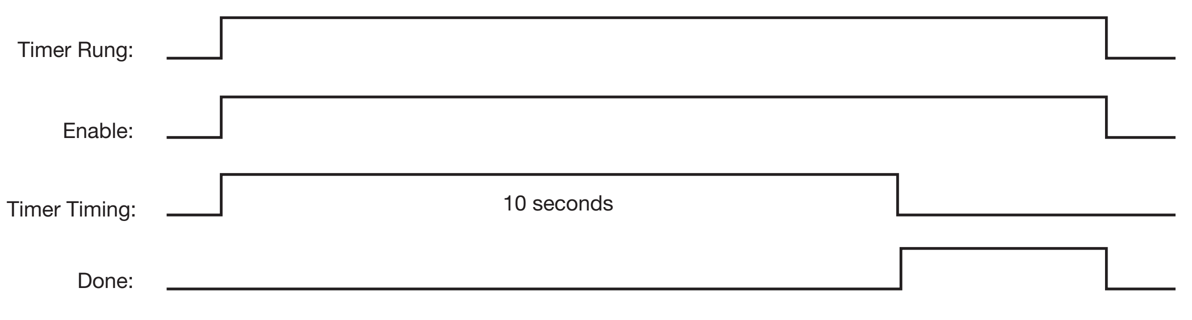


PLC TIMER INSTRUCTIONS - THE FOUR TYPES OF TIMED RELAYS THEY EMULATE

PLC timer instructions emulate the behaviour of "Time Delay Relays" and their associated contacts. The two basic types of timer instructions used to achieve the desired behaviours are the TON on delay timer and the TOF off delay timer.

All timer instructions share a common structure and use the "TIMER" datatype. The datatype has storage for timer "Preset" and "Accumulated" values as well as the three significant status bits used with this kind of instruction. (EN enable, TT timer timing, and DN done). The behaviour of these status bits is key to understanding how the behaviour of time delay relays has been replicated using these simple instructions with normally open and normally closed contacts. We can begin with a quick review of the operation of the TON and TOF delay timer instructions with respect to their status bit behaviour.

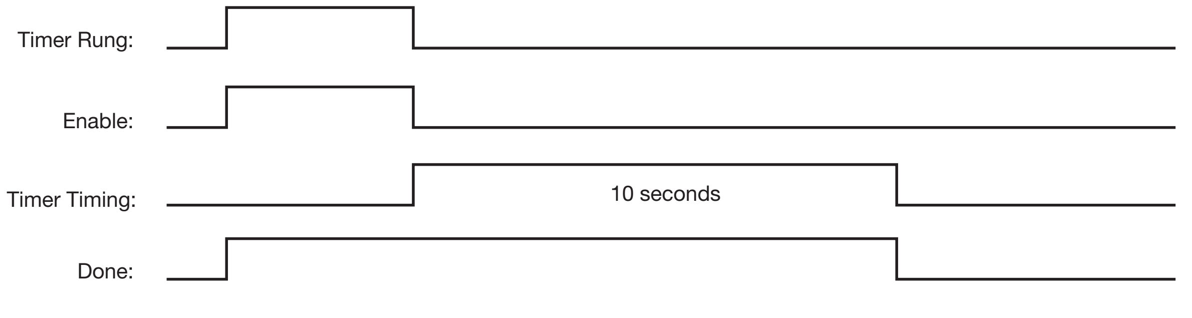
Timing Diagrams and general behaviour of the TON vs TOF timer instructions



TON On Delay Timer

Here we have the timing diagram associated with a TON delay timer instruction. When the timer rung goes high in the figure above, the TON timers EN bit goes high. It will remain high as long as the rung condition is true. In addition to the EN bit going high, the TT, (timer timing) bit also goes high and the timer begins timing out its "preset" time duration (10 seconds in the example graph above). The TT bit remains high until the specified duration times out at which point the TT bit goes low and the DN (done) bit goes high. The DN bit will remain in this state until the rung condition for the timer transitions back to the false (low) state.

Below we have a similar timing diagram, this time, it illustrates the behaviour of the three significant status bits with the TOF delay timer instruction.



TOF Off Delay Timer

The EN (enable) bit for the TOF delay timer instruction behaves exactly like that of its counterpart. When the timer rung goes high, the EN bit goes high and remains high as long as the rung remains true (high). The DN bit is also enabled when the timing rung goes high. This is distinctly different behaviour than that of the TON version. When the timer rung transitions from high to low, (true to false) the TT bit goes high, and the timer begins timing out its preset time duration (10 seconds in the above example). The DN bit remains high during this period. When the specified period times out, the TT bit goes low, and the DN bit also goes low returning the timer status bits to their rest state.

Now that we have reviewed the basic operation of the status bits and the TON and TOF timing diagrams, lets take a look at how they are employed to replace timed relay contacts used with older hardwired control systems.

Listed below are the four timed relay contact behaviours performed by these two timer types. The timer type used to perform each of these functions is also provided:

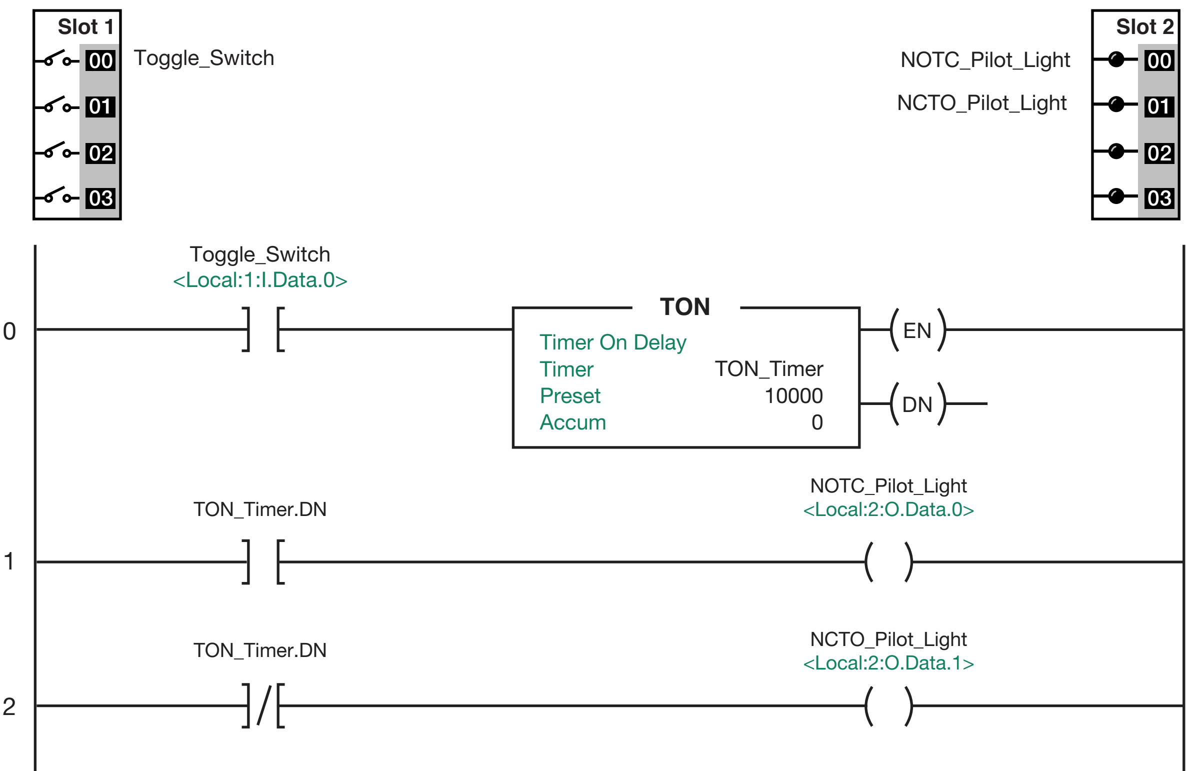
Use the TON on delay timer done (DN) status bit to reproduce the behaviour of:

- **a)** NOTC - Normally Open Timed Closed Time Delay Relay Contacts
- **b)** NCTO - Normally Closed Timed Open Time Delay Relay Contacts

Use the TOF off delay timer done (DN) status bit to reproduce the behaviour of:

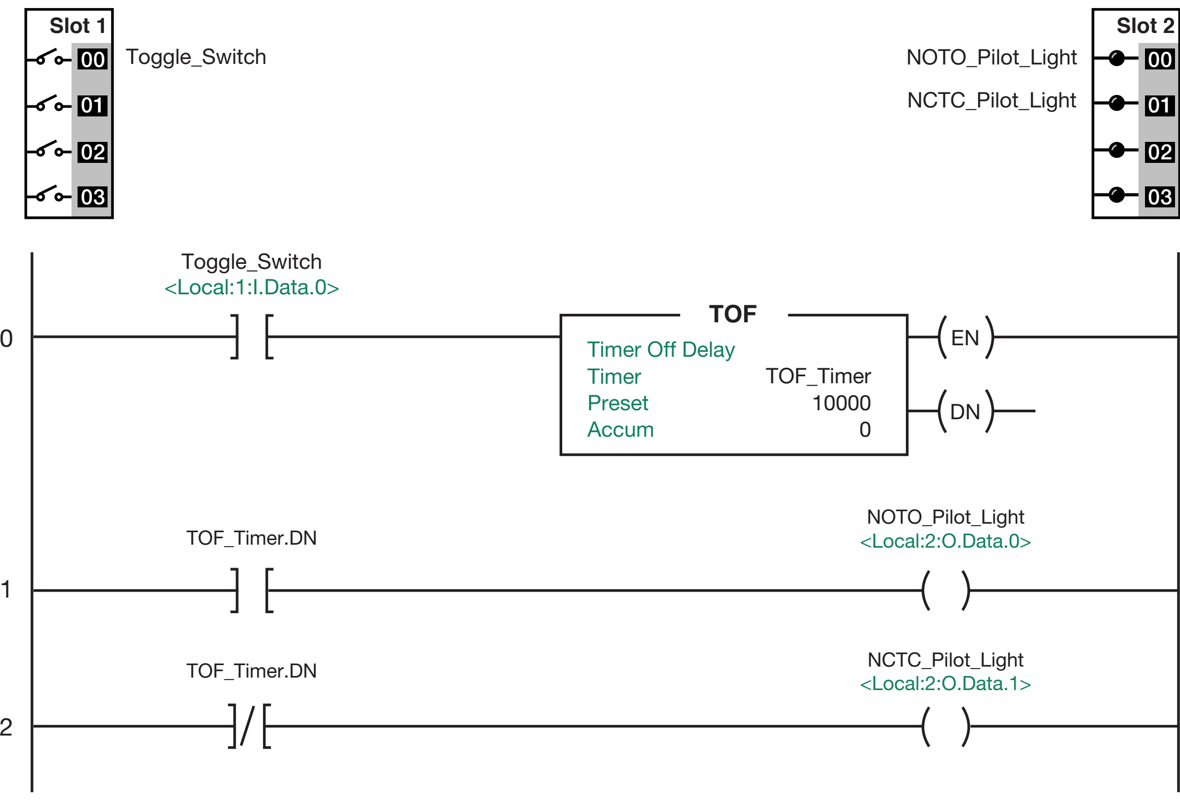
- **c)** NOTO Normally Open Timed Open Time Delay Relay Contacts
- **d)** NCTC Normally Closed Timed Closed Time Delay Relay Contacts

NOTC and NCTO timed relay behaviour using the TON delay timer:



The above figure is a ladder logic program that illustrates the use of the TON delay timer to provide NOTC and NCTO time delay relay contact behaviour. The two outputs in the program segment will behave as if time delay relays are being used to enable them. Although we are not discussing the value fields used with these instructions in this article, it should be noted that for a TON on delay timer instruction, the accumulated value is reset to zero if the timer rung transitions to false at any time. The preset value remains unchanged.

