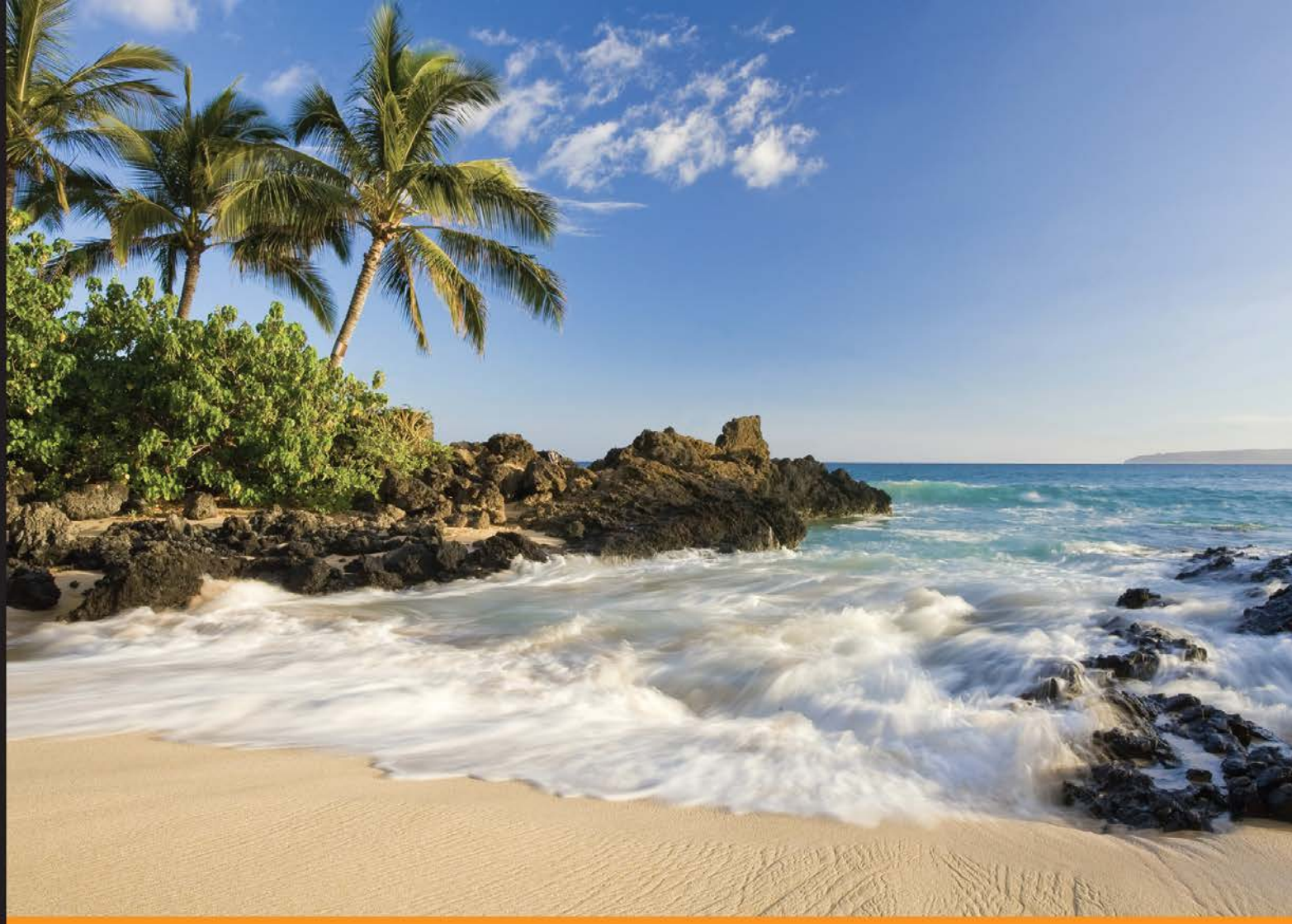


Community Experience Distilled

Learning RSLogix 5000 Programming

Become proficient in building PLC solutions in Integrated Architecture from the ground up using RSLogix 5000

Austin Scott

[PACKT] PUBLISHING

Learning RSLogix 5000 Programming

Become proficient in building PLC solutions in Integrated Architecture from the ground up using RSLogix 5000

Austin Scott

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Learning RSLogix 5000 Programming

First published: August 2015

Production reference: 1260815

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-603-9

www.packtpub.com

Credits

Author
Austin Scott

Reviewers
Tumesh
Yogesh Balajee
Sherif E. Nasr

Commissioning Editor
Usha Iyer

Acquisition Editors
Richard Brookes-Bland
Reshma Raman

Content Development Editor
Riddhi Tuljapurkar

Technical Editors
Novina Kewalramani
Chinmay S. Puranik

Copy Editor
Swati Priya

Project Coordinator
Kinjal Bari

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria
Jason Monteiro

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Austin Scott founded Synergist SCADA in 2006, a successful company that provides vendor-neutral SCADA architecture and development. Synergist has also developed a suite of engineering tools, including Citect Power Tools and Active Network Security. In July 2013, Synergist was acquired by Cimation as the catalyst for its growing Canadian operations and ongoing product development.

With more than a decade of industrial automation and software development experience, Austin has worked on large-scale, high-profile projects across North America and around the globe, incorporating most major SCADA platforms. His professional focus includes developing and refining custom software solutions to enhance the productivity of SCADA developers and improve the integration between the SCADA data and corporate applications. He is also skilled in cyber security, especially the detection of unauthorized access to SCADA networks and the forensic analysis of SCADA breaches. In 2013, he wrote *Instant PLC Programming with RSLogix 5000* by Packt Publishing.

I wish to dedicate this book to Arya, whose arrival helped me prioritize what is most important in life. Also, I would like to thank my friend, Crame Velasquez, who helped me with many of the complex illustrations used in the first chapter of this book, *ControlLogix and CompactLogix Overview and Firmware*. Furthermore, I would like to thank Packt and its incredible team for giving me the opportunity to publish this book. Thank you, Riddhi Tuljapurkar, Melita Lobo, and Richard Brookes-Bland for your seemingly infinite support and patience.

About the Reviewers

Tumesh has a BTech degree in instrumentation with many years of industrial experience. He likes to explore and develop new technologies in the field of instrumentation, and is a pioneer in the usage of new technologies and ideas.

I dedicate this work to my family.

Yogesh Balajee is a senior controls engineer. He currently manages automation optimization for two manufacturing facilities in USA. He recently completed a $50 million plant expansion project, serving as the site lead for automation and controls. He specializes in PLCs, MES, and ERP and has a vision for "Industry 4.0", which he strives to achieve. He also has a master's degree in electrical engineering with a focus on robotics and intelligent systems.

I would like to dedicate this work to my wife and father for their strong support and also to my loving mother, who watches over me.

Sherif E. Nasr has been working as an assistant professor at the Department of Engineering Mathematics in the Fayoum University's Faculty of Engineering since May 2010. He graduated from the Department of Power and Electric Machinery in the Cairo University's Faculty of Engineering in July 1996.

Besides his academic career activities, his interests lie in industrial control and field instruments in various industrial sectors.

He has supervised many projects at power plants, as well as oil and gas and food and beverage industries.

I would like to thank my kids and wife, who usually help me achieve my targets on time or even earlier.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

In 1997, Rockwell Automation launched their current generation control platform, Logix. It represented decades of automation technical advancement for robust, large-scale solutions. When it launched, it included the ControlLogix 5550 controllers (Bulletin 1756), ControlLogix I/O modules, and RSLogix 5000 programming software platform. In 2001, CompactLogix Controller (Bulletin 1769) was added to the Logix family to support intermediate-sized automation solutions under the same development platform. The RSLogix 5000 programming software (in version 21 and higher, is now referred to as Logix Designer within the Studio 5000 software package) provided a unified IEC61131-3 control platform, featuring user-friendly interfaces and workflows. Ultimately, the Logix platform reduced programming complexity, eased troubleshooting, and increased plant reliability.

RSLogix 5000 provides intuitive access to real-time information, easy to follow run-time logic animations, and a comprehensive suite of online change capabilities. Rockwell is the automation market leader in North America. Moreover, due to Rockwell Automation's continued success and the glacial speed at which most plants switch platforms, it will be the market leader for the foreseeable future. Outside North America, it is widely considered to be the fourth largest automation manufacturer (after Siemens, ABB, and Schneider). Its total global installation base is well over 2 million programmable controllers. Needless to say, as an automation professional, learning the Logix platform suite is an excellent investment of your time.

Rockwell Automation has provided a wealth of knowledge in their web-based Literature Library resources, which is the ultimate source of all the Logix platform knowledge. Rockwell has created a web of over 10,000 documents that is often difficult to navigate for beginners. *Learning RSLogix 5000 Programming* is in no way a replacement for this resource (this book would need to be 100,000 pages longer), but provides newcomers with a solid foundation in the Logix platform features and Rockwell Automation terminology. By the end of this book, the reader will have a clear understanding of the capabilities of the Logix platform and how to quickly navigate through the Rockwell Automation Literature Library resources.

Learning RSLogix 5000 Programming provides a gentle introduction to RSLogix 5000 and the Logix platform. If you understand the basics of PLC programming or have experience with programming other PLC platforms, this book will provide you with the knowledge to become proficient at implementing Logix solutions from the ground up.

What this book covers

Chapter 1, *ControlLogix and CompactLogix Overview and Firmware*, introduces the ControlLogix and CompactLogix platforms by exploring the evolution of the Allen Bradley controllers. It provides details of the Rockwell Automation Integrated Architecture and then discusses the important role that firmware plays in the Logix5000 platform.

Chapter 2, *Industrial Network Communications*, details the various communication technologies available for the Logix platform. The focus of this book is on the current state of Rockwell Automation's ControlLogix and CompactLogix controllers, however, this chapter discusses some legacy communications protocols, which you may still find running in the field today.

Chapter 3, *Configuring Logix Modules*, looks at the available modules for the Logix platform, how to configure them, and their usage in a Logix project. It also includes methods for identifying module features by their Logix Module Catalog numbers and the address tree that a typical I/O module creates.

Chapter 4, *SoftLogix*, introduces the Rockwell Automation SoftLogix 5800 Controller and Virtual Chassis. It guides you through the setup of the SoftLogix chassis monitor and configuration of your SoftLogix controller within Logix. Finally, this chapter investigates the techniques for simulating I/O using the 1784 SIM module.

Chapter 5, *Writing Ladder Logic*, looks at the history of ladder logic and the development of the IEC standard programming languages. Then, it lets you jump into ladder logic programming by creating a simple pump control program. It demonstrates how to buffer inputs and outputs in our ladder logic code and discusses the importance of this process. Finally, it explores the buffering capabilities of the new Program Parameter features in Studio 5000 Logix Designer.

Chapter 6, *Writing Function Block*, explores the merits of function block programming by building a small sample application. It also provides instructions for modifying the function block properties and performing online edits.

Chapter 7, *Writing Structured Text*, explores the strengths and weaknesses of structured text programming by exploring the typical uses of this language and demonstrates several sample applications.

Chapter 8, *Building Sequential Function Charts*, implements a sequential function chart routine and breaks down the steps, actions, transitions and branches that are used to construct it. Finally, it lets you work with the online editing capabilities of sequential function chart routines.

Chapter 9, *Using Tasks and Programs for Project Organization*, looks at the ways to structure a Logix project using the basic organization units—tasks, programs, and routines. It also looks at the ways in which task scheduling and prioritization can be used to balance the processing time of a controller.

Chapter 10, *Faults and Troubleshooting in Logix*, teaches you how to identify and troubleshoot faults in a Logix controller. It details a list of fault codes that provide insights into the problems encountered by the platform. It introduces the process of fault recovery, which allows a program to resume its execution after encountering a specific fault type. Finally, it brings you the convenient troubleshooting applications available for your iPhone and iPad.

Appendix, *Rockwell Automation Literature Library Resource*, gives you topic-specific documentation links.

Safety warning – loss of control/view

The designer of any control scheme must consider the potential failure modes of control paths and, for certain critical control functions, provide a means to achieve a safe state during and after a path failure. The examples of critical control functions are emergency and over-travel stop that may include the following capabilities:

- Separate or redundant control paths must be provided for critical control functions.
- System control paths may include communication links. Consideration must be given to the implications of unanticipated transmission delays or failures of the link.
- Each implementation of a control system must be individually and thoroughly tested for proper operation before being placed into service.
- Failure to follow these instructions can result in death, serious injury, or equipment damage.

This book is not comprehensive for any systems using the given architecture. It does not absolve users of their duty to uphold the safety requirements for the equipment used in their systems or compliance with both national or international safety laws and regulations.

What you need for this book

In order to complete the exercises in this book, you will need RSLogix 5000 version 17+ or Studio 5000 Logix Designer version 20+. You will also need a Rockwell ControlLogix Programmable Automation controller or a software controller such as Emulate 5000 / SoftLogix 5800.

Who this book is for

The purpose of this book is to explore the hardware, software, and programming of the Logix platform so that electricians, instrumentation techs, automation professionals, and students who are familiar with automation, get up to speed with a minimal investment of time. I intentionally focus on the essential requirements for selecting, configuring, and programming a modern Logix application in order to get the reader working with the platform as quickly as possible. Once the reader has a solid foundation in Rockwell Automation Integrated Architecture, they will be able to further their knowledge on any topic using the online Literature Library.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This module can be configured to record or latch the time at which a state is changed from `ON` to `OFF`, `OFF` to `ON`, or both."

A block of code is set as follows:

```
(*****  Alarm Totalizer  *****)
if (OSRI_01.OutputBit AND NOT FC1001_FLT_ALM.Disabled) then
    Total_Alarm_Count := Total_Alarm_Count + 1;
end_if;
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "First, in the **Controller Organizer** pane, select and double-click on the **Controller Tags** option to open the **Controller Tags** panel."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
ControlLogix and CompactLogix Overview and Firmware

In this chapter, we will introduce the **ControlLogix** and **CompactLogix** platforms by exploring the evolution of the **Allen-Bradley** controllers. We will provide details of the **Rockwell Automation Integrated Architecture** and then finally, we will discuss the important role that firmware plays in the **Logix5000** platform. Due to 15 to 20 years of industrial controller life span, it is common to encounter older versions of hardware and firmware, and critical to be familiar with legacy systems.

A brief history of Rockwell Automation

This book begins with some background history on the Rockwell Automation ecosystem. It is important to understand the legacy systems provided by Rockwell Automation because some of them can still be found operating in the field today. Also, it is important to understand the overall Rockwell Automation offering and terminology, and how the platforms we focus on in the book fit into the real world.

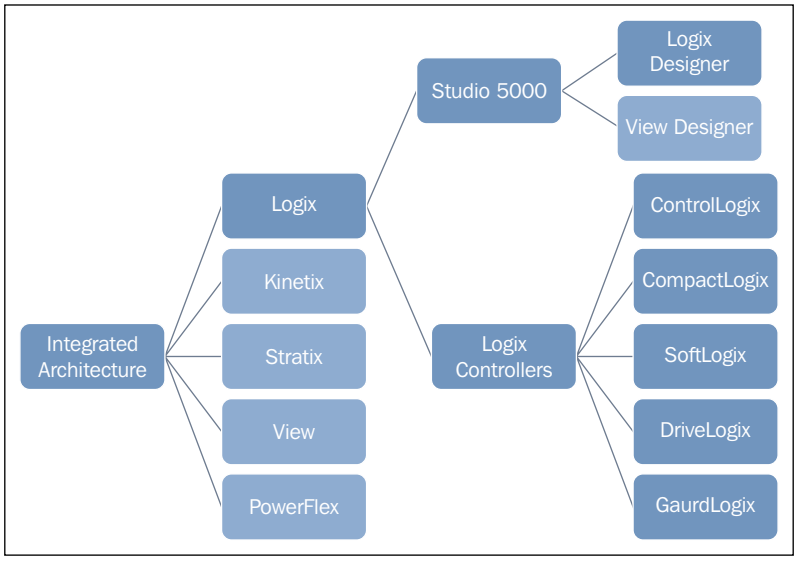
Allen-Bradley was founded in 1904 by brothers, Harry (19 years old) and Lynde Bradley (26 years old), with seed money from Dr. Stanton Allen. As a teenager, Lynde Bradley developed the prototype for what would later become Allen-Bradley's first commercial product. The primary focus of Allen-Bradley was motor controllers for several decades until they received an unusual challenge from **General Motors** (**GM**) in 1968. Each time GM wanted to introduce a new car, they needed to spend two or three months rewiring all their relays to support the production process changes. The request was to build a system to replace their hard-wired relay logic with something more dynamic—**Standard Machine Controller**. Modicon ultimately won the GM contract with their highly robust Modicon 084 Controller. As a result, Allen-Bradley acquired a company called Information Instruments Inc and produced their first functional controller—**Programmable Matrix Controller** (**PMC**) in 1971. Shortly after the release of PMC, Allen-Bradley released a more feature-rich product known as **Programmable Logic Controller 1** (**PLC-1**). Since the introduction of the first Allen-Bradley (later, Rockwell Automation) PLC-1, we have seen several platforms released, including PLC-2 (1978), PLC-3 (1981), PLC-5 (1986), SLC 500 (1991), MicroLogix (1994), ControlLogix (1997), and finally, CompactLogix (2006). In 1985, Allen-Bradley was acquired by Rockwell International and was later spun off as a part of Rockwell Automation. In the field today, the Allen-Bradley name and logo can still be seen on many of the Rockwell Automation's products. The focus of this book will be on the modern ControlLogix and CompactLogix controllers and **Studio 5000 Automation Engineering and Design Environment**, which I will refer to as the Logix family.

Integrated Architecture

Like many other vendors, Rockwell Automation has recently rebranded and reorganized their offering. The ControlLogix family is a part of Rockwell Automation's larger solution offering called Integrated Architecture. It is a relatively new term in the world of Rockwell Automation, but the concept has been in place for quite some time. It represents a convergence of the control and information systems within an industrial operations environment. This convergence is in line with the industry trend we have witnessed over the past decade and has increased the ties between **Operational Technology** (**OT**) and traditional **Information Technology** (**IT**). We have seen a continuous increase in demand for operational information to be provided to the corporate information system in real time in order to fulfill the maintenance needs, environmental reporting, accounting, and other corporate requirements. At the same time, we have seen OT move from proprietary protocols and data access technology to traditional IT technologies such as TCP/IP and Ethernet. The promise of Integrated Architecture is the ability to easily implement plant-wide optimization, reduce technical project risk, increase machine performance, and improve long-term reliability.

The five core technologies of Integrated Architecture **Programmable Automation Controller** (**PAC**) product line include the following platforms:

- ControlLogix
- CompactLogix
- GuardLogix
- DriveLogix
- SoftLogix



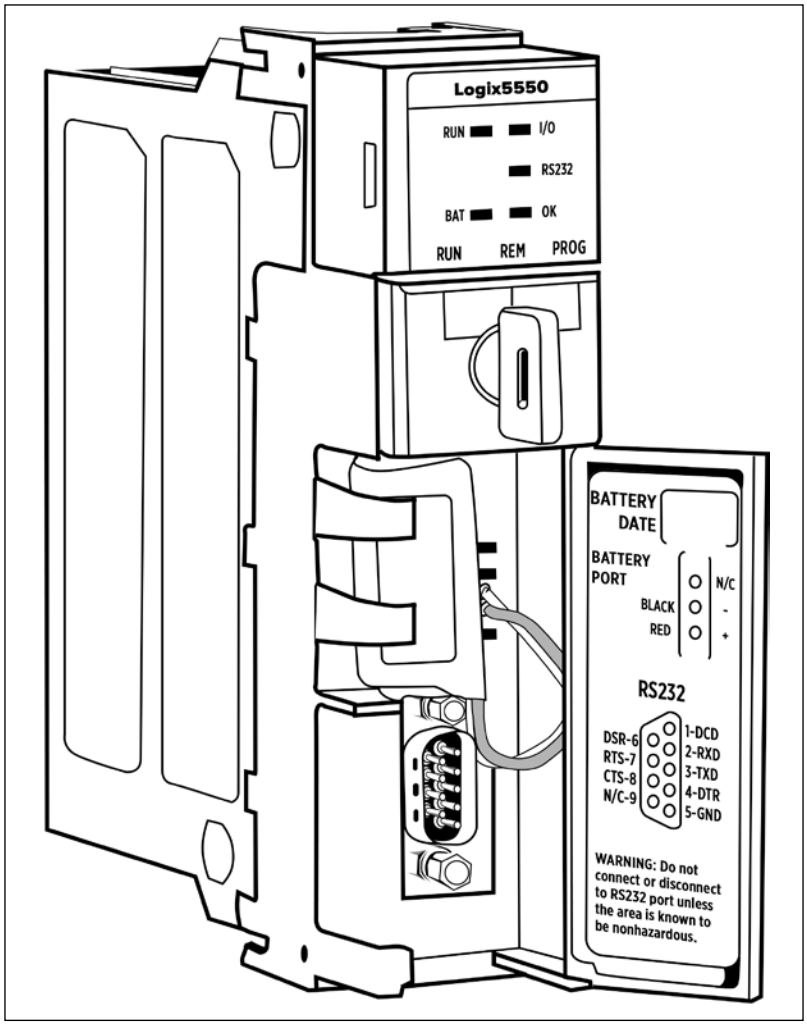
Rockwell Automation Integrated Architecture overview

The preceding diagram outlines the Integrated Architecture structure and shows where ControlLogix fits into the mix. The **FlexLogix** (bulletin 1794) controllers were also part of the Logix PAC family and was used to communicate with PLC-5 and SLC 500 Flex I/O blocks. However, FlexLogix has now been retired from the lineup, so it will not be covered in this book.

The product, formally known as **RSLogix 5000** (used for programming the ControlLogix and CompactLogix controllers), is now included within the automation engineering and design software suite called Studio 5000 and is now referred to as **Logix Designer**. For the remainder of this book, we will be using the terms—Logix Designer, RSLogix, and Logix—interchangeably to refer to the Logix controller family programming environment.

ControlLogix controllers

ControlLogix controller was first launched in 1997 as a replacement for Allen-Bradley's previous large-scale control platform, PLC-5. The ControlLogix platform includes a bulletin 1756 ControlLogix 5550 controller, bulletin 1756 ControlLogix I/O modules, and the RSLogix 5000 programming software platform (now referred to as Studio 5000 Logix Designer). ControlLogix represented a significant technological step forward that included a 32-bit ARM-6 RISC-core microprocessor and an ABrisc Boolean processor combined with a bus interface on the same silicon chip. At launch, the series 5 ControlLogix (also referred to as **L5** and **ControlLogix 5550**) controllers were able to execute the code three times faster than PLC-5. The following diagram is an illustration of the original Logix L5 controller:



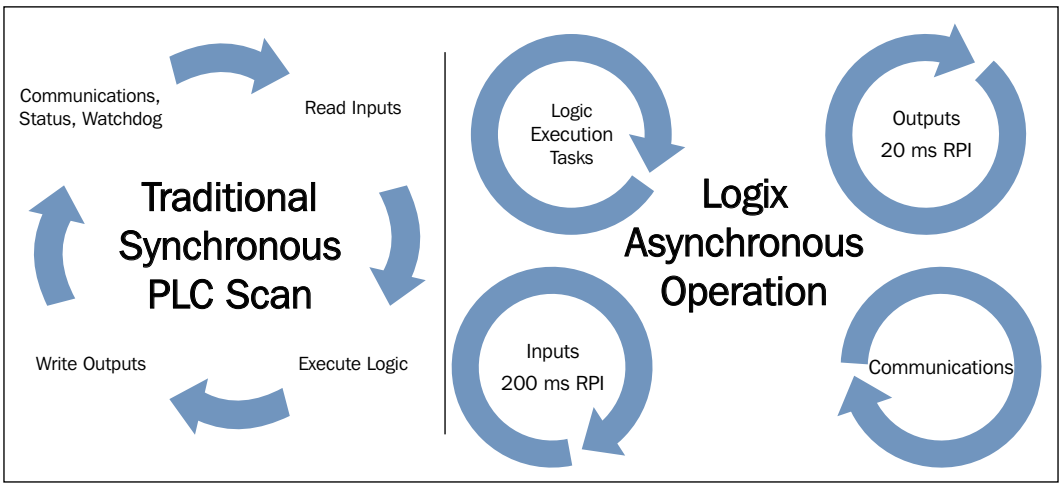
The L5 controller is considered to be a PAC rather than a traditional PLC due to its modern design, power, and capabilities beyond a traditional PLC (such as motion control, advanced networking, batching, and sequential control). The ControlLogix platform is built on the **ControlBus** backplane, which performs like a mini-network and allows devices to be **Removed or Inserted Under Power** (**RIUP**).

Warning: Removing modules while under power can create an arc and have disastrous consequences in explosive environments.

L5 has since been retired from the lineup, so we will focus on the newer **L6** and **L7** controllers in this book. Throughout this book, we will be referring to the ControlLogix controllers as PACs, which are the modern day equivalent of PLCs.

Logix operating cycle

The entire Logix family of controllers (ControlLogix and CompactLogix) has diverged from traditional synchronous PLC scan architecture in favor of a more efficient asynchronous operation. Like most modern computer systems, asynchronous operation allows the Logix controller to handle multiple tasks at the same time by slicing the processing time between each tasks. The continuous updating of information in an asynchronous processor creates some programming challenges, which we will address throughout the book. The following diagram illustrates the difference between the synchronous and asynchronous operation:



ControlLogix series 6 controllers

In 2002, the bulletin 1756 ControlLogix L6 (Logix556x) processor was released with a more powerful processor and more memory, and the **CompactFlash** nonvolatile memory card was added to the entire lineup.

Even though the ControlLogix platform is approaching its 20th birthday, it is still in the early stages of its product life cycle. For example, Allen-Bradley's 1747 series SLC500 family, which was introduced in 1989, is still available for sale today. Although no longer actively being developed, SLC500 represents a product life in excess of 25 years.

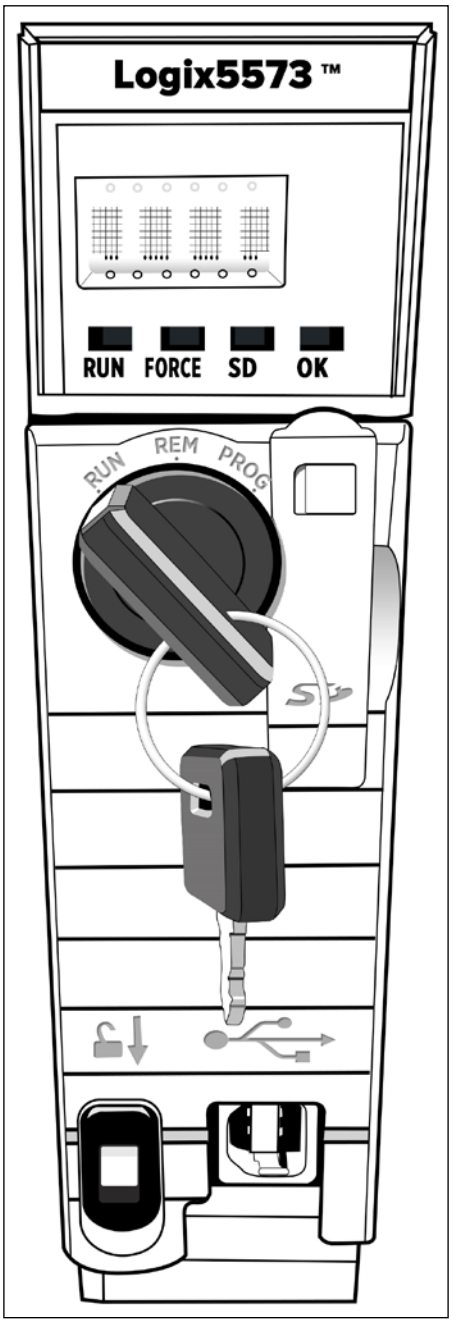
ControlLogix represents a common control engine with a common development environment and tight integration between the programming software, controller, and I/O modules. This close integration greatly reduces automation engineering development time and cost.

ControlLogix series 7 controllers

In 2010, Rockwell Automation launched the series 7 (also referred to as **L7** and **ControlLogix 5570**) controllers, which featured the following enhancements over the series 6 (L6) controllers:

- The performance capability doubled due to a more powerful dual core CPU.
- The adoption of modern SDRAM memory.
- The replacement of the 9-pin serial port with a USB 2.0 port (programs transfer 200 times faster over USB 2.0 than serial).
- The replacement of the CompactFlash memory card with a **Secure Digital** (**SD**) memory card.
- The replacement of the lithium battery with the capacitor-based **Energy Storage Module** (**ESM**). The ESM provides power to the controller during a power loss event to allow it to copy the contents of its memory from volatile memory to the onboard nonvolatile memory. The ESM eliminates the issue with L6 series controllers that would lose the program after a few weeks without power once the battery was completely drained.
- The ability to store program comments and tag descriptions on the controller (firmware v21 and higher).

- The addition of the onboard four character display.



ControlLogix L73 controller

Selecting a ControlLogix controller

When selecting a ControlLogix controller, it is important to consider the following points:

- Supported Logix Designer software versions
- Processing the requirements of your current application and future expansion
- Memory requirements of your current application and future expansion

The ControlLogix series 6 and series 7 controllers and their software version compatibilities are shown in the following table:

ControlLogix controllers		**Logix Designer software (RSLogix 5000)**	
Controller	**Memory**	**Minimum version**	**Maximum version**
Series 6 (L6)			
1756-L61	2 MB	v12	v20
1756-L62	4 MB	v12	v20
1756-L63	8 MB	v10	v20
1756-L64	16 MB	v16	v20
1756-L65	32 MB	v17	v20
Series 7 (L7)			
1756-L71	2 MB	v18	
1756-L72	4 MB	v19	
1756-L73	8 MB	v18	
1756-L74	16 MB	v19	
1756-L75	32 MB	v20	

It is important to note that the L6 controllers are not supported in Version 21 and higher of Studio 5000 Logix Designer.

GuardLogix safety controllers

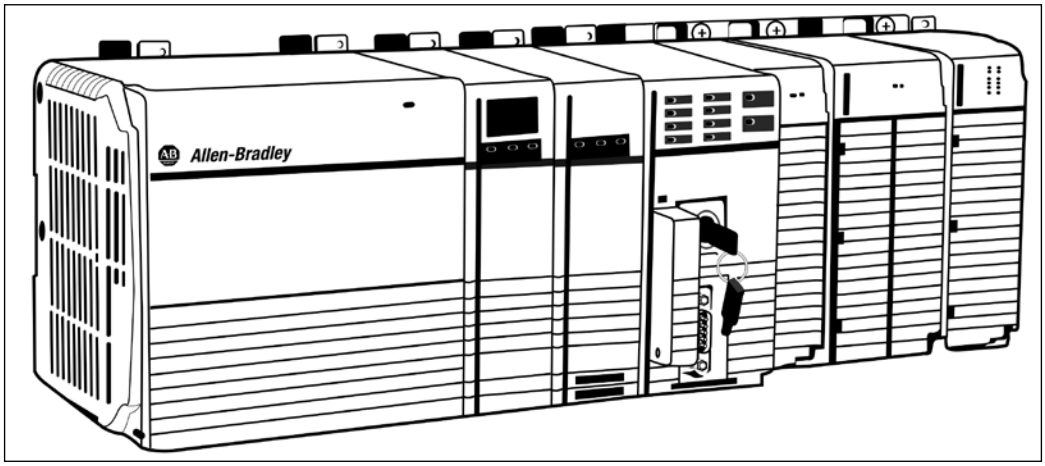
With the launch of the (bulletin 1756) GuardLogix controller in 2005, the ControlLogix platform supported both standard and safety system control in the same chassis. The GuardLogix controller system is designed for use in safety applications, including SIL 3 (IEC 61508) and the ISO standard for Safety of Machinery (ISO 13849-1 General Principles for Design and PLe/Cat.4). GuardLogix safety controllers represent an essential piece of a fail-safe (de-energize to trip) solution. Fail-safe refers to a solution that when a fault is detected, all of its outputs are set to zero. And, in the event of a faulty input or input module, it automatically sets any input values associated with them to zero. Both the L6 and L7 controllers are available in the GuardLogix form factor. Physically, the GuardLogix controllers feature a red faceplate and are usually installed in pairs—primary and safety partner controller. The GuardLogix controllers are only supported in Version 18 and higher of RSLogix 5000 and Studio 5000 Logix Designer.

Extreme environment controllers

The Rockwell Automation's extreme environment controllers (bulletin 1756 **ControlLogix-XT**) share the same features and programming interfaces as the standard ControlLogix controllers, but are certified to operate in extreme conditions. The ControlLogix-XT modules are darker gray in color than the ControlLogix modules and are spaced in every other slot to provide an improved ventilation/isolation. In addition, the ControlLogix-XT modules are treated with a conformal coating that improves the product's resistance to corrosive environments. The ControlLogix-XT controllers and modules are rated for temperatures ranging from -20°C to 70°C (-4°F to 158°F) and have the following environmental certifications—cULus, Class 1, Div 2, C-Tick, CE, ATEX Zone 2, SIL 2, IEC 61131-2, ANSI-ISA-S71.04-1985, Class G1, G2, and G3. The L6 and L7 standard controllers and GuardLogix controllers are all available in **Extreme Environment** (**XT**) form factors.

CompactLogix controllers

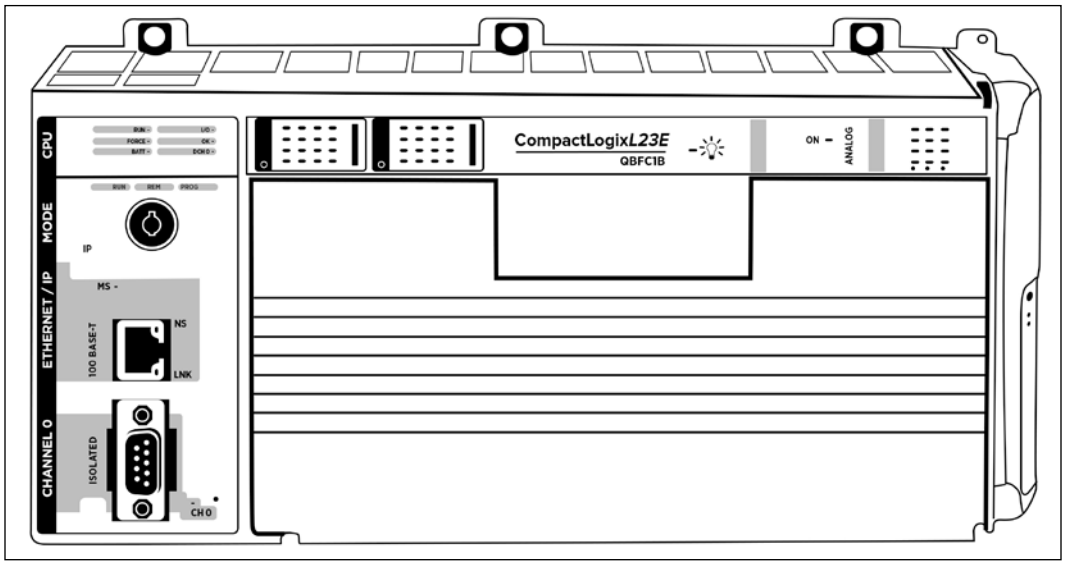
In 2006, Rockwell Automation first shipped the (bulletin 1768) L43 CompactLogix controllers targeted at cost effective, small- to medium-size automation solutions. At the time of launch, CompactLogix controller was planned as the long-term replacement for the SLC 500 controller family. The CompactLogix control platform is designed with an emphasis on the controller software. As the CompactLogix hardware evolves with an improved performance and additional features, the logic will easily migrate to new hardware and firmware versions. Unlike the SLC 500 platform, the CompactLogix controllers can be programmed using the same RSLogix 5000 (Logix Designer) software suite that is used with ControlLogix. In 2006, CompactLogix L43 with integrated motion support was added to the family. It features a CompactFlash memory card, Ethernet port, Serial RS-232 port, 1769 / 1768 modules, and a power supply module. The following is an illustration of the L43 CompactLogix controller:



CompactLogix controller-bulletin 1768—L43 and L45

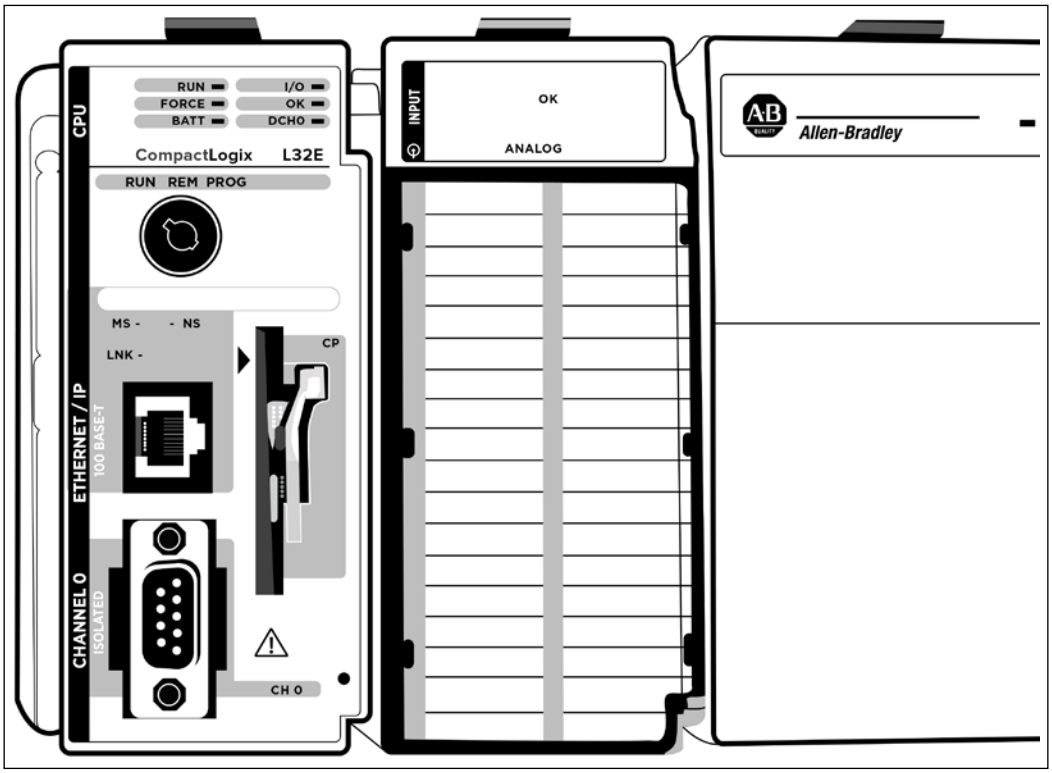
Modules on L43 can only be placed to the right of the power supply.

In 2008, Rockwell Automation released the low-cost CompactLogix L23 controllers (bulletin 1769) with embedded I/O. The L23 controller features a serial RS-232 port, Ethernet port (only on the E models), embedded I/O, and an embedded power supply. The following is an illustration of an L23 controller:



CompactLogix controller-bulletin 1769 L23x Packaged controllers with embedded I/O

Also in 2008, Rockwell Automation released the (bulletin 1769) CompactLogix L3x modular controllers. The 1769 CompactLogix modules do not have a chassis like the ControlLogix modules. The 1769 CompactLogix modules can be connected together using a DIN rail or can be screwed in directly to a panel. CompactLogix L3x features a CompactFlash memory card, serial RS-232, ControlNet or Ethernet port, and a power supply module. The following diagram is an illustration of the CompactLogix L3x controller:



CompactLogix Controller-Bulletin 1769-L3x Modular controllers

The L3x modules can be placed to the left or the right of the power supply.

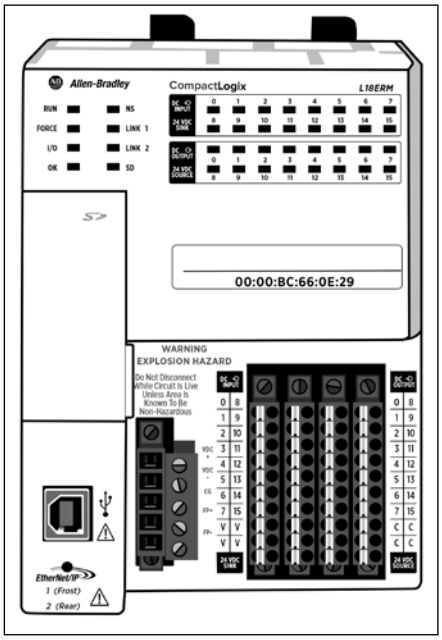
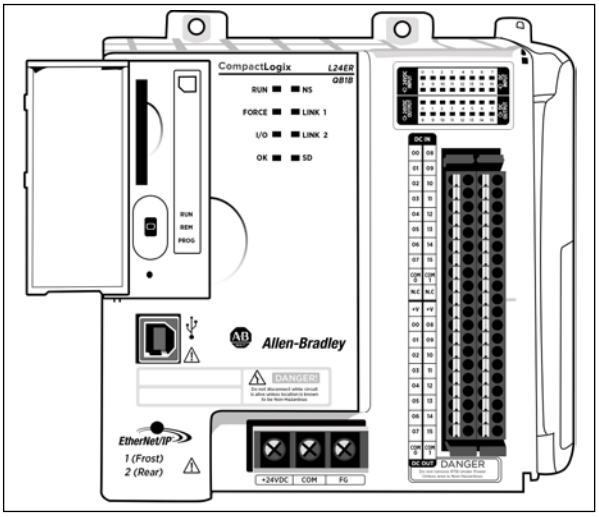
In 2009, Compact GuardLogix, an SIL3 certified controller, with the L43S and L45S CPU supporting integrated safety, was added to the Logix family.

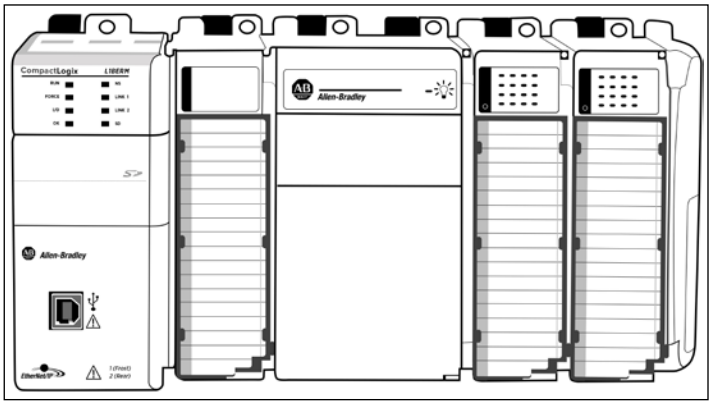
CompactLogix 5370 controllers

In 2012, Rockwell Automation released the (bulletin 1769) CompactLogix 5370 L1, L2, and L3 controllers, which provided a low-cost Ethernet/IP-enabled, high-performance controller in a 40 percent smaller form factor than ControlLogix. The CompactLogix 5370 series controller provides many of the same enhancements that the ControlLogix series 7 provided over the ControlLogix series 6 controllers, including the following properties:

- Twice the performance capability due to a more powerful dual core CPU
- Adoption of modern SDRAM memory
- Replacement of the 9-pin serial port with a USB 2.0 port (programs transfer 200 times faster over USB 2.0 than serial)
- Replacement of the CompactFlash memory card with an SD memory card
- Added the ESM and removed the need for a lithium battery
- Made use of the existing CompactLogix 1769 I/O modules
- Integrated motion control over Ethernet
- Ability to store program comments and tag descriptions on the controller (firmware v21 and higher)

The following table provides illustrations of the CompactLogix 5370 controllers and their distinguishing features:

CompactLogix Controller—bulletin 1769 5370—L1	**Features**
 CompactLogix 5370 L1 Controller	SD memory card 2 X Ethernet ports USB 2.0 port Embedded point I/O modules Expandable with 6 or 8 point I/O modules Embedded power supply Integrated motion control
CompactLogix Controller—bulletin 1769 5370—L2	**Features**
 CompactLogix 5370 L2 Controller	SD memory card 2 X Ethernet ports USB 2.0 port Embedded 1769 I/O modules Expandable with 4 x 1769 I/O modules Embedded power supply Integrated motion control

CompactLogix Controller—bulletin 1769 5370—L3	Features
 CompactLogix 5370 L3 Controller	SD memory card
	2 X Ethernet ports
	USB 2.0 port
	8 to 30 1769 I/O modules
	Power supply module

Selecting a CompactLogix controller

There are many factors to consider when selecting a CompactLogix controller due to their module nature and wide range of form factors which are available:

- Supported Logix Designer software versions
- Cabinet size restrictions
- CompactLogix form factors or I/O module scalability
- Processing the requirements of your current application and future expansion
- Memory requirements of your current application and future expansion

The CompactLogix controllers and their software version compatibilities are shown in the following table:

CompactLogix controllers		Logix Designer software (RSLogix 5000)	
Controller	**Memory**	**Minimum version**	**Maximum version**
Bulletin 1768			
1768-L43	2 MB	v16	v20
1768-L45	3 MB	v16	v20
Bulletin 1769 L23x Packaged controllers with embedded I/O			
1769-L23	512 KB	v16	v20
Bulletin 1769-L3x Modular controllers			
1769-L3x	1.5 MB	v16	v20

CompactLogix controllers		Logix Designer software (RSLogix 5000)	
Controller	**Memory**	**Minimum version**	**Maximum version**
Bulletin 1769 5370			
5370 1769-L16	384 KB	v20	
5370 1769-L18	512 KB	v20	
5370 1769-L24	750 KB	v20	
5370 1769-L27	1 MB	v20	
5370 1769-L30	1 MB	v20	
5370 1769-L33	2 MB	v20	
5370 1769-L36	3 MB	v20	

It is also important to consider that some of the CompactLogix 5730 controllers are slated as direct replacements for some of the older CompactLogix controllers (although the older controllers are still available for purchase):

- 5370 1769-L24 replaces 1769-L23
- 5370 1769-L3x replaces 1769-L3x

ControlLogix software and firmware

Due to the long life span of most industrial PACs, it is common to encounter controllers still running legacy firmware. Controller firmware versions and RSLogix 5000 and Logix Designer versions go hand in hand. If you are working on the ControlLogix or CompactLogix controller that is running firmware version 13.03, you should be using RSLogix 5000 Version 13.03 to program it. As updating firmware can introduce process downtime, it is important to understand and work with the capabilities of older firmware and software versions:

Version	Year	Notes
1	1997	Cross reference support, RSLinx Version 2.0 support, L5x
2	1998	Trending, position and time camming, 1794 FLEX I/O, RSWho
3,4	1998	Internal builds, not released to the public
5	1998	SERCOS, quick view pane, function block diagrams, FLEX EX
6	1999	FlexLogix and SoftLogix support
7	2000	Windows 2000 support, CompactLogix support, Ethernet/IP support
8	2001	ControlLogix redundancy, DH485, nonvolatile memory L55
9	2001	SERCOS Drive support with 1756-M08SE module
10	2002	ControlLogix 5563 controller support

Version	Year	Notes
11	2002	SFC, ST, FBD online edits SoftLogix 5800, point I/O support
12	2003	RSLogix Emulate 5000, event task, CompactLogix support, compare
13	2004	SFC online editing, ST online editing, LD import/export
14	2004	GM only build
15	2005	S88, add 1756 I/O modules during runtime, **user-defined data type** (**UDT**)
16	2007	User-defined **add-on instructions** (**AOI**), ControlLogix 1756-L64
17	2008	Windows Vista, free to download demo, advanced process control
18	2010	1756-L73, 1756-L75 controller, CIP motion, CIP SYNC, CompactLogix safety
19	2010	Windows 7 support, 1756-L72, 1756-L74, integrated motion Ethernet/IP
20	2012	1756-L71, support 200 to 10,000 I/O points, GuardLogix
Studio 5000—Logix Designer		
21	2013	Logix Designer, alarm log, comments and descriptions stored in PAC
22	2014	Internal build, not released to the public
23	2014	Controller firmware updates and fixes
24	2014	Windows 8 support, logical organizer view, program parameter, merge improved