NOTO and NCTC timed relay behaviour using the TOF delay timer:

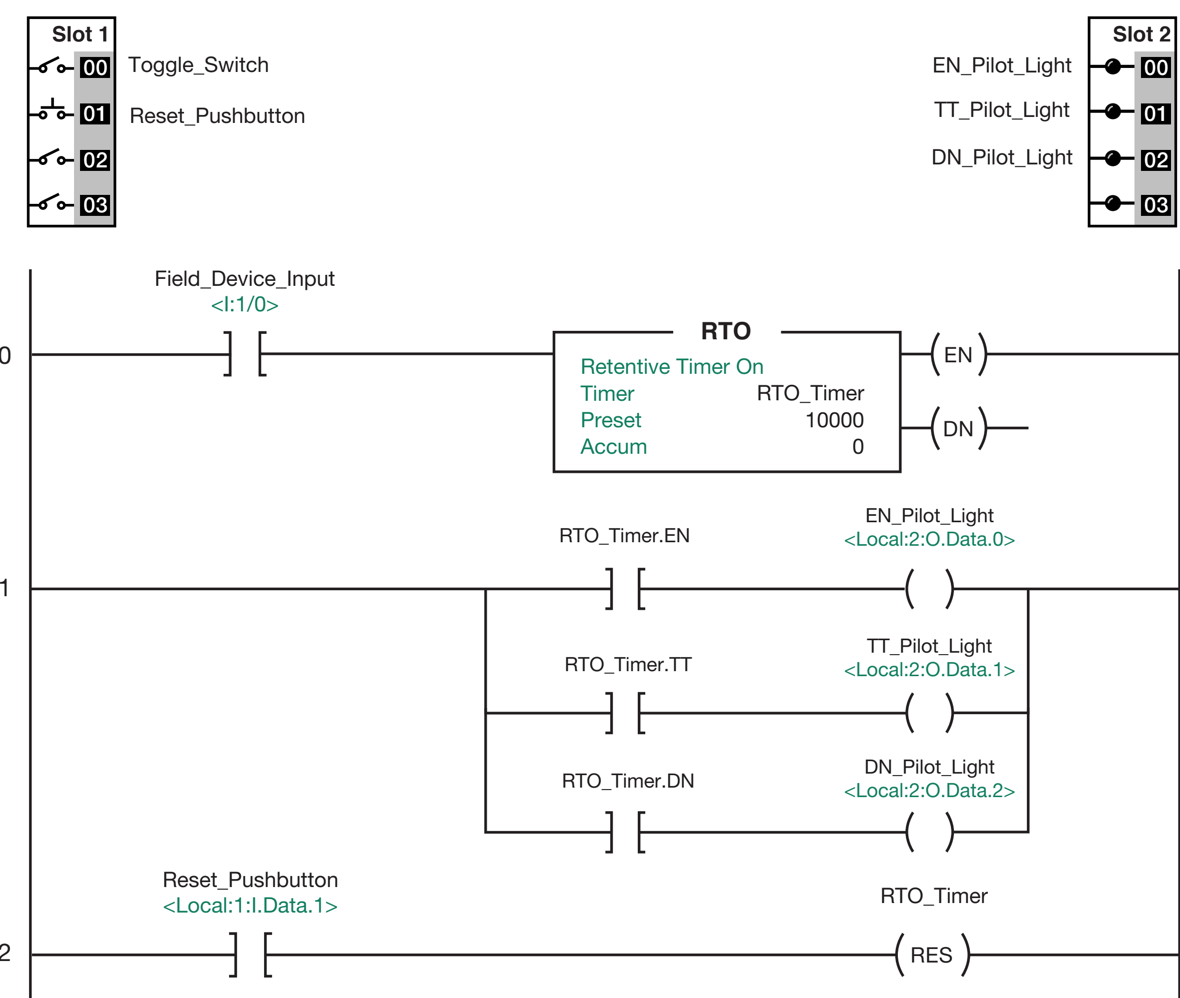


The TOF timer instruction is used to emulate the behaviour of the NOTO and NCTC type of timed relays. Initially, the output on rung 1 is low and the output on rung 2 is high. (Rung 1 Normally Open, Rung 2 Normally Closed). When the timer rung goes high, rung 1 goes high and rung two goes low. When the timer times out, the output on rung 1 returns to its at rest low state for a TO or “timed open” finish, and the output on rung 2 goes back to its initial high state providing the TC or “timed closed” portion of the behaviour.

While the behaviour of the TON delay timer instruction illustrated in the previous section is somewhat intuitive, the TOF timer is often the source of confusion when considering the NOTO and NCTC behaviours that this type of timer instruction emulates.

The Retentive Timer Instruction:

The TON delay timer instruction has a RETENTIVE variation. This timer instruction is refered to as a “Retentive Timer” instruction. It has a couple of special features that differentiate it from a standard TON delay time, however, its general operation and timing diagram matches that of a standard TON delay timer instruction. The simple program segment below would be used to illustrate the behaviour of this timer with respect to its status bits and the TON delay behaviour would be seen.



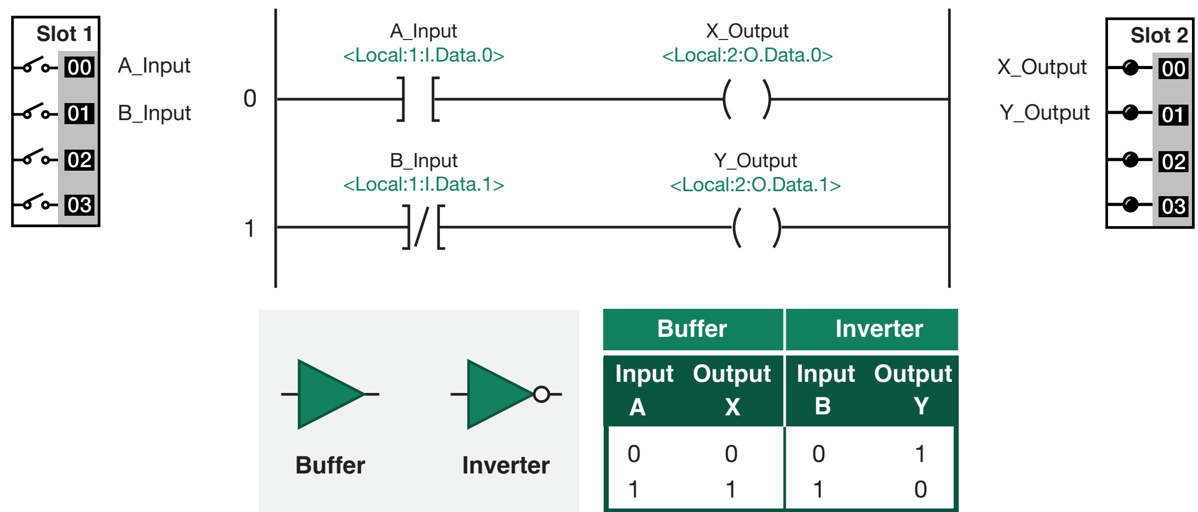
In the figure above, the toggle switch can be used to start and stop the timer. With a retentive timer, the accumulated value is kept when the timer rung transitions to false. The value will remain there until the timer is reset using the RES (reset) instruction as shown on rung 2 above. The general behaviour of the RTO timer is that of a TON delay timer. In addition to the different ACC value behaviour, the DN status bit for the RTO timer will remain set if it times out even if the timer rung transitions from true to false unlike that of the TON time which goes low immediately. The DN status bit will remain set until the RES instruction resets it along with the accumulated value.

DECISION MAKING - LADDER LOGIC EQUIVALENT RUNGS FOR THREE INPUT LOGIC GATES

Boolean logic gates are often used in decision making applications. In this article we will go over the rung configurations as well as the truth tables for the major logic gates used in decision making circuits/applications.

We will begin with the SINGLE INPUT buffer and inverter gates along with their truth tables. A truth table lists all possible input combinations with their corresponding output condition.

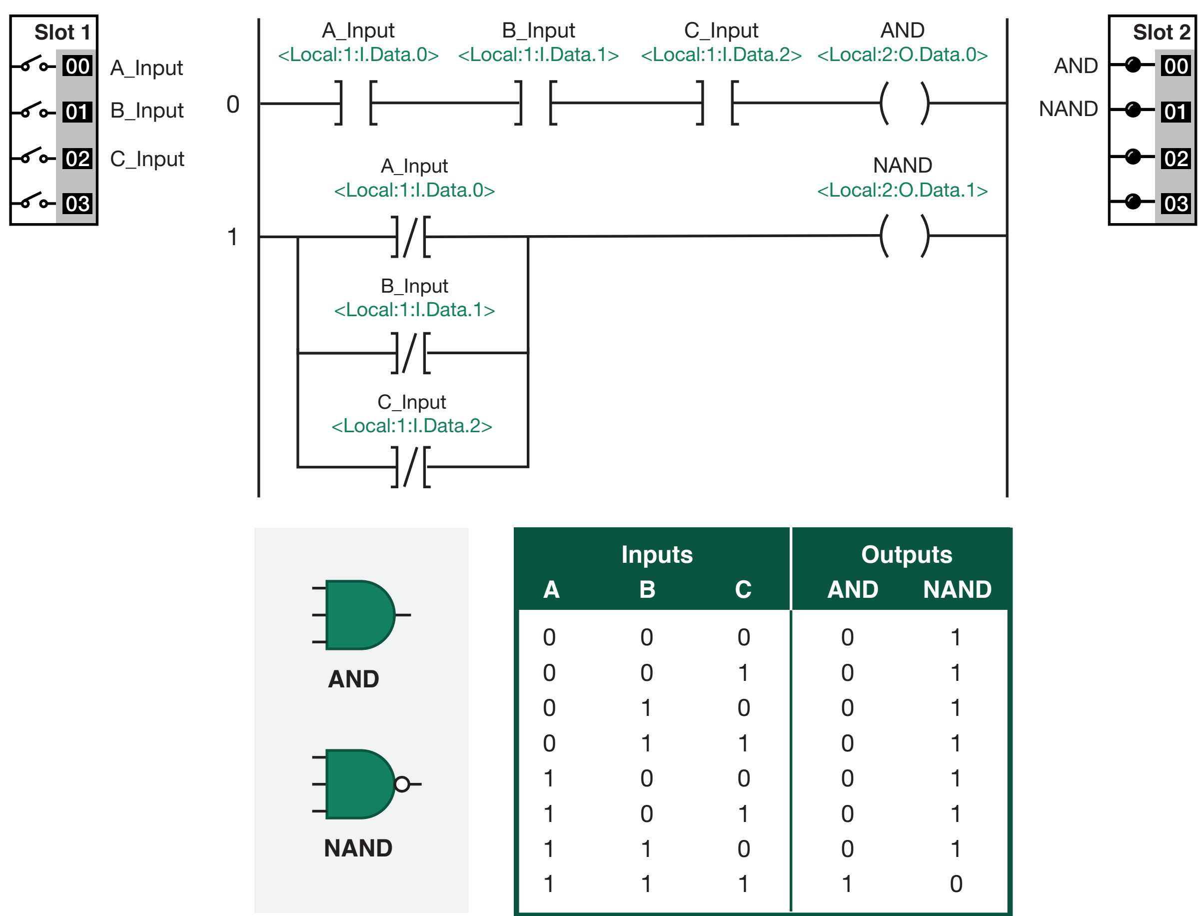
Buffer And Inverter Behaviour



Buffer		Inverter	
Input A	Output X	Input B	Output Y
0	0	0	1
1	1	1	0

In the figure above, rung 0 provides buffer logic. This is accomplished using an NO (Normally Open) contact with the normally open switch (field device). The inverter is located on Rung 1 and uses the NC (Normally Closed) contact with the normally open switch in the field.

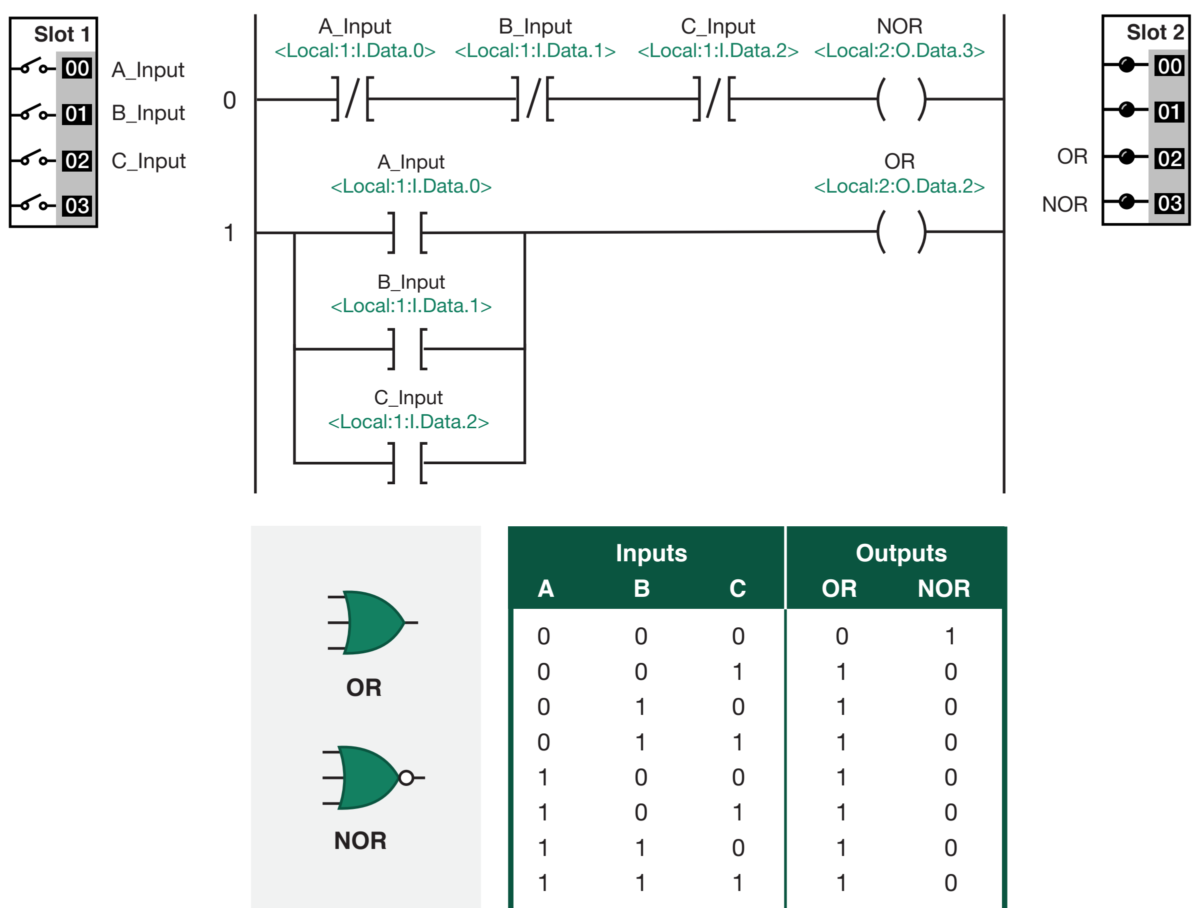
AND and NAND Gate Behaviour



Inputs			Outputs	
A	B	C	AND	NAND
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Here we have the 3 INPUT AND as well as the NAND logic gates along with their corresponding truth table. Each input condition is presented on the input switches and the output state is recorded in the truth table. In this way, the complete behaviour of these two decision making rungs can be easily illustrated and conveyed.

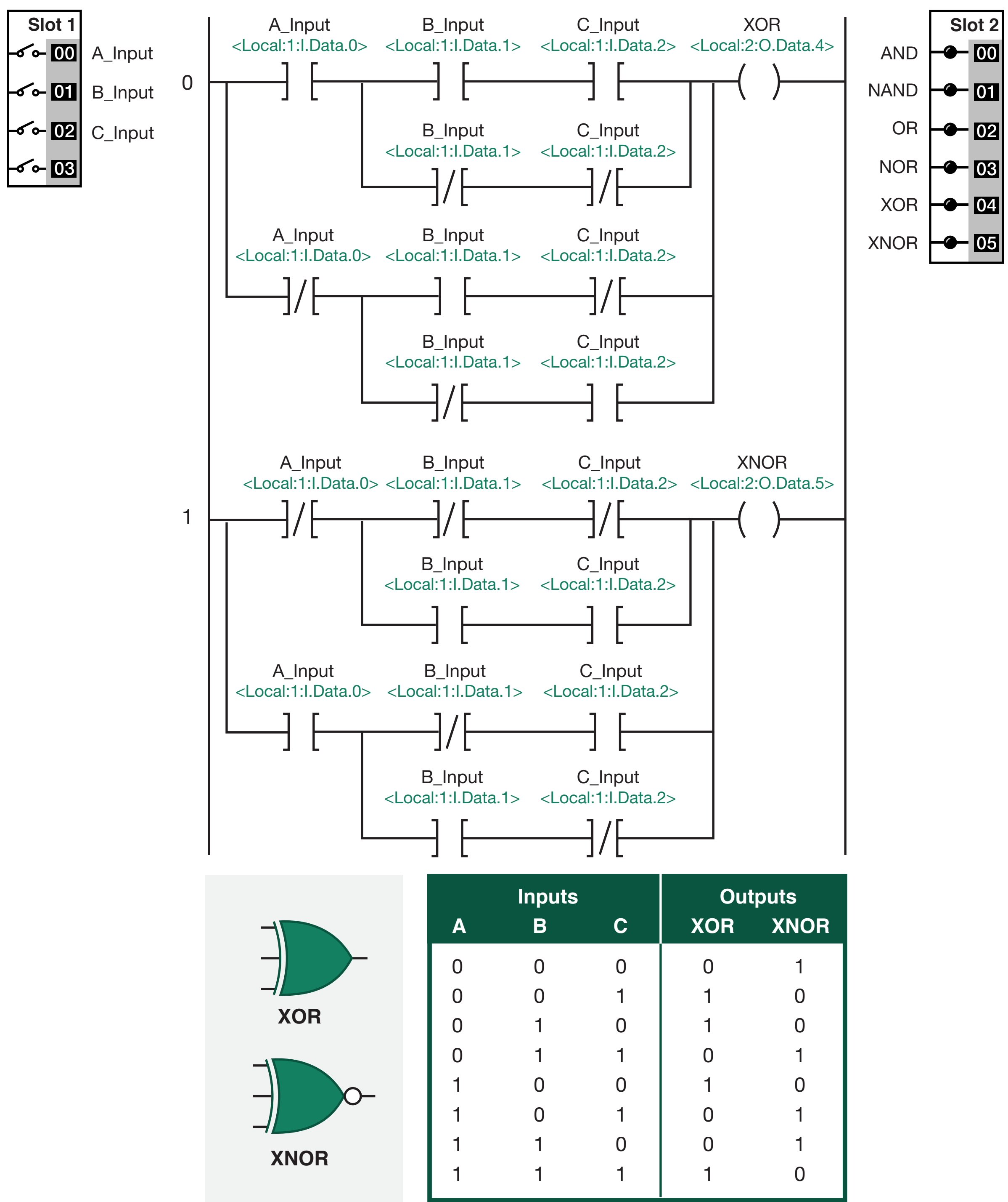
OR and NOR Gate Behaviour



Inputs			Outputs	
A	B	C	OR	NOR
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Rung 0 in the above figure provides the 3 INPUT NOR logic function. The NOR output will go LOW if any one of the 3 input switches are closed. Effectively, neither Input A, NOR Input B, NOR Input C can be closed if you want the output to be HIGH.

XOR and XNOR Gate Behaviour



Inputs			Outputs	
A	**B**	**C**	**XOR**	**XNOR**
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

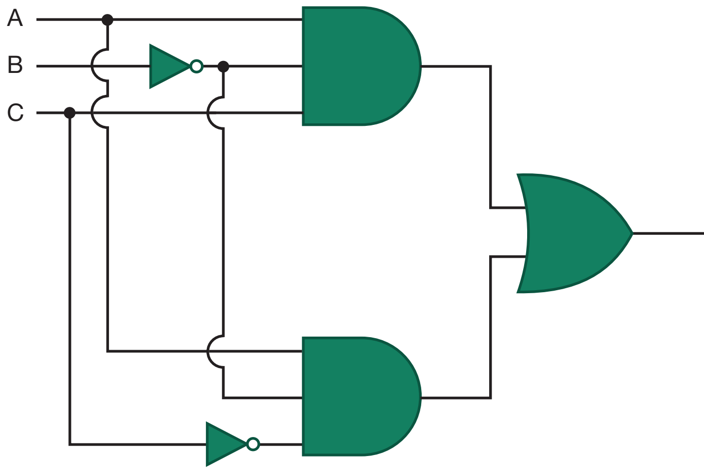
Rung 0 in the above figure provides a variation on the basic OR function. The XOR (exclusive OR) logic function Is based on exclusivity. The output of an XOR logic function rung will go HIGH when only ONE of the inputs is HIGH. This holds true for all input combinations with the exception of the state where all three input switches are closed.