25	2015	Internal build, not released to the public
26	2015	Windows 8.1 support, license-based source protection

Product Selection Toolbox

Rockwell Automation provides a software suite called **Product Selection Toolbox**, which is designed to help you select and design Integrated Architecture solutions. This software suite provides helpful tools for evaluating the size of your application, generating drawings, and even estimating the cost of your application. This product is available for free to approved partners and customers.

Rockwell Automation Product Catalog for iPad

Rockwell Automation has created an iPad-based product selection tool. Rockwell Automation **Product Catalog** is a portable version of **Product Selection Toolbox** that allows you to select and configure thousands of products from Rockwell Automation and their industry partners. Product Catalog will even help you find the nearest distributor to your location. It is available for free in the App Store.

Summary

In this chapter, we learned about the controllers available within Rockwell Automation's Integrated Architecture. We also explored the history of Rockwell Automation and evolution of the industry-leading Logix platform. We now have an idea of the controller solutions available within Integrated Architecture, and are capable of making basic solution architecture decisions. In the appendix of this book, you can find links to **Rockwell Automation Literature Library** where you can dive deeper into the topics covered in this chapter.

In the next chapter, we will introduce the various networking and communication options available for the Rockwell Automation Logix controllers.

2
Industrial Network Communications

In this chapter, we will discuss the various communication technologies available for the Logix platform. The focus of this book is the current state of Rockwell Automation's **ControlLogix** and **CompactLogix** controllers; however, we will touch on some legacy communication protocols, which you may still find running in the field today. Communications allows us to interface with controllers, racks, and devices on our network. Establishing communications is an important step that enables us to connect with a device and transfer configuration changes and programs. After completing this chapter, you will be familiar with all the Rockwell Automation communication technologies that are actively used in the field today. This chapter will cover the following key areas of industrial network communications in the Logix platform:

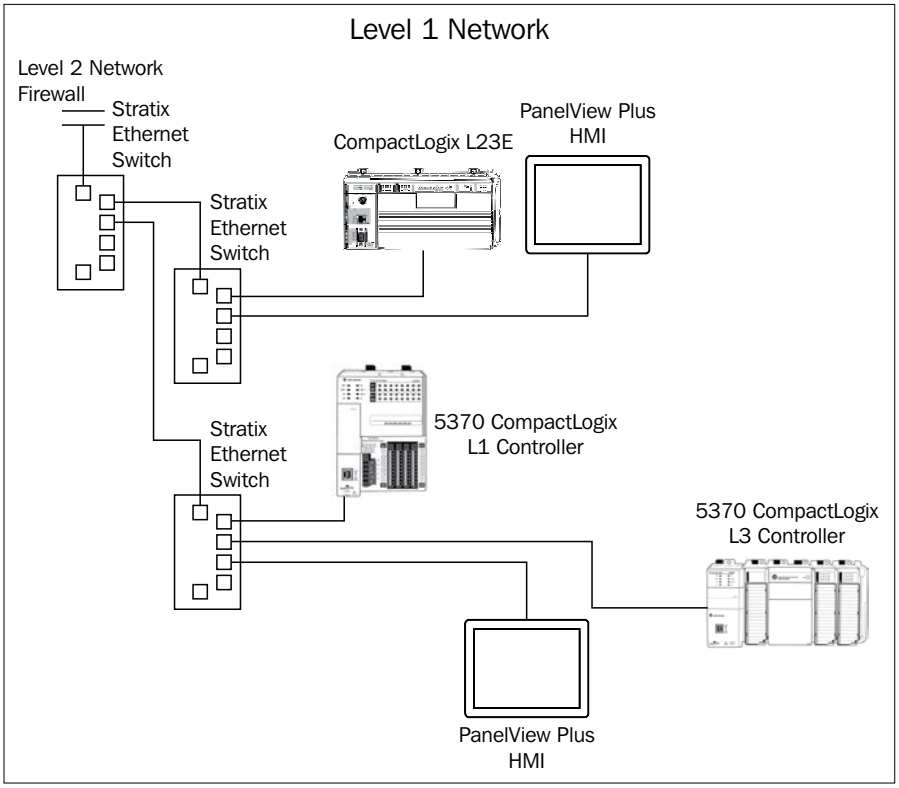
- Key terms in industrial communications
- Primary network technologies used today
- Legacy network technologies you may encounter in the field
- EtherNet/IP Capacity Tool
- RSLinx
- Rockwell Automation Integrated Architecture Builder mobile app

Key terms in industrial communications

Communications enables the exchange of data between devices on an industrial control network. Without communications, controllers are unable to see the values coming from field devices, and operators are unable to see the values coming from controllers. Here are some of the key terms we will use while talking about industrial networking and communications:

- **Media**: Communication requires wires, cables, or fiber optic connections between devices. The connection used for communication is called the media, which differ from one another by properties such as the maximum distance for each connection and maximum data transmission speed.
- **Node**: Each device on the network is called a node.
- **Node Address**: Each node will typically have a unique identifier called a node address.
- **Network**: When multiple devices are connected together under a common device address space, we call this a network.
- **Topology**: The physical structure of the network is called the topology.
- **Bridge**: This device creates a connection between two separate networks.
- **Router**: This device can forward data between two or more networks.
- **Hub**: This device can channel data between multiple nodes within a single network.
- **Switch**: This device can channel data from one node to another. Unlike a hub, a switch will intelligently route only the data destined for a device and is not prone to data loss due to network packet collisions.
- **Segment**: This device divides a localized section of the network with bridges, routers, or switches.
- **Protocol**: The language used to communicate data between each node across a network is called a protocol.

The following diagram illustrates each of the preceding industrial networking terms:

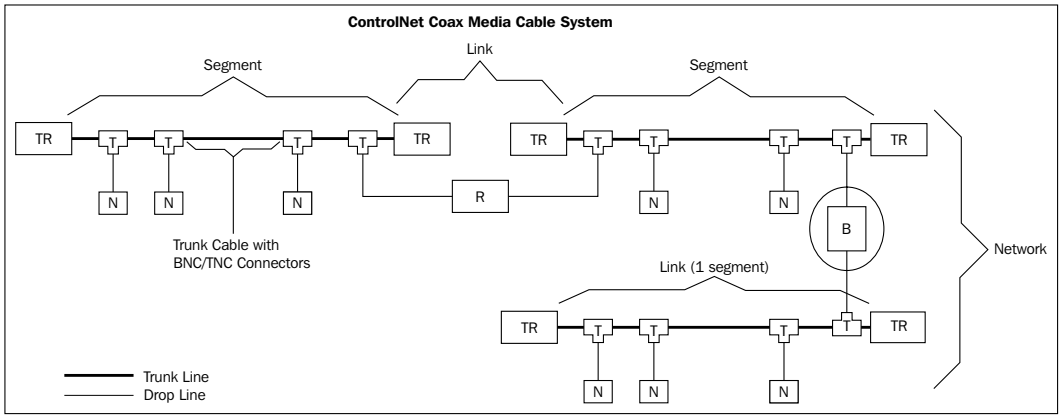


In networks like ControlNet, which use coax cable as a media, you will find the following terminology:

- **Tap**: This device is small and T-shaped, and connects a Trunk line cable (the top of the T) to a Drop line cable (the bottom of the T).
- **Terminating resistor**: This device is a 75 Ohm resistor that can be connected to a trunk-line connection that absorbs energy and prevents electrical signal reflection.
- **Trunk line**: This device is the main cable that connects to the taps and drop line cables. A trunk line will have a terminating resistor at either side and one or more drop lines attached to it.

- **Drop line**: This device is a cable connection from a tap that connects down to a node.
- **Repeater**: This device is a two-port device, which connects two trunk lines together and boosts the signal. It is helpful for connecting segments across longer distances or boosting the signal for a long network segment.

The following diagram illustrates a typical coaxial cable network:



Now that we have covered some of the basic terms, let's look at the various communication solutions used by Rockwell Automation.

Network communication technologies

Communications play a vital role in most industrial automation solutions, and Rockwell Automation's Integrated Architecture offers a variety of options. I have classified these options as primary network technologies (those that are actively used in modern implementations of a Logix system) and legacy network technologies (those that have been used frequently in the past but are not being installed in the new Logix control systems).

Primary network technologies

Today, Logix relies on the following three primary technologies for network communication:

- DeviceNet
- ControlNet
- EtherNet/IP

Let's discuss each of them in detail.

DeviceNet

DeviceNet is designed specifically to communicate with and provide power to the lowest-level field devices. It can communicate with intelligent devices that support a DeviceNet module such as sensors, VFDs, valves, motors, and distributed I/O blocks. DeviceNet is convenient as it provides both signal and power to the device. However, it's not designed for handling high volumes of network traffic. The DeviceNet networks must be configured using Rockwell Automation's RSNetworx for DeviceNet. In this book, we will be primarily focusing on EtherNet/IP networking, but more DeviceNet resources are available in the appendix.

ControlNet

ControlNet is a deterministic network technology that acts as the I/O communication backbone for a control system. Deterministic data collection guarantees that new data will arrive within a predefined interval. It enables the interconnection of multiple DeviceNet networks and shares data with other controllers. It is capable of full network redundancy, so if a redundant ControlNet network is configured and one cable is broken, the network can continue operating normally.

Connections over ControlNet are configured as **scheduled** or **unscheduled**.
The **ControlNet Network Update Time** (**NUT**) is the millisecond interval for collecting updated data in a ControlNet network. ControlNet is a highly repeatable deterministic communications method, albeit slow (locked at 5 Mbits/s) but deterministic nonetheless. Deterministic networks are ideal for processes such as motion control that require continuous synchronized data and cannot tolerate any data update delays. The ControlLogix backplane ControlBus is nearly identical to the ControlNet networking.

It is based on an open standard for industrial network protocols known as **fieldbus** and is based in the same fieldbus **IEC 61158** communication standard as **Foundation Fieldbus**, **PROFIBUS**, and **Interbus**. The fieldbus architecture provides the skeleton that ControlNet is based upon. However, due to fundamental implementation differences, it is not able to directly communicate with any other fieldbus protocols. On top of the fieldbus architecture, at the protocol application layer, ControlNet uses the **Common Industrial Protocol** (**CIP**) to provide its functionality. ControlNet can upload and download programs, perform I/O forcing and online editing, and communicate with the remote I/O racks.

ControlNet uses **Quad Shield RG6 Coaxial Cable** as a networking media, which requires it to use network taps for each drop, and terminating resistors at the end of each segment. In this book, we will be primarily focusing on the EtherNet/IP networking, but more ControlNet resources are available in the appendix.

EtherNet/IP

EtherNet/IP is the most widely used communications technology in the Integrated Architecture ecosystem today because of its speed, scalability, and ease of integration with enterprise-level network hardware. EtherNet/IP will be the primary communication method used in the examples found in this book. It combines the IEEE 802 standard Ethernet technology stack with the object-based CIP. Basing the communications on Ethernet allows ease of integration with the existing enterprise IT networks. And, the CIP application layer protocol allows the Logix controllers to communicate control, safety, synchronization, motion, configuration, and diagnostic information with devices from hundreds of different vendors. CIP enables EtherNet/IP to upload a program, download a program, force I/O values, monitor code, perform online edits, and connect to the remote I/O racks.

In the legacy I/O systems, the PLC would poll (request data at a set interval) digital input modules for new data. The CIP protocol on EtherNet/IP digital input modules can perform the following tasks:

- Return data on **Change of State** (**COS**)
- Return data at a **Request Packet Interval** (**RPI**) scheduled in milliseconds