General Overview Three Input Logic Gate Truth Table

Three Input Logic Functions								
Inputs			Outputs		Outputs		Outputs	
A	B	C	AND	NAND	OR	NOR	XOR	XNOR
0	0	0	0	1	0	1	0	1
0	0	1	0	1	1	0	1	0
0	1	0	0	1	1	0	1	0
0	1	1	0	1	1	0	0	1
1	0	0	0	1	1	0	1	0
1	0	1	0	1	1	0	0	1
1	1	0	0	1	1	0	0	1
1	1	1	1	0	1	0	1	0

The above figure illustrates the output logic for the basic 6 three input logic gates. Logic gates are used in decision making rungs when conditional logic is required to direct the behaviour of a specific output.

Combinational Logic Gate Behaviour



Decision Making Logic Circuit - Combinational Logic Diagram Using Standard 2 and 3 Input Logic Gates as well as Inverters.

Although the basic gates have been covered here, as can be seen in the figure above, the power of these basic building blocks with respect to decision making and direction taking resides in the ability to combine them to create combinational logic decision making program segments. The gate representation of this decision making circuit can actually be reduced using Boolean algebra to the following logic function:

A and NOT B = Z

This can be arrived at by creating a truth table, and then using reduction theorems to simplify the logic.

Basic Boolean Reduction Theorems			
Theorems	**Equivalent electric circuit**	**Reduced form**	**Boolean form**
1	A A	A	$A \cdot A = A$
2	A open circuit	open circuit	$A \cdot 0 = 0$
3	A short circuit	short circuit	$A + 1 = 1$
4	A $\bar{A}$	short circuit	$A + \bar{A} = 1$
5	A A	A	$A + A = A$
6	A $\bar{A}$	open circuit	$A \cdot \bar{A} = 0$
7	A short circuit	A	$A \cdot 1 = A$
8	A open circuit	A	$A + 0 = A$

The above table lists the basic 8 boolean reductions. These identities allow you to reduce the number of logic gates being used to perform the desired logical operation.

An additional 7 general Boolean Reduction Theorems

9. x + y = y + x

10. (x + y) + z = x + (y + z)

11. xy = yx

12. 1x(yz) = (xy)z

13. x(y + z) = xy + xz

14. x + xy = x

15. x +~xy = x + y

These additional boolean reduction theorems will allow you to reduce complex multiterm boolean expressions down to their simplest forms. These 15 theorems in combination with DeMorgans theorems, (next) constitute the basis for boolean algebra.

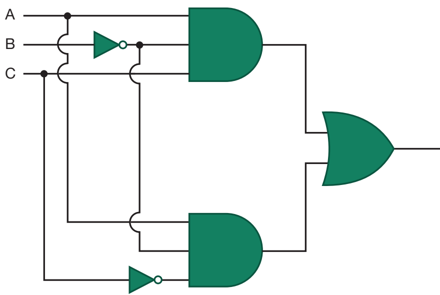
DeMorgan's Reduction Theorems

Theorem 1. The complement of a sum equals the product of the complements. $\overline{A + B} = \overline{A} \cdot \overline{B}$

Theorem 2. The complement of a product equals the sum of the complements. $\overline{A \cdot B} = \overline{A} + \overline{B}$

Theorem 1 allows you to convert a NOR based pair of conditions to a NAND based pair of conditions. Theorem 2 reverses this function. It can often be convenient to re-express the way in which you are viewing the condition when trying to reduce a Boolean expression to its simplest form.

A simple reduction for our combinational gate logic example:



Truth Table For This Gate Array				
A	**B**	**C**	**Z**	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	1	(A~B~C)
1	0	1	1	(A~BC)
1	1	0	0	
1	1	1	0	

Boolean Reduction

Truth Table: A~B~C + A~BC = Z

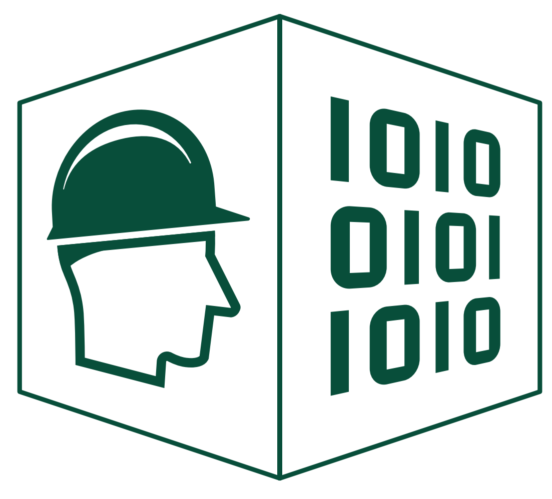
Theorem 18. (A~B) (~C + C) = Z

Theorem 4. (A~B) (1) = Z

Theorem 7. A~B = Z

Z = A and NOT B

3 SUPPLEMENTAL CONTENT



COMMON PLC FIELD DEVICES AND SCHEMATIC SYMBOLS

In this section, you are provided with some common input and output field devices used in PLCs, their relevant instruction, along with a brief description and schematic symbols.

Instruction	Full name	Symbols	Description
XIC/ NO	Examine If Closed / Normally open	Momentary pushbutton	Pushing the button will complete the connection between the terminals and allow the current to flow.
		Single Pole Single Throw Switch (SPST)	SPST is a switch that only has a single input and can connect only to one output. This means it only has one input terminal and only one output terminal. Serving as an on off switch, the circuit is on when the switch is closed and off when it's open.
		Double Pole Single Throw Switch (DPST)	This is a pair of single pole switches that are electrically separate but are linked to the same mechanical switch. A DPST switch is often used for switching mains operated devices as both the Live and Neutral wires are switched on or off simultaneously.
		Limit switch	When used in a circuit a NO contact in the limit switch remains open until its actuator is pressed, and closes the circuit.
		Normally open held closed switch	With this type of limit switch, something has to continuously push the switch to keep the contacts closed; if the pushing stops the contacts will open.
		Pressure switch	A pressure switch is a device that operates an electrical contact when a preset fluid pressure is reached. The switch may be designed to make contact either on pressure rise or on pressure fall. A NO pressure switch closes with the detection of a critical pressure value.

Instruction	Full name	Symbols	Description
XIC/ NO	Examine If Closed / Normally open	Level switch	Level switches detect the level of liquid or solid (granules) in a vessel. They often use float as a level sensing element. In a NO level switch, the switch closes and let the current flows when the level of liquid or solid rises to the threshold level and disconnected when the level is below the desired level.
		Temperature switch	A temperature switch is responsible for the monitoring and controlling of temperature, with the ability to turn on and off when a certain temperature is reached. NO temperature switch closes when the threshold temperature is reached.
		Flow switch	Flow switches are used to monitor the flow rate of air or liquid within an industrial process system. Flow switch often use paddles as the flow sensing elements. In NO flow switches, the switch remains in the default state (open) until it is triggered by a setpoint value.
		Normally-Open, Timed-Closed Switch (NOTC)	This type of contact is normally open when the coil is unpowered (de-energized). The contact is closed only after power has been applied to the relay coils for a specific amount of time (after the required time has elapsed).
		Normally-Open, Timed-Open Switch (NOTO)	The normally- open time opening (NOTO) contact will close the instant that the timing coil receives power. When the timing coil is de-energized, the time delay is initiated, and the NOTO leg will return to its open position after the timer has timed out.
XIO/NC	Examine If Open / Normally closed	Momentary pushbutton	When connected to the circuit, shorts the connected terminals, and current readily flows through the circuit. Pushing the button will separate the contacts and interrupt current flow for the time the switch is pressed.

Instruction	Full name	Symbols	Description
XIO/NC	Examine If Open / Normally closed	Limit switch	When used in a circuit a NC limit switch breaks the circuit or current flow when its actuator is pressed.
		Normally closed held open switch	With this type of switch something has to continuously push the switch to keep the contacts open; if the pushing stops the contacts will close.
		Pressure switch	In NC pressure switch, an increase in pressure beyond the desired value will cause the switch to open, and disconnect the power supplied. This would typically be applied as a high-pressure switch.
		Level switch	A level switch is a sensor that detects the presence of liquids, powder, or granulated materials at a specific location. Below threshold level the contact remains closed and when the level rises beyond the threshold, the contact opens and disconnects the current flow between the terminals.
		Temperature switch	NC temperature switch remains closed when it senses minimal temperature but gets open when senses the threshold temperature.
		Flow switch	A flow switch is used to monitor and control the flow rate of fluid within an industrial process system. When the flow rate reaches a switch's set-point, it can either open or close the circuit which triggers an action. With a NC switch, the circuit is closed (ON) until triggered otherwise.
		Normally-Closed, Timed-Open Switch (NCTO)	This type of contact is normally closed when the coil is unpowered (de-energized). The contact is opened after the coil has been continuously powered for the specified amount of time with the application of power to the relay coil.
		Normally-Closed, Timed-Closed Switch (NCTC)	This type of contact is normally closed when the coil is unpowered (de-energized), the timing leg will open the instant the timing coil receives power, and will remain open until the timer has timed out. After the timing cycle is completed, the NCTC leg will return to its closed position.

Instruction	Full name	Symbols	Description
OTE	Out put Energize	Solenoid valve	Solenoid valves consist of a coil, plunger and sleeve assembly. Unlike electrical switches, the terms open and closed have opposite meanings for valves. A normally “open” valve freely allows fluid to flow through it and closes when actuated. On the other hand a normally “closed” valve prevents the flow of fluids at rest state and opens when actuated allowing fluid flow. Instead of controlling electrical power, solenoid valves generally control fluidic power.
		Relay coil	Relay coil is a type of electromagnetic switch. When a relay is used as an OTE and the coil gets energized, a magnetic field is generated. This magnetic field attracts the contacts of the relay, causing them to be latched as long as a specific value of holding current flows through the coil. When the current through the coil is reduced below this value, the core becomes unmagnetized and the armature is pulled away to its unactuated position.
		Pilot lamp R	Pilot lamps are commonly used as output devices in the control system. They usually used as an indicator of the system status.
		Electric motor M	Motors are also commonly used output field devices in automation system. Motors are devices that convert electrical energy into mechanical energy using magnetic fields. There are different types of AC and DC motors used in automation system.

NUMBER SYSTEMS AND CODES USED WITH PLC

A number system is essentially a code consisting of symbols which are assigned for each individual quantity. Once a code is memorized, it is possible to count using this code. There are several numbering systems used in PLCs, such as **Decimal, Binary, Octal, Hexadecimal and BCD**. In this section we will examine these number systems and see how we can convert any of the other base system to the equivalent decimal number.

The radix, or base, of a number system is the total number of individual symbols in that system. The largest-valued symbol always has a magnitude of one less than the radix. For example, the decimal number system has a radix of 10, so the largest single digit is 10 – 1, or 9. Each number is represented by its base. If the base is 2 it is a binary number, if the base is 8 it is an octal number, if the base is 10 it is a decimal number and if the base is 16, it is the hexadecimal number system.

All number systems use position weighting to represent the significance of an individual digit in a group of numbers. As the digits move to the left of a decimal point, the value of the digit increases by its base power. If a digit moves to the right of the decimal place, it decreases by its base power. When the digits are grouped together, larger quantities can be expressed.

Decimal Number System

The **decimal number system** is the most common number system we are familiar with and consist of ten individual digits, 0 through 9 to represent any number imaginable. Since there are ten digits in this system, the decimal number system is referred to as a base 10 system and each value in this number system has the place value of power 10. As the digits move to the left of the decimal place, they increase by a power of 10. This means the digit in the hundreds place is ten times greater than the digit in the tens place. For example:

$$(625)_{10} = 6x10^2 + 2x10^1 + 5x10^0$$

As the digits move to the right, they decrease by a power of 10 as shown in this example:

$$(37.140)_{10} = 3x10^1 + 7x10^0 + 1x10^{-1} + 4x10^{-2} + 0x10^{-3}$$

The total value of the decimal number **5911.14** is determined as follows:

$4 \times 10^{-2} = 0.04$

$1 \times 10^{-1} = 0.1$

$1 \times 10^{0} = 1$

$1 \times 10^{1} = 10$

$9 \times 10^{2} = 900$

$5 \times 10^{3} = 5{,}000$

5,911.14

Binary Number System

In PLCs the Base 2 numbering system know as the **binary numbering system** is used. It uses two digits 1s and 0s to represent electrical signals of discrete numerical quantities. A discrete signal has one of two possible states: ON or OFF, 1 or zero, high or low, energized or de-energized. A single discrete signal is also known as a **bit**. When four bits are grouped together, they form what is known as a nibble. Eight bits or two nibbles is a byte. Sixteen bits or two bytes is a word. Thirty-two bits or two words is a double-word.

Figure 1 below is an illustration of a word in bits:

Word															
Byte								Byte							
Nibble				Nibble				Nibble				Nibble			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 1

Typically, 5 volts is used to represent binary 1, and 0 volts is used to represent binary 0 and this indicates ON or OFF for voltage level of 0 V or 5 V. For example, the binary word 10110111 would appear as the following voltage levels:

5 V, 0 V, 5 V, 5 V, 0 V, 5 V, 5 V, 5 V

When the processor 'sees' these voltage levels it considers the binary number 10110111 to be present. Because only two symbols, or digits, are used in the binary number system, the base, or radix, is two. Therefore, the position weighting that is assigned to a binary symbol will double each position a digit is moved to the left of the decimal point, and it will decrease by ½ each position a digit is moved to the right of the decimal point. The digit of a binary number that has the lowest weight is called the Least Significant Bit, or **LSB**, and the digit with the highest value is called the Most Significant Bit, or **MSB**.

A nibble of 1011_2 would be equal to a decimal number **11 or $(1x2^3 + 1x2^1 + 1x2^0)$ or $(8_{10} + 2_{10} + 1_{10})$**. A byte of 11010111_2 would be equal to **215_{10} or $(1x2^7 + 1x2^6 + 1x2^4 + 1x2^2 + 1x2^1 + 1x2^0)$ or $(128_{10} + 64_{10} + 16_{10} + 4_{10} + 2_{10} + 1_{10})$**.

Figure 2 shows how base 2 numbers relate to their decimal equivalent:

Bit #	Power	Decimal Bit Value	Max Value
0	2^0	1	
1	2^1	2	
2	2^2	4	
3	2^3	8	
4	2^4	16	
5	2^5	32	
6	2^6	64	
7	2^7	128	65535_{10}
8	2^8	256	
9	2^9	512	
10	2^{10}	1024	
11	2^{11}	2048	
12	2^{12}	4096	
13	2^{13}	8192	
14	2^{14}	16384	
15	2^{15}	32678	

Figure 2: Binary – Decimal Equivalent

Positive Decimal Numbers

The far-left position will always be 0 for positive values. As indicated in **Figure 3**, this limits the maximum positive decimal value to 32767. All positions are 1 except the far-left position.

For example:

$$0000\ 1001\ 0000\ 1110 = 2^{11} + 2^{8} + 2^{3} + 2^{2} + 2^{1}$$

$$= 2048 + 256 + 8 + 4 + 2$$

$$0000\ 1001\ 0000\ 1110 = 2318$$

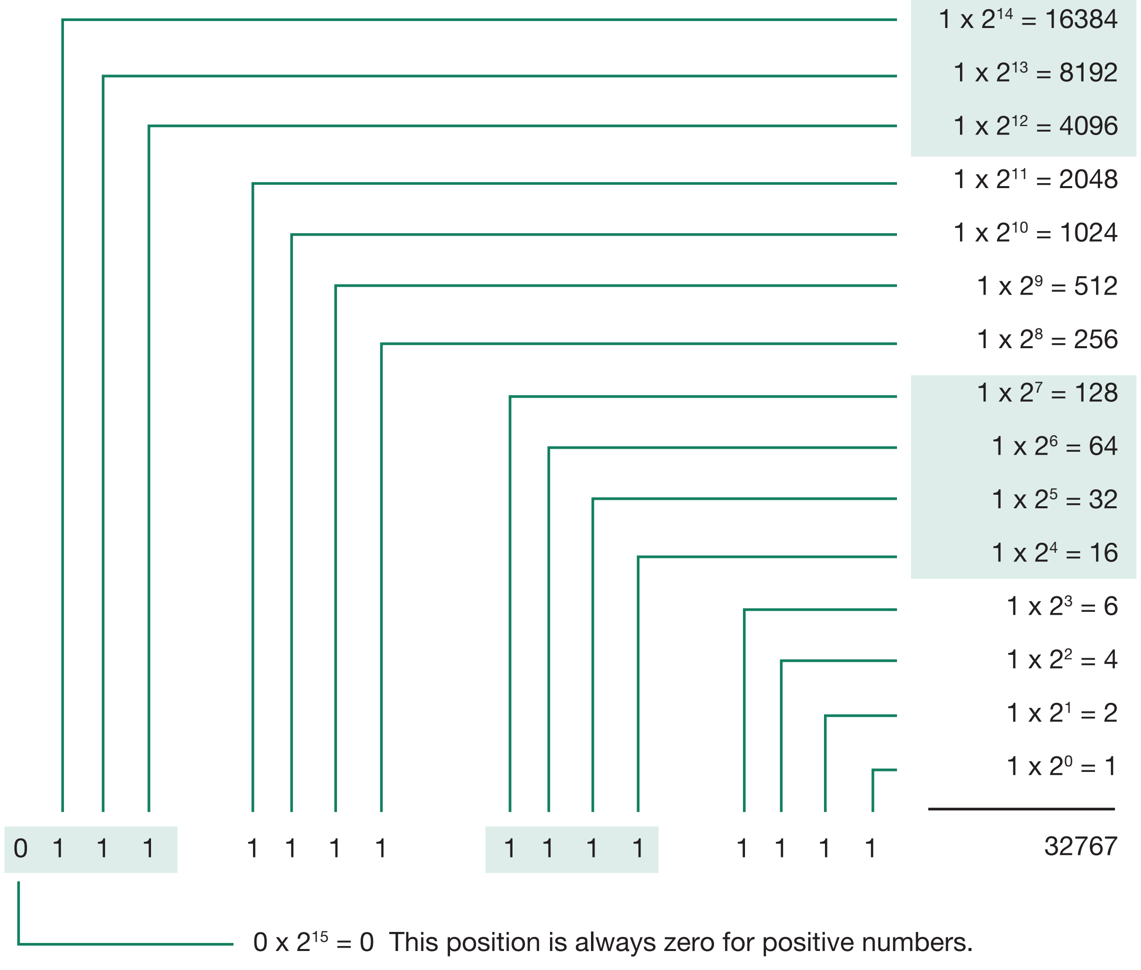


Figure 3

Negative Decimal Numbers

A negative number is represented by a subtraction symbol in front of the number and if a decimal number is positive, it has a plus sign; this indicates that each number has both a magnitude and a sign. It is not possible to use positive and negative symbols to represent the polarity of a number in PLCs, so we simply use an extra digit, or a sign bit at the MSB of the number. **Figure 4** shows a four-bit binary number expressed in sign magnitude form:

Sign	Magnitude			Decimal value
0	1	1	1	+ 7
0	1	1	0	+ 6
0	1	0	1	+ 5
0	1	0	0	+ 4
0	0	1	1	+ 3
0	0	1	0	+ 2
0	0	0	1	+ 1
0	0	0	0	0
1	0	0	1	- 1
1	0	1	0	- 2
1	0	1	1	- 3
1	1	0	0	- 4
1	1	0	1	- 5
1	1	1	0	- 6
1	1	1	1	- 7

Figure 4: Signed Binary Number

The far-left position is always 1 for negative numbers. The equivalent decimal value of the binary number is obtained by subtracting the value of the far-left position, 32768, from the sum of the values of the other positions. In **Figure 5**, the value is 32767 - 32768 = -1. All positions are 1.

For example:

1111 1000 0010 0011

$= (2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{5} + 2^{1} + 2^{0}) - 2^{15}$

= (16384 + 8192 + 4096 + 2048 + 32 + 2 + 1) - 32768

= 30755 - 32768

= -2013.