One notable difference with EtherNet/IP is that this method of data collection is nondeterministic. Deterministic in the context of communications means the delays in delivering a packet of data across a network are known in advance and are not subject to change. Nondeterministic data collection does not guarantee that the new data will arrive within the RPI. IEEE 802 standard Ethernet is fundamentally nondeterministic as it is not scheduled within a set time window. However, the speed of EtherNet/IP with a modern full duplex switch almost negates this fact.

EtherNet/IP uses **User Datagram Protocol** (**UDP**) to communicate the basic I/O and non-critical information on the network. UDP does not perform any error checking or handshaking mechanisms, so the delivery of information is not guaranteed, and the data is susceptible to any network-related data loss. UDP is ideally suited for real-time information as it trades off speed for guaranteed data delivery. In real-time systems, dropped packets are preferable to waiting for delayed packets. It is far better to get the next most recent value than circling back for a packet that is already stale. Using UDP, EtherNet/IP is capable of collecting data by polling, cyclic, and change-of-state monitoring. EtherNet/IP makes use of UDP port number 2222.

Transmission Control Protocol (**TCP**) is used by EtherNet/IP for critical data such as writing set points, parameters, and recipes, and uploading and downloading programs. TCP has a built-in error checking and three-way handshake mechanism that insures that no packets are lost during the data transfer. TCP sacrifices data transfer speed for guaranteed delivery of information. EtherNet/IP makes use of TCP port number 44818.

Stratix is a line of industrial networking and security solutions from Rockwell Automation that has been engineered specifically for EtherNet/IP (and is based on the ubiquitous Cisco hardware platform). However, because EtherNet/IP is based on the IEEE 802.x standards, it is possible to use normal network switchgear with EtherNet/IP. Rockwell Automation recommends that you use robust industrial grade networking equipment with the Logix controllers. Furthermore, Stratix switches seamlessly integrate into the Logix platform and can easily provide health and status information as the native Logix tags.

Legacy network technologies

There are a few other communication technologies you may also encounter in a Rockwell Automation solution. Some are legacy networks, while others are solution-specific technologies:

- Data Highway Plus
- Universal Remote I/O
- Serial Real-time Communications System
- DF1
- DH-485

Data Highway Plus

Data Highway Plus (**DH+**) was used by the older controllers (such as PLC-2, PLC-3, PLC-5, and SLC 500) for networking. It was developed as a proprietary protocol in the late 70s by Allen-Bradley. There are Logix communication modules available that can connect to the DH+ networks and can save customers from having to rip and replace their older DH+ networks. DH+ supported remote programming and messaging over the network. Specifically, it allowed uploading and downloading programs, forcing values, monitoring, and online edits. It is common to encounter the DH+ networks still in the field today, but it is obsolete, so we will not cover it in detail in this book.

Universal Remote I/O

Universal Remote I/O (**RIO**) was also used by older controllers (such as PLC-2, PLC-3, PLC-5, and SLC 500) to communicate to the remote I/O chasses. There are Logix communication modules available that can connect to RIO and save the cost of replacing the RIO networks. Using a remote rack allows you to place the I/O modules closer to the actual devices with which they are communicating. The RIO rack was connected back to the main controller using a two-wire Belden 9463 cable, also known as **blue hose cable**. Although it is common to encounter the RIO networks in the field today, it is also obsolete, so we will not cover it in detail.

Serial Real-time Communications System

Serial Real-time Communications System (**SERCOS**) or IEC 61491 is a communication technology created for the real-time motion control. It provides high-speed serial communication over an electrical noise-immune fiber-optic cable, and is commonly used in the manufacturing industry. It was developed by an international consortium of companies called **Interest Group SERCOS**, which included Siemens, ABB, AEG, AMK, Robert Bosch, and Indramat. Logix communication modules are available that can communicate with the SERCOS devices for motion control. However, we do not touch on motion control in the book, so we will not explore this communication technology in detail.

SynchLink

SynchLink is a fiber-optic communication technology for communicating with the **PowerFlex700S** products. It is a streamlined protocol, focused on high-speed drive and motion control. It does not transmit diagnostic information and should be used in conjunction with a standard control network such as ControlNet or EtherNet/IP.

DH-485 and DF1

The **DH-485** and **DF1** networks are legacy serial technologies that provided communication for the PLC-5, SLC 500 and MicroLogix controllers, and **Human Machine Interface** (**HMI**) terminals and computers. There are third party communication modules available from ProSoft that allow DH-485 and DF1 to communicate with the Logix devices. Although it is common to encounter DH-485 and DF1 in the field today, they have both been retired and will not be covered in this book.

A comparison of network communications

The following table compares the media, maximum distances, maximum nodes, maximum speeds, and topologies of these communication technologies:

Protocol	Media	Distance	Nodes	Speed	Topology
EtherNet/IP	Ethernet CAN/CIP over IP	100 M/328 ft	Many	100 Mbits/s (1 Gbits/s)	Star, linear, and ring
ControlNet	Quad Shield RG-6 Coaxial Cable, CIP	1000 M/3280 ft	99	5 Mbits/s	Star, trunk line, drop line, tree, and ring
DeviceNet	4 wire – 2 signal, 2 power, and CIP	100 M/328 ft to 380 M/1246 ft	64	125 Kbits/s to 500 Kbits/s	Trunk line, drop line, and star
DH+	Twinaxial cable, peer-to-peer, and token based	Trunk line: 3050 M/10000 ft and drop line: 30 M/100 ft	64	57.6 Kbits/s half duplex	Daisy chain, trunk line, and drop line
RIO	Twinaxial cable and scanner based	Trunk line: 3050 M/10000 ft and drop line: 30 M/100 ft	32	230.4 Kbits/s	Trunk line and drop line
SERCOS	Fiber-optic serial based	250 M/820 ft	254	16 Mbits/s	Ring
SynchLink	Fiber-optic	250 M/820 ft	257	10 Mbps	Star, daisy chain, and ring
DF1	RS-485 serial	1219 M/4000 ft	255	19.2 Kbits/s	Trunk line and drop line
DH-485	RS-485 serial	1219 M/4000 ft	32	19.2 Kbits/s	Trunk line and drop line

Listed in the preceding table are the properties of each Rockwell Automation communication technology. However, there are fiber-optic converters for most communications technologies that can extend their max distances.

DeviceNet and ControlNet were developed by Rockwell Automation and are based on the CIP and maintained by **ODVA** (formally, **Open DeviceNet Vendors Association**). EtherNet/IP (note the capitalized N, and that the IP, in this case, stands for Industrial Protocol) was developed by ODVA in 2001 and adopted by Rockwell Automation.

EtherNet/IP Capacity Tool

Rockwell Automation has provided a tool called **EtherNet/IP Capacity Tool** that has been designed to help you calculate the resources required to support a control network. The tool takes a conservative approach to estimating the requirements of your network usage based on a few data points you provide it.

The EtherNet/IP Capacity Tool is used to measure the networking capacity of a single **Scanner Processor**. A Scanner Processor is either an EtherNet/IP module, such as the ControlLogix ENBT, or a Logix controller with a built-in EtherNet/IP port, such as the CompactLogix L32E. If a Scanner Processor is at maximum capacity, in most cases, an additional EtherNet/IP module or controller can be added to the control solution. A separate EtherNet/IP Capacity Tool report can be created for each Scanner Processor network.

When you calculate your capacity, the tool provides you with the available CIP connections, TCP connections, I/O **packets per second** (**PPS**), and HMI PPS. The EtherNet/IP Capacity Tool is designed to highlight potential design issues early in the network architecture process.

CIP connections are the real-time implicit (scheduled at a set RPI) UDP data connections. You have a limited number of CIP connections, which varies according to the Scanner Processor you select.

TCP connections are explicit (unscheduled request, response) data communications.

Here are a few common scenarios that use a CIP connection:

- Each I/O rack added to the network for reporting the optimized digital values will use a CIP connection
- Each analog I/O module added to a rack consumes an entire CIP connection
- Each produced tag and each consumed tag processed from another Logix controller consumes an entire CIP connection

Here are a few common scenarios that use the TCP connections:

- Each I/O rack uses a TCP connection (used for writing set points or setting digital values)
- Each controller uses a TCP connection (used for uploading and downloading programs)
- Each HMI uses a TCP connection (the HMI MSG statements between the Logix controllers)

I/O PPS is the volume of data moving through the network. There is a maximum amount of PPS that any particular Scanner Processor is capable of handling. The PPS value is proportional to the CIP and TCP connections, and EtherNet/IP Capacity Tool creates a conservative estimate of the traffic.

EtherNet/IP Capacity Tool is an easy way to learn the topology of industrial Ethernet network and experiment with the various network devices and understand the available communication modules. In the following exercise, we will demonstrate how to use EtherNet/IP Capacity Tool.

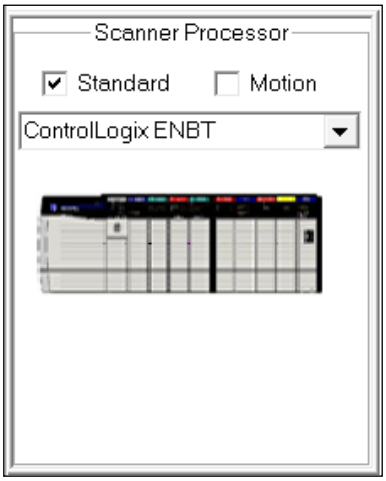
Using EtherNet/IP Capacity Tool

You will need to have a copy of the freely available Rockwell Automation EtherNet/IP Capacity Tool in order to complete the following exercise.

The EtherNet/IP Capacity Tool is on the annual release of Rockwell Automation toolkit. It can also be found in Integrated Architecture tools on the Rockwell website as a free download for approved customers and system integrators.

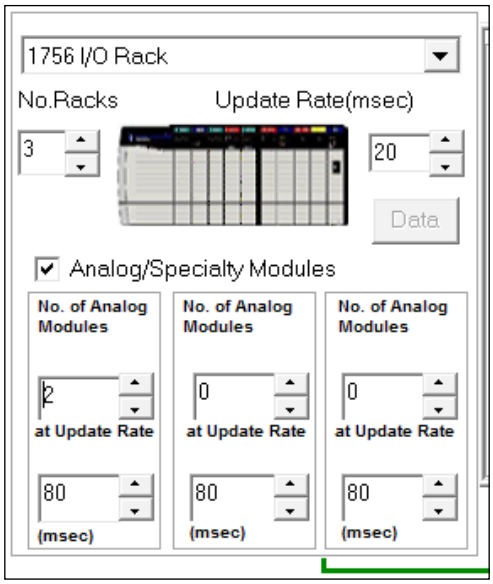
Let's begin with opening EtherNet/IP Capacity Tool and performing these steps:

1. Select the Scanner Processor for this exercise. In the center of the window you will see a group box with the label, **Scanner Processor**. Click on the drop-down box in the **Scanner Processor** group box and select the ControlLogix EtherNet/IP module's **ControlLogix ENBT** from the list:



2. Now, we will add some node group devices to our network and compute the network usage they add. Find the box that contains the **Node Group 2** label to the left of **Scanner Processor**. Next, we will add a ControlLogix I/O rack by selecting the **1756 I/O Rack** from the **Node Group 2** drop-down list.
3. Set the number of the ControlLogix I/O racks on our network by selecting **3** in the **No. Racks** numeric field.

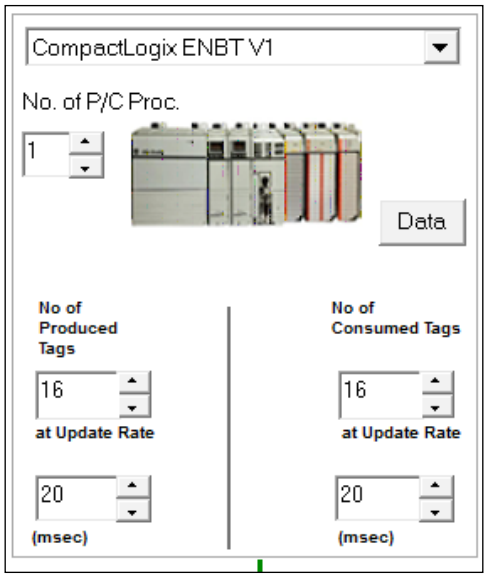
4. Click on the checkbox labeled **Analog/Specialty Modules** to indicate we have analog modules on our I/O rack. Add analog modules for each I/O rack by setting the **No. of Analog Modules** value in the first column to **2**, as shown in the following screenshot:



5. Now, let's hit the **Compute** button to see our current EtherNet/IP usage in the center of the screen.
6. Next, we will add a CompactLogix controller that we will communicate with using produced and consumed tags. The produced and consumed tags are a method of passing values from one controller to another, every RPI.

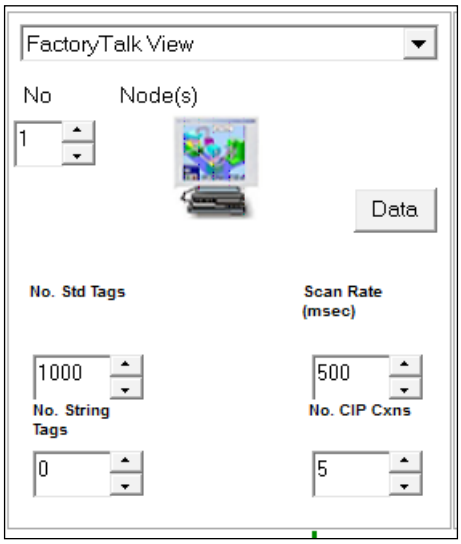
The produced and consumed tags allow a Logix controller to pass information back and forth. The produced tags are tags within a Logix program that are broadcast on a network for other controllers to receive. The consumed tags are tags within a Logix program received from other controllers.

7. In the **Node Group 1** box, select **CompactLogix ENBT V1**. Next, set the number of produced tags to **16**. Set the number of the consumed tag to **16** (32 is the maximum number of produced tags and consumed tags for our Scanner Processor):

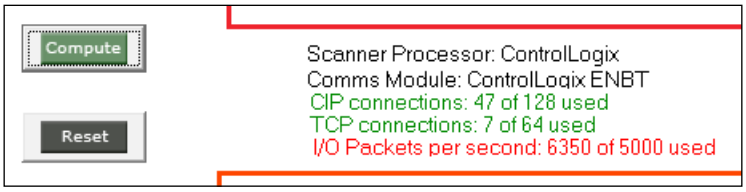


8. Our control system needs a window into the process so that the operator can control and monitor it. Next, we will add our **FactoryTalk HMI**.
9. In **Node Group 3**, select **FactoryTalk HMI** from the drop-down list to add an HMI to our network.

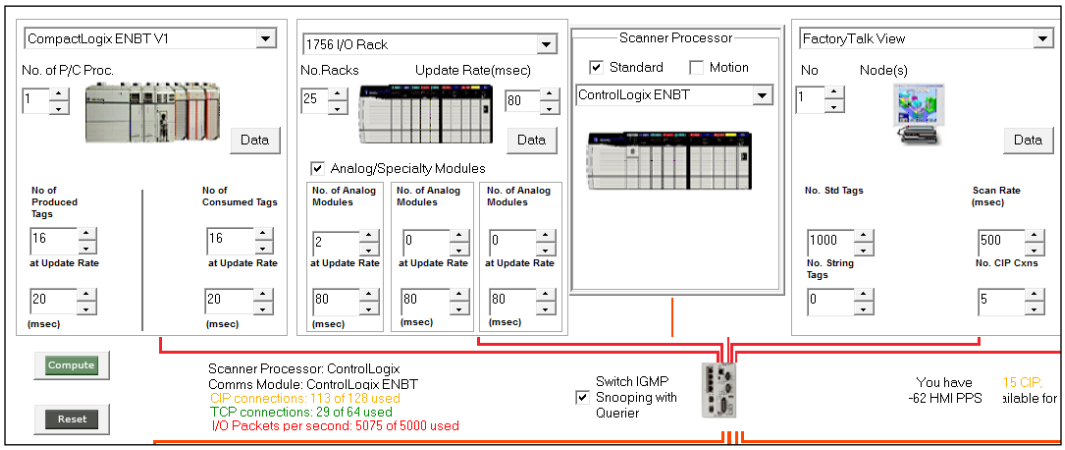
10. Set the **No. Std Tags** field to **1000** and the **No. CIP Cxns** field to **5**:



11. Next, we will hit the **Compute** button to see how many network resources we have left.
12. Now, we increase the size of our network to exceed the capacity of our Scanner Processor and view the results.
13. In our **Node Group 2** where we have set up our 1756 I/O rack, change the **Update Rate(msec)** field to **2** and then hit the **Compute** button.
14. You will notice that the network connections have turned red and our I/O PPS now exceed the limit for our Scanner Processor. As you can now see, adjusting the **Update Rate(msec)** field greatly impacts the PPS:



15. Finally, let's change the **Update Rate(msec)** field back to 80 msec but update the **No. Racks** field for **1756 I/O Rack** to **25**.
16. As you can see from the results, increasing the number of racks increased the CIP connections, TCP connections, and the I/O PPS values:



RSLinx

RSLinx is a communication server application in the Rockwell Automation product line. There are two types of RSLinx today:

- RSLinx Classic
- RSLinx Enterprise

RSLinx Enterprise is typically packaged with FactoryTalk HMI software. RSLinx Enterprise is an HMI communications gateway that also provides the following properties:

- Diagnostics
- Security
- Auditability
- Redundancy

For more information on RSLinx Enterprise, refer to the links in the appendix.

RSLinx Classic is typically packaged with RSLogix 5000 and Studio 5000 and is a communication gateway that also provides the following properties:

- Batch sequencing
- Firmware updates
- Uploading and downloading programs to controllers

In this chapter, we will be working with RSLinx Classic to create the communications gateway between our computer and our controller. Before we start to work with RSLinx Classic, we will define some key terms regarding controller communications:

- **Upload**: This term is used to copy a program from the controller to the computer.
- **Download**: This term is used to copy a program from the computer to the controller.
- **Equal**: This term is used when the computer and the controller both have the same program loaded. When a program is equal, you will be able to go online and view the program's current values and make changes online.
- **Online**: This term is used while working on a program that is currently residing on the controller.
- **Offline**: This term is used while working on a program that is currently residing on the computer.

There are many, many combinations of the communication methods and the Logix controllers that can be configured using RSLinx Classic, and most of the combinations follow a similar setup procedure. In the next exercise, we will establish communications between RSLinx Classic and a ControlLogix L7X controller using a USB connection.

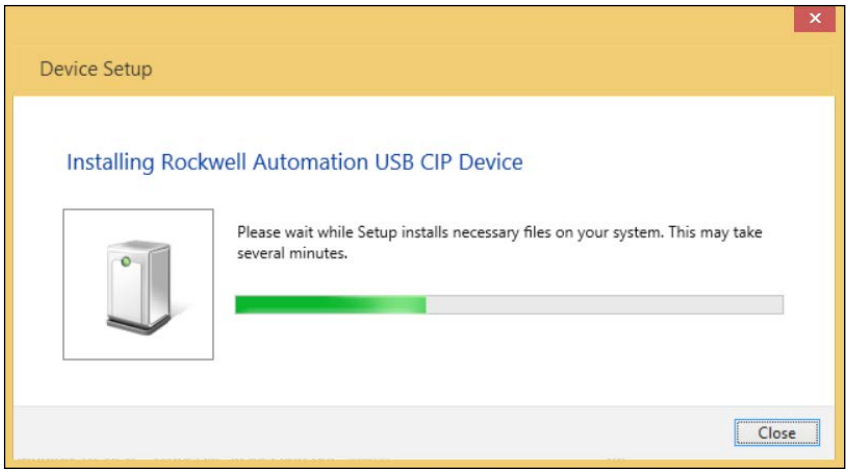
RSLinx communication using ControlLogix and a USB connection

In this exercise, we will be connecting to an L7x controller using RSLinx and the USB 2.0 connection on the front of the controller. Setting up communication to a device on the Logix platform requires three separate tools:

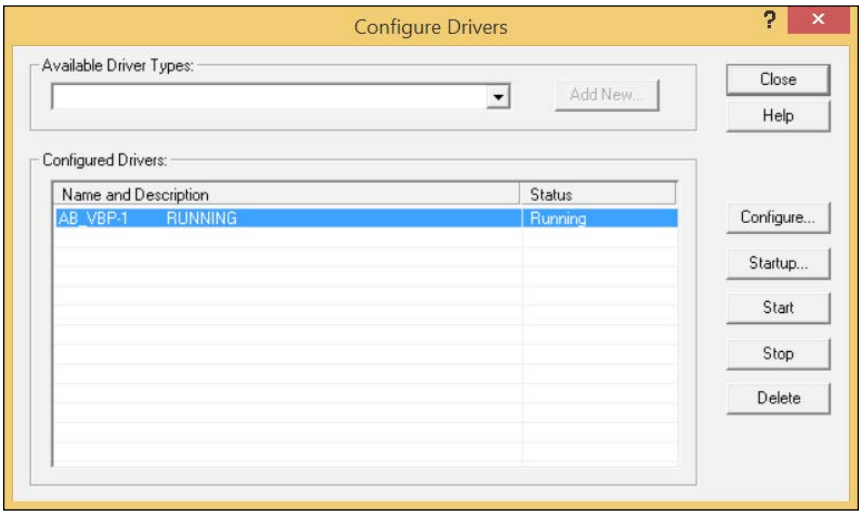
- **RSLinx**: This tool is used to configure the driver. The driver is a piece of software that allows you to communicate with hardware such as a controller.
- **RSWho**: This tool is a component of RSLinx and is used to specify the driver and the path to the device.
- **RSLogix 5000 / Logix Designer**: This tool communicates to the device using the path and the specified driver.

You will need to have a version of RSLogix 5000 / Studio 5000 installed that supports the L7x platform (Version 18.11 and higher). If you are using an older ControlLogix or a CompactLogix, review the following exercise and then navigate to **RSLinx Help | Quick Start** to find the detailed procedures for your controller:

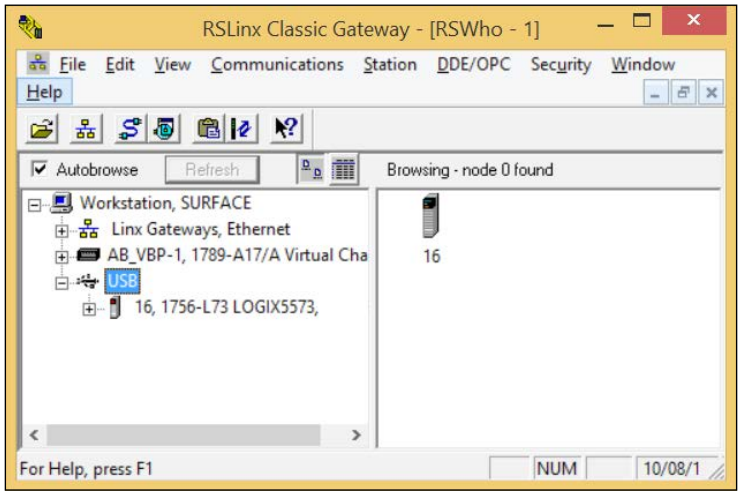
1. Before opening RSLinx, the first step is to connect your computer to your powered on ControlLogix controller using a standard USB 2.0 cable. The first time the Logix controller is connected to your computer, you will be prompted by a **Windows Found New Hardware** setup. On older versions of Windows, the first step of the wizard will ask whether you want to connect to Windows update to search for the driver. If the driver was installed automatically with RSLinx, select **No**, not at this time. Then, follow the wizard to complete the installation of the USB CIP Device. The device setup form will vary depending on your version of Windows. The following screenshot shows the **Device Setup** popup that appears in Windows 8.1:



2. Next, open RSLinx and configure the driver to connect to the L7X controller. Open the **Configure Drivers** window by navigating to **Communications | Configure Drivers**. If everything is installed correctly with your USB connection, you should see a driver named **AB_VBP-1** with the status column displaying **Running**:



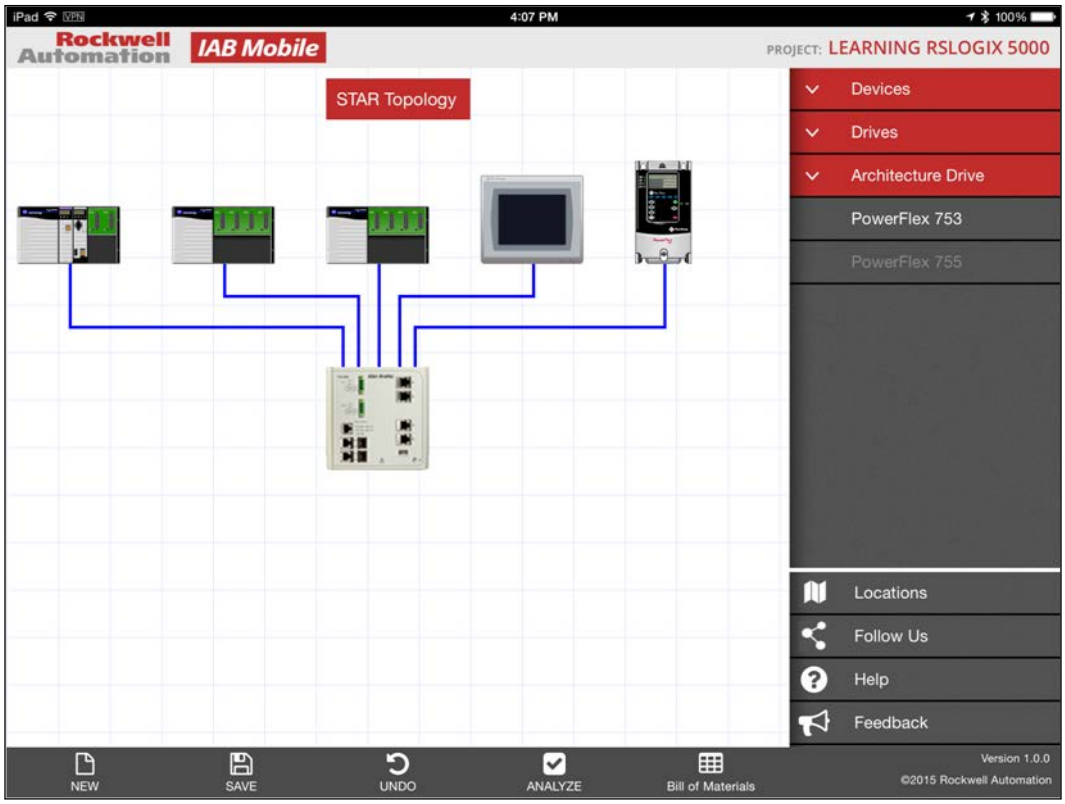
3. Next, let's open RSWho in RSLinx to verify that we can map a path to the device using the USB driver. Close the **Configure Drivers** window to return to RSLinx and open RSWho by selecting **Communications** | **RSWho**. You should be able to expand the **USB** tab on the left panel by clicking on the **+** symbol and selecting your controller:



4. Now that we have established the driver and communication's path to our controller, we will be able to select it using RSLogix 5000 or Logix Designer and start uploading or downloading programs. We can also use RSLinx Classic as an **OLE for Process Control** (**OPC**) communications gateway by configuring the OPC topics. More information on RSLinx can be found in the appendix or in Rockwell Automation Literature Library.

The Rockwell Automation Integrated Architecture Builder mobile app

The **Rockwell Automation Integrated Architecture Builder** (**IAB**) mobile app provides an iPad-based tool for creating industrial control network designs. The IAB app allows you to architect a simple Logix network using all of Rockwell Automation's major hardware platforms and EtherNet/IP. Using the IAB app, you can drag and drop your Logix solution together, analyze the EtherNet/IP performance data, generate a bill of materials, and send it to the nearest distributor/sales office. In addition, you can export your IAB app network to the IAB desktop application version for system validation. The IAB app is available for free in the Apple App Store and appears to be a replacement for the **Rockwell Automation Small System Sketcher**. The following screenshot is of the IAB app running on an iPad:



Summary

In this chapter, we learned about the communication technologies that are available in the Logix controller family. We investigated the strengths and limitations of each communication method and took a deep dive into EtherNet/IP and EtherNet/IP Capacity Tool. We explored RSLinx and its place within Integrated Architecture as the communications gateway. We also established communication to a ControlLogix L7x using the USB cable. Finally, we introduced the easy to use IAB mobile app for designing your Logix networks.

In the next chapter, we will introduce the modules that are available in the Logix platform and demonstrate their configuration with the RSLogix 5000 / Logix Designer.

3
Configuring Logix Modules

In this chapter, we will look at the available modules for the Logix platform, how to configure them, and their usage in a Logix project. We will also include methods for identifying module features by their Logix module **Catalog Numbers**, and introduce the address tree that a typical I/O module creates. After completing this chapter, you will be able to select and add I/O modules to your projects, modify the module configurations, and reference their real-time values using the recommended best practices. This chapter will cover the following aspects of the Logix modules:

- Module terminology
- Analog modules
- Digital modules
- Specialty modules
- Logix terminal blocks
- Configuring a ControlLogix module
- Logix module Catalog Numbers
- Module special features
- Addressing the module I/O data
- Buffering the module I/O data

Module terminology

Let's begin by taking a look at some of the common Logix module properties:

- **Voltage**: This attribute is the difference in electrical potential between two points, measured in voltage A/C (VAC) or D/C (VDC). When visualizing voltage, I prefer the age-old Super Soaker (water gun) analogy in which voltage is the pressure (the number of times you have pumped the Super Soaker).
- **Current**: This attribute is the flow of electrical charge measured in amps (A) and milliamps (mA). In the Super Soaker analogy, it is the diameter of the water gun's nozzle.
- **Signal**: This attribute is the modulation in voltage or current, which relays the operational state of a device. A signal is representative of values such as pressure, temperature, and flow.
- **Input**: This module is wired to detect the values being sent from the field to the controller. Input values are used to determine, for example, whether a motor is running and the speed of a motor's rotation.
- **Output**: This module is wired to detect values being sent from the controller out to the field. Output values are used to, for example, start a motor and tell a motor how fast it should run.
- **Rack**: This attribute is a chassis that contains the controller modules, and typically, range in the size from 4 to 17 module slots. ControlLogix also supports multiple racks that can be connected to each other using communications technologies such as EtherNet/IP or ControlNet. Most CompactLogix controllers mount along a DIN rail and do not use a rack.
- **Slot**: This attribute refers to a module's position in a rack. The number of slots will vary by rack size and the Logix controller you are using.
- **Module**: This attribute is a modular card that mounts in a slot of a rack or along a DIN rail. This module is used to handle a wide variety of automation tasks and is available from a number of vendors. In this book, we will focus on the most common modules, which simply process the input or output signals.
- **Channel**: This attribute is the individual input or output circuit on a module that links with one signal connection with the field.
- **Address**: This attribute is the complete path from a Logix controller to a module channel, property, or configuration value.
- **Adapter**: This attribute is the communication module that mounts in a slot on a rack, which enables a controller to communicate with a remote rack over EtherNet/IP or ControlNet. The adapter name is the root of the path to any address for modules located in remote racks.

Module types

In general, modules are classified as analog, digital, communication, controller, and specialty. They differ by the number of channels, the ranges of input and output they are capable of handling, and by the special features that are optionally available. In this section, we will explore the base module types and explain the possible feature sets.

Analog modules

The analog modules process the input and output of signals that vary by current and voltage and translate to real-world values such as pressure, temperature, and flow. The analog modules vary by the number channels (4 to 16), the operating temperature range, the maximum isolation voltage they can handle, the range of current (0 mA to 21 mA), and the range of voltage (+/- 25 VDC). There are even combination analog modules that house both input and output channels. Each analog channel usually requires three wires in order to complete an analog circuit correctly. The way in which the channel is wired changes depending on whether you are using voltage or current, so be sure to review the wiring diagrams for your module at Rockwell Automation Literature Library available online.

Digital modules

Digital modules process the input and output of signals that vary by current and voltage and translate to either `ON` or `OFF` values. Digital modules will vary by the number of channels (8 to 32), the operating temperature range, the maximum isolation voltage they can handle, and a range of voltage (0 VDC to 146 VDC, 10 to 265 VAC) supported.

Communication modules

There is a wide range of communication modules available that enable the remote rack communications, device communications, and motion control. There are communication modules by Rockwell and third-party vendors that allow the Logix controllers to communicate using the network technologies we talked about in the preceding chapter and many other non-Rockwell Automation protocols.

Controller modules

We covered the controller modules in *Chapter 1*, *ControlLogix and CompactLogix Overview and Firmware*, in detail. It is important to remember that ControlLogix can support multiple controller modules in a single rack (note that CompactLogix does not support multiple controllers on a single DIN rail).

Specialty modules

The specialty modules enable your automation project to perform the following specialized tasks:

- High-speed counting
- Flow meter measurement
- Limit switch monitoring
- Hydraulics control

Logix terminal blocks

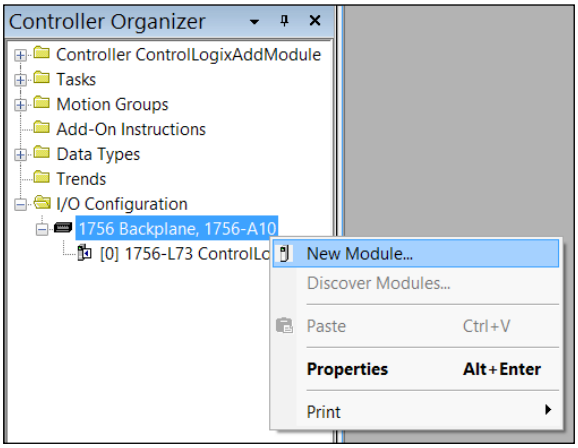
In the Logix family, most modules do not come with built-in screw terminals, and **Removable Terminal Blocks** (**RTBs**) or bulletin 1492, that is, **Interface Module** (**IFM**) must be purchased separately. You should carefully review your wiring requirements for your module using the online Rockwell Automation Literature Library resource (relevant links can be found in the appendix of this book).

Configuring a ControlLogix module

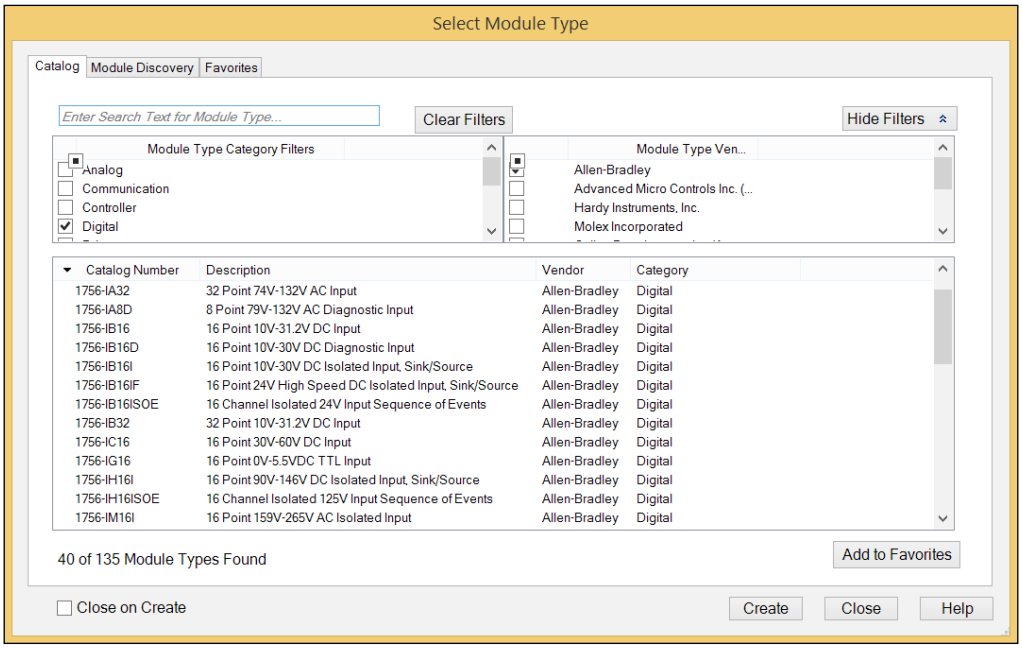
In this exercise, we will learn to add a ControlLogix module to a Logix project and look at a typical module configuration by performing these steps:

1. First, we will need to open RSLogix 5000 / Studio 5000 Logix Designer. Create a new project, and select a ControlLogix controller (in my case, I selected **1756-L73** on **Slot #0**). This process varies between versions of the Logix, so I will not show these steps in detail.

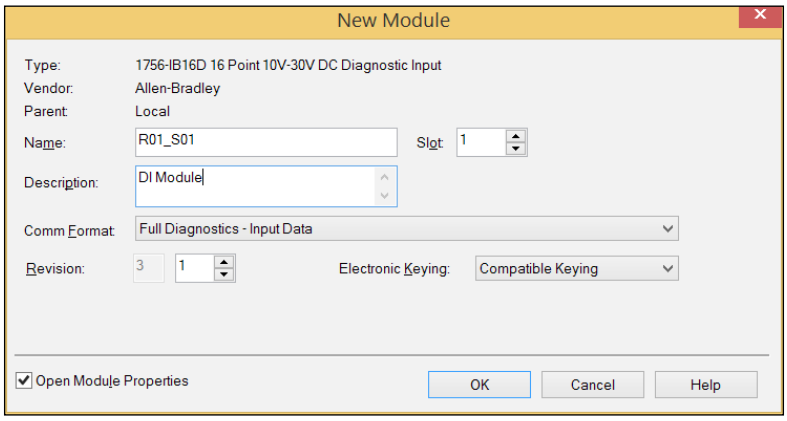
2. Next, we will add the module by right-clicking on the **Controller Organizer** pane's **IO Configuration** tree, and selecting **New Module…**, as shown in the following screenshot:



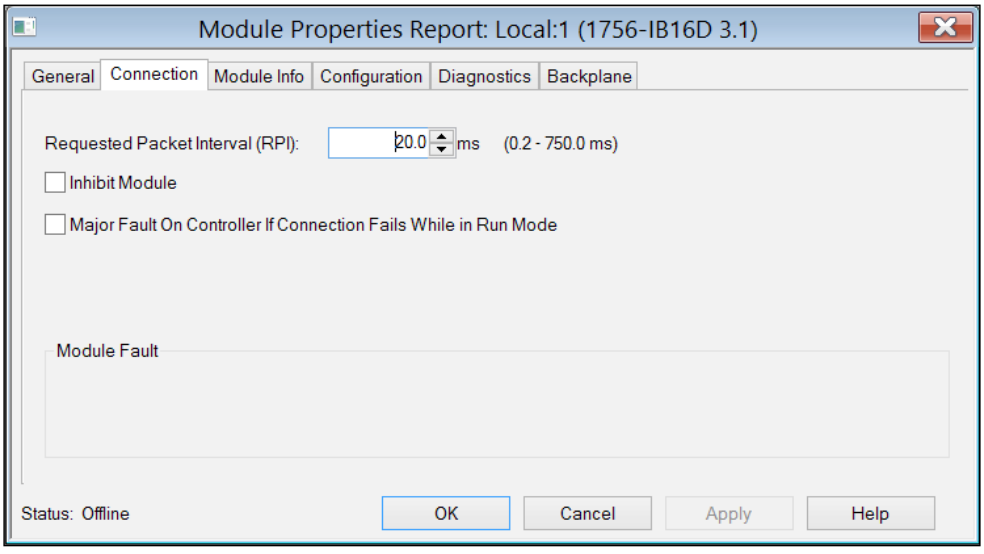
3. Now, we can select the module we wish to configure. For our example, it will be a digital input module, **1756-IB16D 16 Point 10V-30V DC Diagnostic Input**. The **Select Module Type** window varies from version to version of Logix, but regardless of the software, it is relatively easy to locate our module.



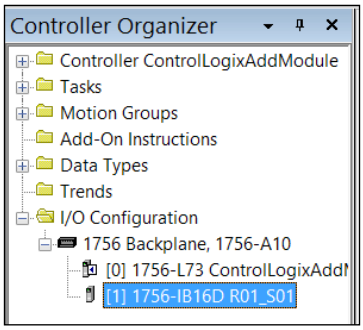
4. Next, we will configure the module by providing these inputs:
    - **Name**: `R01_S01`
    - **Slot**: `1`
    - **Description**: `DI Module`



5. Next, we can configure the module properties and adjust the setup of the module. Each module in Logix has unique properties and configuration requirements, so it is imperative that you refer to the Rockwell Automation Literature Library document for any module you are configuring. Once you have reviewed the module properties, click on the **OK** button, as shown in this screenshot:

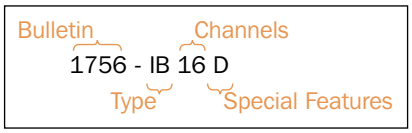


6. Finally, our ControlLogix digital input module has been configured, and you can see it in the **Controller Organizer** pane in our rack:



Logix module – Catalog Numbers

The modules in Integrated Architecture are referred to by their Rockwell Automation Catalog Numbers. Catalog Numbers are made up of four parts, as illustrated by the following diagram:



The preceding diagram breaks apart the Catalog Numbers for a ControlLogix digital input module with 16 channels and built-in diagnostics.

The bulletin number is a four-digit identifier for the Logix controller family. ControlLogix will begin with the **1756** bulletin number, the SoftLogix modules begin with the **1789** bulletin number, and CompactLogix will begin with the **1769** or **1768** bulletin numbers.

The module type is the second part of the Rockwell Automation module Catalog Numbers. Types that begin with `I` are input cards and types that begin with `O` are output cards. Let's take a look at a few sample types for commonly used modules. Digital input types are usually `IQ` or `IG` for VDC and `IA` or `IM` for VAC. Analog input types are typically `IF`, `IR`, or `IT`. Digital output types are generally `OB` for VDC or `OA` for VAC. Analog output types are typically `OF`.

Channels are the third part of Catalog Numbers. The channels represent the number of field signals that can be wired to and processed by the module.

The last field of Catalog Numbers indicates the special features of the module. The special features indicate any unique capabilities of the module. In the next section, we will take a look at a few special features from commonly used modules and their letter designations.

Special features of a module

Special features of a module provide additional support to the Logix modules and optional features as follows:

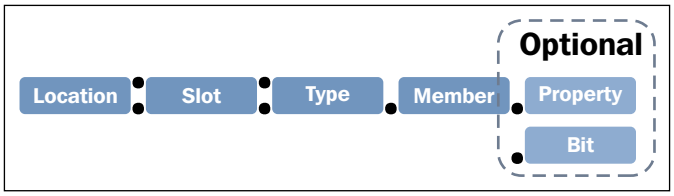
- **HART**: This feature allows the modules to read the transmitter status and health information or adjust the configuration and calibration of equipment through a Logix controller.
- **Diagnostic information**: This feature provides diagnostic information for each channel on the module. The following diagnostic information values are available in Logix:
    - **Field Power Loss detection**: When the field power to the module is lost, it can cause values to be misrepresented. The Field Power Loss detection will generate a point-level fault to the controller.
    - **Open Wire detection**: This feature is used to verify that the field wiring is connected correctly by measuring the minimum leakage current. A leakage resistor must be connected across the contacts of the device in order to provide the minimum leakage current.
    - **No Load detection**: This feature is a diagnostic feature of a module that detects a break in the field wiring by comparing it to a specified minimum load current (3 mA or 10 mA, depending on the module).
    - **Output State verification**: This module confirms with the controller that it received a command and whether the field-side device connected to the module has executed the command.
- **Electronic fusing**: This feature is the internal electronic fusing that prevents over-current through the module.
- **Individually isolated channels**: This feature is the per point isolation where each channel can be wired with its own individual power source.