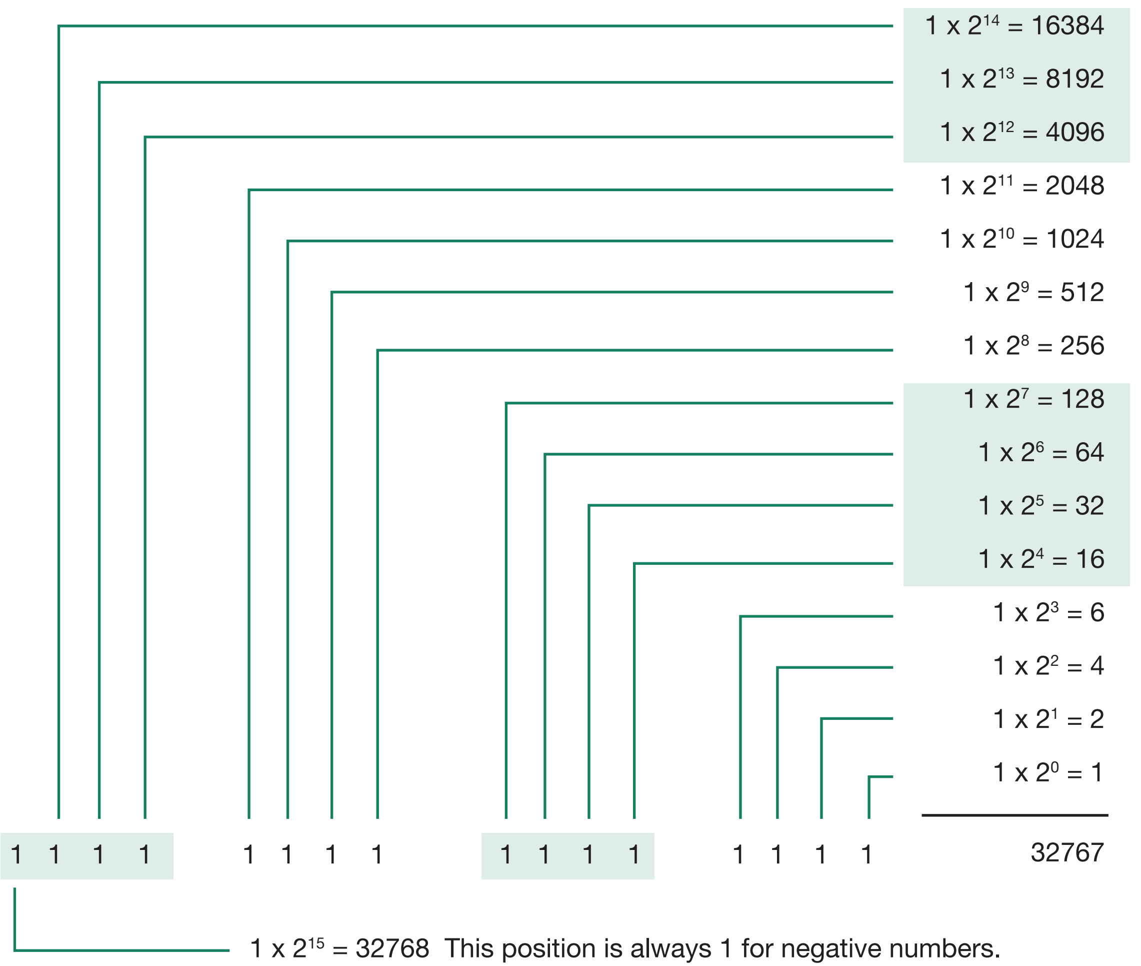


Figure 5

An often-easier way to calculate a value is to locate the last 1 in the string of 1s beginning at the left and subtract its value from the total value of positions to the right of that position.

For example:

$$1111\ 1111\ 0001\ 1010 = (2^4 + 2^3 + 2^1) - 2^8$$

$$= (16 + 8 + 2) - 256$$

$$= -230.$$

The 2s complement notion is also used to express a negative binary number. If a binary number has the value 1, the complement would be 0 and if the binary number is 0, the complement would be 1. In other words, the

1's complement of 1001 is 0110. When negative numbers are represented in 1's complement, each negative number's magnitude is the 1's complement of the corresponding positive number's magnitude. **Figure 6** shows the 1's complement of four-digit binary numbers.

Sign	Magnitude			Decimal value
0	1	1	1	+ 7
0	1	1	0	+ 6
0	1	0	1	+ 5
0	1	0	0	+ 4
0	0	1	1	+ 3
0	0	1	0	+ 2
0	0	0	1	+ 1
0	0	0	0	0
1	1	1	0	- 1
1	1	0	1	- 2
1	1	0	0	- 3
1	0	1	1	- 4
1	0	1	0	- 5
1	0	0	1	- 6
1	0	0	0	- 7

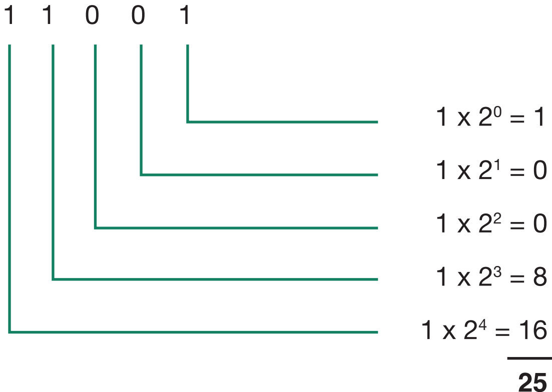
Figure 6

2s complement is similar to the 1's complement but 1 digit is used to represent the sign. Using the 2's complement, binary numbers are actually added together to produce a subtraction operation. **2's complement = 1 + 1's complement**.

Binary and Decimal Conversion

The conversion of binary digits to decimal is referred to as decoding. When a number is manually converted from binary to decimal, the position weight of each binary digit is added together. The result is the equivalent decimal number.

Example: **Convert 11001_2 to decimal.**



The sum of 1 + 8 + 16 = 25. Therefore, the decimal equivalent of binary number 11001 is 25.

Decimal values that are entered into a digital machine must be converted, or encoded, into binary form. To convert a decimal number into binary form, divide the decimal number by 2 and record the remainders as shown:

Convert 2510 to binary:

$\frac{25}{2}$ = 12 + remainder of 1

$\frac{12}{2}$ = 6 + remainder of 0

$\frac{6}{2}$ = 3 + remainder of 0

$\frac{3}{2}$ = 1 + remainder of 1

$\frac{1}{2}$ = 0 + remainder of 1

1 1 0 0 1

MSB **LSB**

The equivalent binary number of decimal number 25 is 11001.

Octal Number System

The octal number system has a base of eight, which means that it has eight possible digits: 0, 1, 2, 3, 4, 5, 6, and 7. Position weighting is applied to the octal system in the same way it is applied to the decimal and binary systems. So as digits move to the left of the decimal place, they increase by the power of their base and as the digits move to the right of the decimal place, they decrease by the power of base 8.

The weights of the digit positions in an octal number are as follows:

$$8^n \ldots\ 8^4\ 8^3\ 8^2\ 8^1\ 8^0\ .\ 8^{-1}\ 8^{-2}\ 8^{-3}\ 8^{-4} \ldots 8^{-n}$$

Converting Octal to Decimal

To convert an octal number to a decimal number, multiply each octal digit by its weight and add the resulting products. For example, octal number 17 is converted to decimal in the following manner:

$$1(8^1) + 7(8^0) = 8 + 7 = 17 \text{ decimal}$$

Converting Decimal to Octal

To convert a decimal number to an octal number we divide the decimal number by 8. The remainders form the decimal number, with the first remainder being the least-significant digit and the last remainder being the most-significant digit.

Here is an example of converting 214_{10} to octal:

$$\frac{214}{8} = 26 \text{ with a remainder of } 6$$

$$\frac{26}{8} = 3 \text{ with a remainder of } 2$$

$$\frac{3}{8} = 0 \text{ with a remainder of } 3$$

Result = 326

When converting a decimal fraction to octal, the decimal number is multiplied instead of divided. Also, the first number divided is now the most- significant digit, and the last number is the least- significant digit.

For example, to change 0.3510 to octal, proceed as follows:

0.35 x 8 = 2.8 = 0.8 with a carry of 2

0.8 x 8 = 6.4 = 0.4 with a carry of 6

0.4 x 8 = 3.2 = 0.2 with a carry of 3

etc.

Result = 0.263

Converting Octal to Binary

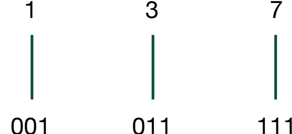
To convert an octal number to its binary equivalent we change each octal digit to its three-bit binary equivalent. The eight possible digits are converted as shown in **Figure 7**:

Octal Number	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111

Figure 7

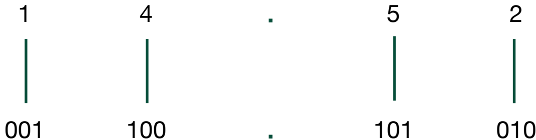
Here are examples of converting octal numbers to their binary equivalent:

1. **Converting octal 137 to binary:**



The binary equivalent of 137_8 is **001 011 111**, or **1011111**

2. **Converting octal 14.52 to binary:**



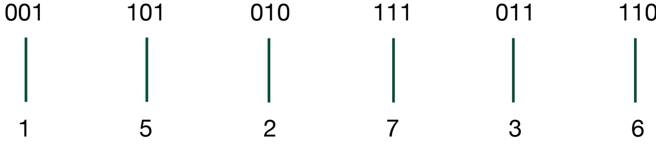
The binary equivalent of 14.52_8 is **001 100 . 101 010**, or **1100 . 101010**

Converting Binary to Octal

To convert a binary number to octal, simply divide the binary number into groups of three bits starting from the least significant bit to the most significant bit, then convert each three bits group binary number to their octal equivalent value using the 4-2-1 weighting. If the last group does not have three bits, add 0s to most significant digit side of the binary number.

Here are examples of converting binary numbers to their octal equivalent:

3. **Convert binary number 1101010111011110 to octal number**



11010101110111102 = 152736_8

4. Convert binary number 100011.1011 to octal number

$100011.1011_2 = 43.54_8$

Hexadecimal Number System

Hexadecimal or Hex is a numbering system that uses Base 16. This system uses the ten digits in the decimal system, 0 through 9, as well as the first six letters of the alphabet, A, B, C, D, E, and F with each letter representing the decimal numbers 10 through 15. Since binary number system is widely used in PLC and easy to interpret for a few bits we will see that Hex aid in simplifying the lengthy combination of bits for very large numbers since it can become difficult to keep track of the bit position when converting very large binary numbers. **Figure 8** shows the comparison between decimal, binary and hexadecimal digits.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Figure 8

The weights of the digit positions in a hexadecimal number are as follows:

$16^4\ 16^3\ 16^2\ 16^1\ 16^0\ .\ 16^{-1}\ 16^{-2}\ 16^{-3}\ 16^{-4}$

Converting Hexadecimal to Decimal

To convert a hexadecimal number to a decimal number, multiply each hexadecimal digit by its weight and add the resulting products together. The following examples illustrates hexadecimal to decimal conversion:

1. $637_{16} = 6 \times 16^2 + 3 \times 16^1 + 7 \times 16^0$

$= 1,536 + 48 + 7$

$637_{16} = 1,591_{10}$

2. $4CF_{16} = 4 \times 16^2 + 12 \times 16^1 + 15 \times 16^0$

$= 1,024 + 192 + 15$

$4CF_{16} = 1231_{10}$

Converting Decimal to Hexadecimal

Similar to converting from a decimal number to binary or to octal number, we convert form a decimal to a hexadecimal number by dividing the decimal number by 16 (the base) and keep track of the remainders. The remainder forms the equivalent hexadecimal number.

Here is an example of converting 541 to hex:

$\frac{541}{16}$ **= 33 with a remainder of 13 (D) - (MSB)**

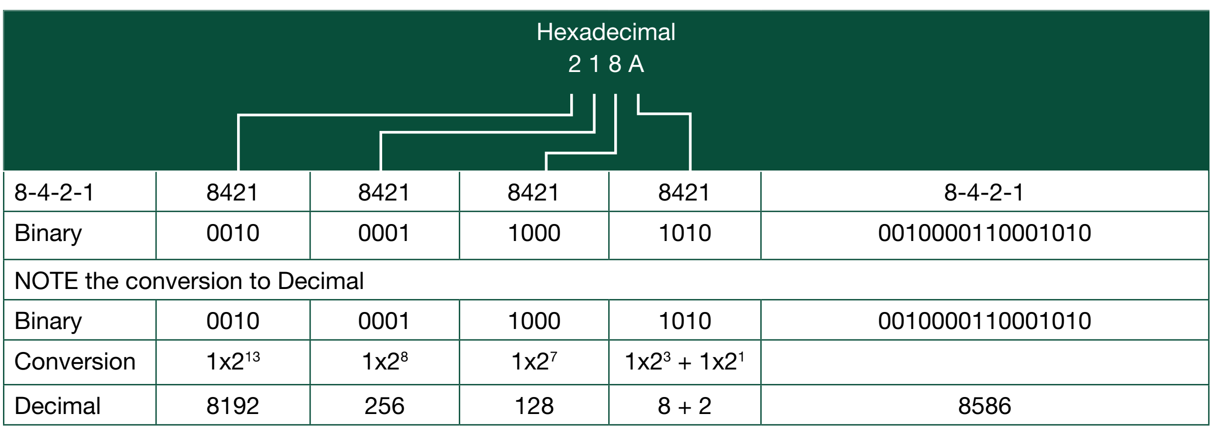
$\frac{33}{16}$ **= 2 with a remainder of 1**

$\frac{1}{16}$ **= 0 with a remainder of 1 - (LSB)**

The hexadecimal equivalent of the decimal number 541_{10} is $11D_{16}$.

Converting Hexadecimal to Binary

To convert hexadecimal to binary we determine the four bit equivalent binary digit for each hexadecimal digit using the 8-2-4-1 system shown in **Figure 9** and the decimal equivalence is also shown.



Hexadecimal 2 1 8 A					
8-4-2-1	8421	8421	8421	8421	8-4-2-1
Binary	0010	0001	1000	1010	0010000110001010
NOTE the conversion to Decimal					
Binary	0010	0001	1000	1010	0010000110001010
Conversion	1×2^{13}	1×2^{8}	1×2^{7}	$1 \times 2^{3} + 1 \times 2^{1}$	
Decimal	8192	256	128	8 + 2	8586

Figure 9a: Hexadecimal 218A = 0010000110001010

Decimal 8586					
Binary	0010	0001	1000	1010	= 8586
1's Complement	1101	1110	0111	0101	
2's Complement	1101	1110	0111	0110	= -8586
Hexadecimal	D	E	7	6	= 56905

Figure 9b: Decimal number -8586 in equivalent binary and hexadecimal forms

Hexadecimal number **DE76 = $13 \times 16^3 + 14 \times 16^2 + 7 \times 16^1 + 6 \times 16^0$ = 56950.**

We know this is a negative number because it exceeds the maximum positive value of 32767. To calculate its value, subtract 16^4 (the next higher power of 16) from 56950: 56950 - 65536 = -8586 **OR** work backwards from 2's complement to 1's complement, then invert the binary digits and convert to decimal to obtain the negative decimal equivalent value.

Binary Coded Decimal (BCD)

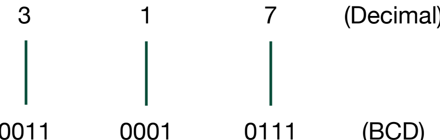
Like Octal and Hexadecimal, the BCD numbering system relies on bit-coded digits in base 10. It modifies the binary number system where the decimal digits are independently coded as four-bit binary numbers. We will examine the 8-4-2-1 BCD system in this discussion. The designation 8-4-2-1 indicates the binary weights of the four bits (2^3, 2^2, 2^1, 2^0). Using four bits, it is possible to count from 1 to 15. However, in the BCD system, the maximum value is 9, the six numbers over 9 are not valid, since it must convert to a single digit. Any number over 9 must be represented by two BCD numbers, or eight bits as shown in **Figure 10**.

Decimal	BCD	Binary
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	0001 0000	1010
11	0001 0001	1011
12	0001 0010	1100
13	0001 0011	1101
14	0001 0100	1110
15	0001 0101	1111

Figure 10

BCD Conversion

A decimal number is expressed as BCD by changing each digit to its binary equivalent as illustrated for decimal value 317:



001100010111 is the BCD equivalent of decimal number 317, however the 0s on the MSB side can be ignored so 1100010111 is also a correct expression of the decimal number 317.

To determine a decimal number from a BCD number. Start at the least significant bit and break the code into groups of four bits. The decimal digit represented by each four-bit group is then written as shown:

Parity Bit, Grey Code and ASCII

In PLCs when data is being transferred between PLCs or from a PLC to a peripheral device, a binary digit can be changed accidentally. A **parity bit** is used in transmission to detect errors when a word is being moved. There are two methods Even parity and Odd parity which is used to detect errors. **Even parity** adds a digit to a binary word to make the total number of 1s in the word even. For instance the binary word 010101. There are three 1s in this word, therefore, to make the word conform to even parity, a 1 is added to the end of the word, which results in 0101011. A word that already had an even number of ones would have a 0 parity bit. Likewise, **Odd parity** is used to make the number of 1's odd. **Figure 11** shows the 8-4-2-1 code with even and odd parity.

8-4-2-1 Code	Even Parity Bit	Odd Parity Bit
0000	0	1
0001	1	0
0010	1	0
0011	0	1
0100	1	0
0101	0	1
0110	0	1
0111	1	0
1000	1	0
1001	0	1

Figure 11: 8-4-2-1 Code with Even and Odd Parity

The **Gray code** is a special type of binary code that does not use position weighting. In other words, each position does not have a definite weight. It is used to minimize the error that may occur when transitioning form one number to the next through a sequence. Gray code is set up so that as we progress from one number to the next, only one bit changes. This makes the speed of transmission faster than codes like BCD since only one bit changes at a time. The Gray code is ideally suited for absolute encoders and other digital counting devices due to the high degree of accuracy associated with this code. Here in **Figure 12** is a comparison of Gray code and binary equivalents with respect to decimal:

Decimal	Binary	Gray Code	Decimal of Gray
0	0000	0000	0
1	0001	0001	1
2	0010	0011	3
3	0011	0010	2
4	0100	0110	6
5	0101	0111	7
6	0110	0101	5
7	0111	0100	4
8	1000	1100	12
9	1001	1101	13
10	1010	1111	15
11	1011	1110	14
12	1100	1010	10
13	1101	1011	11
14	1110	1001	9
15	1111	1000	8

Figure 12

The decimal value "1" is representation in binary would normally be "0001" and "2" would be "0010". In Gray code, these values are represented as "0001" and "0011". That way, incrementing a value from 1 to 2 requires only one bit to change, instead of two.

The abbreviation **ASCII** stands for **American Standard Code for Information Interchange**. It is an alphanumeric code which includes letters as well as numbers and special characters such as those found on standard typewriters and computer keyboards. These includes 10 numeric digits, 26 upper-case letters, 26 lower-case letters and 25 special characters.

The ASCII code is a seven-bit code in which the decimal digits are represented by the 8-4-2-1 BCD code preceded by 011. Upper-case letters are preceded by 100 or 101. Lower-case letters are preceded by 110 or 111. Character symbols are preceded by 010, 011, 101, and 111. This seven-bit code provides all possible combinations of characters used when communicating with peripherals or interfaces in a PLC system. In **Figure 13a** we find a partial listing of the ASCII code and a Standard ASCII Control Code and Character Code in **Figure 13b**.

Character	7-bit ASCII	Character	7-bit ASCII
A	100 0001	P	011 0100
B	100 0010	Q	011 0101
C	100 0011	R	101 0010
D	100 0100	S	101 0011
E	100 0101	T	101 0100
F	100 0110	U	101 0101
G	100 0111	V	101 0110
H	100 1000	W	101 0111
I	100 1001	X	101 1000
J	100 1010	Y	101 1001
K	100 1011	Z	101 1010
L	100 1100	‘	010 1100
M	100 1101	+	010 1010
N	100 1110	#	010 0011
O	100 1111	-	011 1101

Figure 13a: Partial Listing of ASCII Code

Bin		000	001	010	011	100	101	110	111
	Hex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0000	0	**NUL**	**DLE**	**SP**	**0**	**@**	**P**	**`**	**p**
0001	1	**SOH**	**DC1**	**!**	**1**	**A**	**Q**	**a**	**q**
0010	2	**STX**	**DC2**	**"**	**2**	**B**	**R**	**b**	**r**
0011	3	**ETX**	**DC3**	**#**	**3**	**C**	**S**	**c**	**s**
0100	4	**EOT**	**DC4**	**$**	**4**	**D**	**T**	**d**	**t**
0101	5	**ENQ**	**NAK**	**%**	**5**	**E**	**U**	**e**	**u**
0110	6	**ACK**	**SYN**	**&**	**6**	**F**	**V**	**f**	**v**
0111	7	**BEL**	**ETB**	**'**	**7**	**G**	**W**	**g**	**w**
1000	8	**BS**	**CAN**	**(**	**8**	**H**	**X**	**h**	**x**
1001	9	**HT**	**EM**	**)**	**9**	**I**	**Y**	**i**	**y**
1010	A	**LF**	**SUB**	*****	**:**	**J**	**Z**	**j**	**z**
1011	B	**VT**	**ESC**	**+**	**;**	**K**	**[**	**k**	**{**
1100	C	**FF**	**FS**	**,**	**<**	**L**	****	**l**	**l**
1101	D	**CR**	**GS**	**-**	**=**	**M**	**]**	**m**	**}**
1110	E	**SO**	**RS**	**.**	**>**	**N**	**^**	**n**	**~**
1111	F	**SI**	**US**	**/**	**?**	**O**	**_**	**o**	**DEL**

Figure 13b: Standard ASCII Control Code and Character Code

CORE INSTUCTION SET

The PLCLogix instructions encompass all the main ladder logic programming commands associated with Logix 5000. The PLCLogix Instruction Set consists of the following groups of commands: Bit Instructions, Timer and Counter Instructions, Program Control, Compare, Communications, Math, and Data Handling/Transfer instructions.

1. Bit Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
XIC	Examine If Closed	?] [	Examines a bit for an On (set, high) condition.
XIO	Examine If Open	?]/[	Examines a bit for an Off (cleared, low) condition.
OTE	Output Energize	? ( )	When rung conditions are true, the OTE will either set or clear the data bit.
OTL	Output Latch	? (L)	When enabled, the instruction signals to the controller to turn on the addressed bit. The bit remains on, regardless of the rung condition.
OTU	Output Unlatch	? (U)	When enabled, it clears (unlatches) the data bit. The bit remains Off, regardless of rung condition.
ONS	One Shot	? [ONS]	Enable/disable outputs for one scan. Storage bit status determines whether this instruction enables or disables the rest of the rung.
OSR	One Shot Rising	OSR One Shot Rising Storage Bit ? Output Bit ? (OB) (SB)	A retentive input instruction that triggers an event to occur once. It either sets or clears the output bit, depending on the storage bit status.
OSF	One Shot Falling	OSF One Shot Falling Storage Bit ? Output Bit ? (OB) (SB)	This instruction either sets or clears the output bit, depending on the storage bit's status.

2. Timer and Counter Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
TON	Timer ON Delay	TON Timer On Delay Timer ? Preset ? Accum ? (EN) (DN)	A non-retentive timer that accumulates time when the instruction is enabled. The accumulated value is reset when rung conditions go false.
TOF	Timer Off Delay	TOF Timer Off Delay Timer ? Preset ? Accum ? (EN) (DN)	A non-retentive timer that accumulates time when the rung makes a true-to-false transition.
RTO	Retentive Timer On	RTO Retentive Timer On Timer ? Preset ? Accum ? (EN) (DN)	A retentive timer that accumulates time when the instruction is enabled. Retains its accumulated value when rung conditions become false.
CTU	Count Up	CTU Count Up Timer ? Preset ? Accum ? (EN) (DN)	An instruction that counts false-to-true rung transitions. It counts upward and the accumulated value is incremented by one count on each of these transitions.
CTD	Count Down	CTD Count Down Timer ? Preset ? Accum ? (EN) (DN)	This instruction counts downward on each false-to-true rung transition. The accumulated value is decremented by one count on each of these transitions.
RES	Reset	? (RES)	This instruction is used to reset a timer, counter or control structure. The accumulated value of these instructions are cleared when the RES instruction is enabled.

3. Program and Control Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
JSR	Jump to Subroutine	JSR Jump to Subroutine Routine name ? Input par ? Return par ?	This instruction jumps execution to a specific routine and initiates the execution of this routine, called a subroutine.
SBR	Subroutine	SBR Subroutine Input par ?	Stores recurring sections of program logic.
RET	Return	RET Return Return par ?	Used to return to the instruction following a JSR operation.
JMP	Jump to Label	? (JMP)	Skips sections of ladder logic.
LBL	Label	? [LBL]	Target of the JMP instruction with the same label name.
MCR	Master Cont. Res.	(MCR)	Used in pairs to create a program zone that can disable all rungs between the MCR instructions.
NOP	No Operation	(NOP)	This instruction functions as a placeholder.
END	End	(END)	End rung in ladder logic circuit.
AFI	Always False Instruction	[AFI]	Sets the rung condition to False.

4. Compare Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
EQU	Equal	EQU Equal Source A ? ?? Source B ? ??	This instruction is used to test whether two values are equal. If Source A is equal to Source B, the instruction is logically true.
GEQ	Greater Than or Equal To	GEQ Grtr Than or Eql (A >= B) Source A ? ?? Source B ? ??	Determines whether source A is greater than or equal to Source B. If the value at Source A is greater than or equal to the value at Source B, then the instruction is true.
GRT	Greater Than	GRT Grtr Than (A > B) Source A ? ?? Source B ? ??	This instruction is used to test whether one value (Source A) is greater than another value (Source B).
LEQ	Less Than or Equal To	LES Less Than (A<B) Source A ? ?? Source B ? ??	Determines whether one value (Source A) is less than or equal to another (Source B).
LES	Less Than	LEQ LessThan or Eql (A <= B) Source A ? ?? Source B ? ??	This instruction determines whether Source A is less than Source B.
LIM	Limit	LIM Limit Test (CIRC) Low Limit ? ?? Test ? ?? High Limit ? ??	This instruction is used to test for values within the range of the Low Limit to the High Limit.

4. Compare Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
MEQ	Mask Equal To	MEQ Mask Equal Source ? ?? Mask ? ?? Compare ? ??	Passes the Source and Compare values through a Mask and compares the results.
NEQ	Not Equal To	NEQ Not Equal Source A ? ?? Source B ? ??	This instruction tests whether Source A is not equal to Source B.

5. Math Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
ADD	Add	ADD Add Source A ? ?? Source B ? ?? Dest ? ??	Adds Source A to Source B and stores the result in the Destination.
SUB	Subtract	SUB Subtract Source A ? ?? Source B ? ?? Dest ? ??	Subtracts Source B from Source A and places the result in the Destination.
MUL	Multiply	MUL Multiply Source A ? ?? Source B ? ?? Dest ? ??	Multiplies Source A by Source B and stores the result in the destination.
DIV	Divide	DIV Divide Source A ? ?? Source B ? ?? Dest ? ??	Divides Source A by Source B and places the result in the Destination.
MOD	Modulo	MOD Modulo Source A ? ?? Source B ? ?? Dest ? ??	Divides Source A by Source B and stores the remainder in the Destination.

5. Math Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
SQR	Square Root	SQR Square Root Source A ? ?? Source B ? ?? Dest ? ??	Calculates the square root of the source and places the float result in the Destination.
NEG	Negate	NEG Negate Source ? ?? Dest ? ??	Changes the sign (+, -) of the Source and stores the result in the Destination.
ABS	Absolute Value	ABS Absolute Value Source ? ?? Dest ? ??	Takes the absolute value of the Source and places the result in the Destination.

Advanced Math Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
SIN	Sine	SIN Sine Source ? ?? Dest ? ??	Takes the sine of the Source value (in radians} and places the result in the Destination.
COS	Cosine	COS Cosine Source ? ?? Dest ? ??	Takes the cosine of the Source value (in radians) and places the result in the Destination.
TAN	Tangent	TAN Tangent Source ? ?? Dest ? ??	Takes the tangent of the Source value (in radians) and stores the result in the Destination.
ASN	Arc Sine	ASN Arc Sine Source ? ?? Dest ? ??	Takes the arc sine of the Source value and places the result in the Destination (in radians).
ACS	Arc Cosine	ACS Arc Cosine Source ? ?? Dest ? ??	Takes the arc cosine of the Source value and stores the result in the Destination (in radians).
ATN	Arc Tangent	ATN Arc Tangent Source ? ?? Dest ? ??	Takes the arc tangent of the Source value and stores the result in the Destination (in radians).

Advanced Math Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
LN	Natural Log	LN Natural Log Source ? ?? Dest ? ??	Takes the natural log of the Source value and stores the result in the Destination.
LOG	Log to the Base 10	LOG Log Base 10 Source ? ?? Dest ? ??	Takes the log base 10 of the Source value and stores the result in the Destination.
XPY	X to the power of Y	XPY X To Power Of Y Source X ? ?? Source Y ? ?? Dest ? ??	Takes Source A (X) to the power of Source B (Y) and stores the result in the Destination.

6. Data Handling /Transfer Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
TOD	Convert to BCD	TOD To BCD Source ? ?? Dest ? ??	This instruction converts a decimal value to a BCD value and stores the result in the Destination.
FRD	Convert to Integer	FRD FROM BCD Source ? ?? Dest ? ??	Converts a BCD value (Source) to a decimal value and stores the result in the Destination.
MOV	Move	MOV MOV Source ? ?? Dest ? ??	Copies the Source (which remains unchanged) to the Destination.
MVM	Masked Move	MVM Masked Move Source ? ?? Mask ? ?? Dest ? ??	Copies the Source to a Destination and allows segments of the data to be masked.
DEG	Degrees	DEG Radians to Degrees Source ? ?? Dest ? ??	Converts the Source (in radians) to degrees and places the result in the Destination.
RAD	Radians	RAD Degrees to Radians Source ? ?? Dest ? ??	Converts the Source (in degrees) to radians and stores the result in the Destination.

6. Data Handling /Transfer Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
XOR	Bitwise Exclusive OR	XOR Bitwise Exclusive OR Source A ? ?? Source B ? ?? Dest ? ??	Performs a bitwise XOR operation using the bits in Source A and Source B and stores the result in the Destination.
CLR	Clear	CLR Clear Dest ? ??	Clears all the bits of the Destination
SWPB	Swap Byte	SWPB Swap Byte Source ? ?? Order Mode ? Dest ? ??	Rearranges the bytes stored in a tag.

7. Array/Shift Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
BSL	Bit Shift Left	BSL Bit Shift Left ? Array ? Control ? Source Bit ? Length (EN) (DN)	Shifts the specified bits within the Array (DINT) one position left.
BSR	Bit Shift Right	BSR Bit Shift Left ? Array ? Control ? Source Bit ? Length (EN) (DN)	Shifts the specified bits within the Array one position right.
FFL	FIFO Load	FFL FIFO Load ? Source ? FIFO ? Control ? Length ? Position (EN) (DN) (EM)	Copies the Source Value into a FIFO queue on successive false-to-true transitions.
FFU	FIFO Unload	FFU FIFO Unload ? FIFO ? Dest ? Control ? Length ? Position (EU) (DN) (EM)	Unloads the Source value from the first position of the FIFO and stores that value in the Destination..
LFL	LIFO Load	LFL LIFO Load ? Source ? LIFO ? Control ? Length ? Position (EN) (DN) (EM)	Copies the Source value to the LIFO.
LFU	LIFO Unload	LFU LIFO Unload ? LIFO ? Dest ? Control ? Length ? Position (EU) (DN) (EM)	Unloads the value at .POS of the LIFO and stores 0 in that location.

8. Sequencer Instruction

Instruction Mnemonic	Instruction Name	Symbol	Description
SQI	Sequencer Input	SQI Sequencer Input ? Array ? Mask ? Source ? Control ? Length ? Position	Detects when a step is complete in a sequence pair of SQO/SQI instructions.
SQO	Sequencer Output	SQO Sequencer Output ? Array ? Mask ? Dest ? Control ? Length ? Position (EN) (DN)	Sets output conditions for the next step of sequence pair of SQO/SQI instructions.
SQL	Sequencer Load	SQL Sequencer Load ? Array ? Source ? Control ? Length ? Position (EN) (DN)	Loads reference conditions into a sequencer array.
SQC	Seq. Compare	SQC Sequencer Compare ? File ? Mask ? Source ? Control ? Length ? Position (EN) (DN) (FD)	RSLogix 500 instruction. Supported by PLCLogix.

9. Communication Instructions

Instruction Mnemonic	Instruction Name	Symbol	Description
GSV	Get System Data	GSV Get System Value ? Class Name ? Instance Name ? Attribute Name ? Dest ??	Gets controller system data that is stored in objects.
SSV	Set System Data	SSV Set System Value ? Class Name ? Instance Name ? Attribute Name ? Dest ??	Sets controller system data that is stored in objects.

PLC Technician Handbook - 2022 Edition

The PLC handbook serves as a useful tool to technicians who are in training and in the workforce. It is a text designed to provide guidance by illustrating important concepts, tips and common practices and supplemental content that is relevant to the field.

To learn more about George Brown College School of Distance Education Technical Training, visit www.gbctechtraining.com

For specific information about each of our Online Certificate Programs, visit the following sites:

Automation Technician Certificate Program - www.automationprogram.com

Electronics Technician Certificate Program - www.etcourse.com

Electromechanical Technician Certificate Program - www.emcourse.com

Programmable Logic Controllers Technician Certificate Program - www.plctechnician.com

Robotics Technician Certificate Program - www.onlinerobotics.com

Any questions? Speak to a Program Consultant toll-free at 1 888-553-5333 or email us at info@gbctechtraining.com.

www.gbctechtraining.com