- **Per point timestamping**: This module can be configured to record or latch the time at which a state is changed from `ON` to `OFF`, `OFF` to `ON`, or both.
- **FIFO mode operation**: This feature stores 160 timestamps, event sequence numbers, status, and input point numbers on the module for recording high-speed events (for example, shutdowns).
- **Ultra fast on/off times**: This feature is capable of switching within 15 uS.

The examples of the Logix module Catalog Numbers are as follows:

- `1756-IA32`: ControlLogix Digital 74-132 VAC Input 32 Pts (36P)
- `1756-IA8D`: ControlLogix Digital 79-132 VAC Diagnostic Input 8 Pts (20 Pin)
- `1756-IF16H`: ControlLogix Analog Input—16 Point HART
- `1769-IF8`: CompactLogix 8 Channel Analog Current/Voltage Input Module
- `1769-IQ16F`: CompactLogix 16 Point High-speed 24VDC Input Module

Addressing module I/O data

Individual channels on a module can be referenced in your Logix Designer / RSLogix 5000 program using its address. An address gives the controller directions to where it can find a particular piece of information about a channel on a module.



The first field of an address specifies the location of the channel (and is followed by a colon). The location can either be local to the controller or on a remote rack, which connects through a network adapter or bridge module. So this field can either be one of the following bridges:

- **LOCAL**: This module is on the same rack or DIN rail as the controller
- **Adapter name**: This module is the name you have configured for the network adapter or bridge module, which connects to the remote rack where the module is located

The second field of an address is the slot number of the I/O module in its rack or DIN rail (and is followed by a colon). The address slot numbering starts at zero. And in the case of CompactLogix (where power supplies can be placed in the middle of the DIN rail), power supplies do not count as a slot position.

The third field of an address is a single letter that represents the type of data. There are the following four types that are specified in an address:

- **I**: input
- **O**: output
- **C**: configuration
- **S**: status

The fourth field of an address specifies the member data of the I/O module. Different modules will store data of different types. For a digital module, a data member usually stores the input or output bit values. For an analog module, a channel member (CH#) usually stores the data for a channel.

The fifth field can be either a property or a bit of the member. A property provides specific data related to a member. A bit will provide a specific point on a digital I/O module. The bit range will depend on the size of the I/O module and like the slot position, it also starts at zero (0 to 31 for a 32-point module).

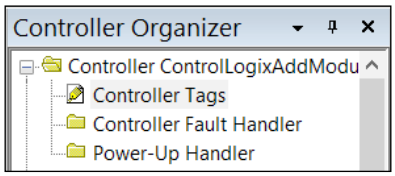
The examples of the Logix module addresses are as follows:

Logix module addresses	**Description**
`MyRack_3:11:O.Ch4Data`	Channel 4 of the analog output module on slot 11 of the adapter, **My Rack**.
`Local:3:I.Data.24`	Channel 24 of the digital input module on slot 3 in the local rack.
`Local:3:I.Fault.24`	Fault status for channel 24 of the digital input module on slot 3 in the local rack.
`Local:3:C.DiagCOSDisable`	Configuration Boolean value for disabling the COS diagnostic information for the digital input module on slot 3 in the local rack.

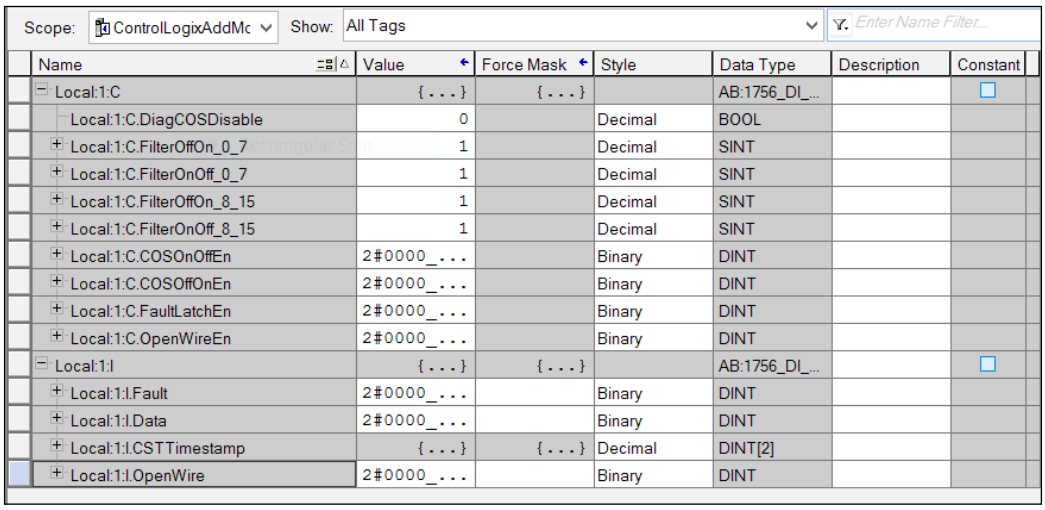
Exploring module addresses

In this exercise, we will explore the I/O module addresses for the digital module we added earlier in the chapter. Perform the following steps:

1. First, in the **Controller Organizer** pane, select and double-click on the **Controller Tags** option to open the **Controller Tags** panel, as shown in the following screenshot:



2. You will notice that because we added a diagnostic module, there are two address trees associated with the local slot 1. There is one address tree for the diagnostic configuration type data and another address for the input type data, `Local:1:C` and `Local:1:I`.



Name	Value	Force Mask	Style	Data Type	Description	Constant
Local:1:C	{...}	{...}		AB:1756_DI_...		
Local:1:C.DiagCOSDisable	0		Decimal	BOOL		
Local:1:C.FilterOffOn_0_7	1		Decimal	SINT		
Local:1:C.FilterOnOff_0_7	1		Decimal	SINT		
Local:1:C.FilterOffOn_8_15	1		Decimal	SINT		
Local:1:C.FilterOnOff_8_15	1		Decimal	SINT		
Local:1:C.COSOnOffEn	2#0000_...		Binary	DINT		
Local:1:C.COSOffOnEn	2#0000_...		Binary	DINT		
Local:1:C.FaultLatchEn	2#0000_...		Binary	DINT		
Local:1:C.OpenWireEn	2#0000_...		Binary	DINT		
Local:1:I	{...}	{...}		AB:1756_DI_...		
Local:1:I.Fault	2#0000_...		Binary	DINT		
Local:1:I.Data	2#0000_...		Binary	DINT		
Local:1:I.CSTTimestamp	{...}	{...}	Decimal	DINT[2]		
Local:1:I.OpenWire	2#0000_...		Binary	DINT		

3. After expanding the type address space, we can see the member data contained within the configuration and input types.

4. Under the configuration member addresses, we can view and adjust the configuration values for all the diagnostic features of our I/O module (recall `D` at the end of the Logix module Catalog Number of the I/O module we selected—`1756-IB16D`). We can see the configuration member data for COS, Open Wire, Fault Latching, and Filtering. You can easily modify the configuration of these features by double-clicking on the **Value** field and entering a new value.
5. Under the input member address tree, we see the real-time values from our input module. In addition to the normal digital input values, our input module is armed with diagnostic information such as channel faults, channel open wire detection, and channel change of state timestamping. These addresses can be referenced in our program code (ladder logic, function block, structured text, sequential flow diagrams) and evaluated directly in our logic.

It is recommended as the best practice to buffer the module I/O data before evaluating it in logic. In the next section, we will introduce the concept of module I/O data buffering.

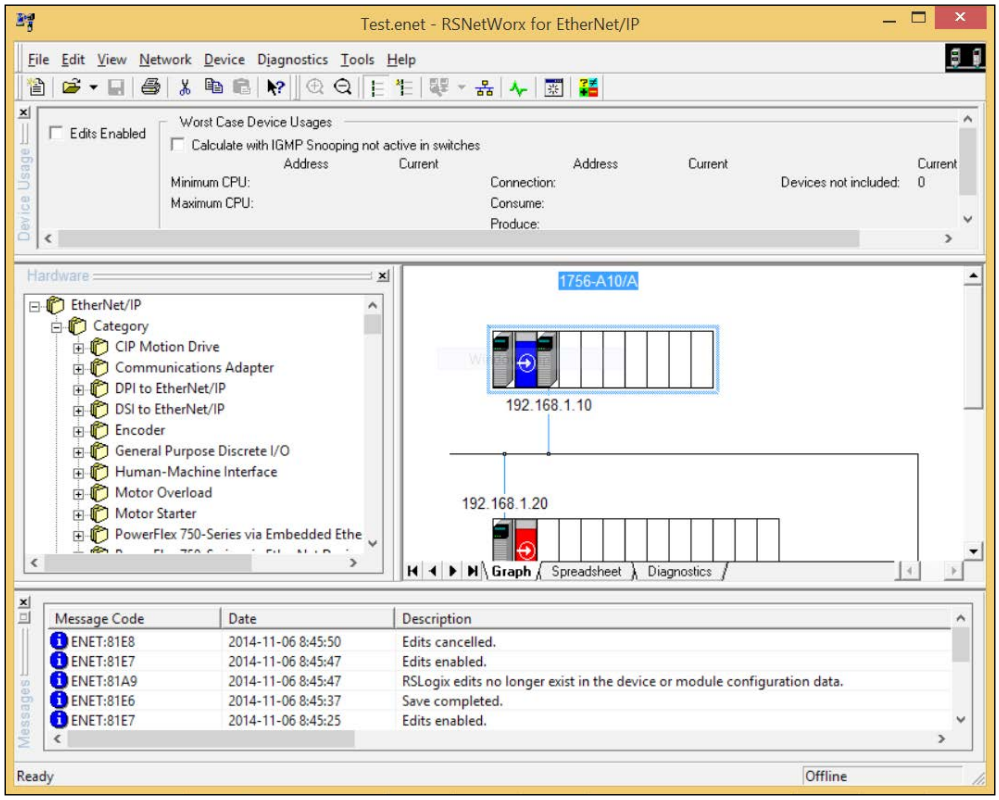
Buffering module I/O data

In *Chapter 1*, *ControlLogix and CompactLogix Overview and Firmware*, we briefly looked at the Logix operating cycle and the differences between asynchronous and synchronous execution. In the olden days of PLC5s and SLC500s, before we had access to high-performance asynchronous controllers such as those of the ControlLogix, SoftLogix, and CompactLogix families, program execution was synchronous and very predictable. In an asynchronous controller, there are many activities that appear to be happening at the same time. The input and output values can change in the middle of a program scan and put the program in an unpredictable state. Today, there is a rule in most automation companies that require programmers to write code that buffers the I/O data to base tags that will not change during a program execution. We will look at the buffering examples later in the book, and also explore a new alternative to buffering I/O available in Logix Designer Version 24 using the program parameters.

Configuring remote racks with RSNetWorx

RSNetWorx is a standalone application that is used to configure the Logix network topologies and export them to an EDS file, which we can import into our Logix application. Without an EDS file, we are unable to set up remote racks in an application. Remote racks are an advanced topic in the Logix platform, and we will not cover it in detail in this book. For more details on RSNetWorx, see our link to the Rockwell Automaton Literature Library document in the *Getting Results with RSNetWorx* section in the appendix of this book.

RSNetWorx is on the annual release of the Rockwell Automation toolkit. It can also be found on the Rockwell website as a free download for approved customers and system integrators. There are three versions of the product, one for each Logix network type (EtherNet/IP, ControlNet, and DeviceNet).



RSNetWorx for EtherNet/IP

Summary

In this chapter, we learned about the types of modules that are available in the Logix controller family. We introduced the basic module terminology that is commonly used in the industry and the procedure for adding modules to our project, and demonstrated the methods for addressing the module values.

In the next chapter, we will introduce the Rockwell Automation SoftLogix platform, which is a PC-based automation controller we can use for simulation, testing, or control.

4
SoftLogix

In this chapter, we will introduce the Rockwell Automation SoftLogix 5800 controller and virtual chassis. We will step through the setup of SoftLogix Chassis Monitor and the configuration of our SoftLogix controller within Logix. Finally, we will investigate the techniques for simulating I/O using the 1784-SIM module. This chapter will cover the following elements of the SoftLogix platform:

- SoftLogix controllers
- Components of a SoftLogix solution
- SoftLogix 5800 versus RSLogix Emulate 5000
- SoftLogix 5800 Chassis Monitor
- Configuring RSLinx virtual-backplane driver
- Creating a Logix Designer SoftLogix project
- Simulating I/O with 1784-SIM modules

SoftLogix system overview

The SoftLogix 5800 controllers (SoftLogix) enable you to create a PC-based Logix controller rack. Using the SoftLogix application, you can create a virtual rack that houses your virtual controllers and virtual communication modules. SoftLogix is another component of Rockwell Automation's Integrated Architecture and can interface with the other Logix controllers, communication modules, and I/O modules. By taking advantage of the computing power of modern PCs, the SoftLogix controllers are capable of processing larger volumes of data and at a higher speed than even the most powerful Logix controller.

Furthermore, the SoftLogix controllers support the 1784-SIM (I/O simulator) modules, which emulate the real-world I/O modules for debugging, operator training, acceptance testing, and simulation. Studio 5000 allows you to build the custom C++-based DLLs, called **external routines**, which can be executed by the SoftLogix external routines, allowing you to take control over all the aspects of the Windows-based PC and perform resource-intensive program execution. In the next section, we will review the three SoftLogix processors that are available today.

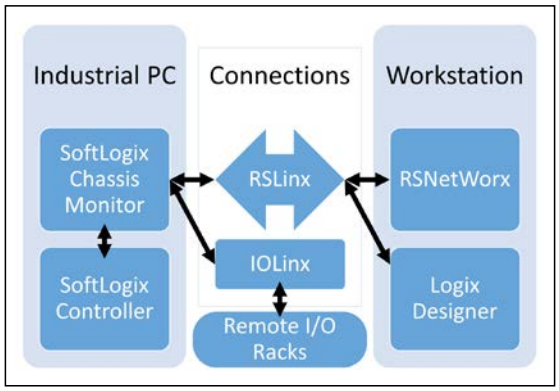
SoftLogix controllers

There are three bulletin 1789 SoftLogix controllers: 1789-L10, 1789-L30, and 1789-L60, and they are all fully integrated into the Logix platform. The SoftLogix controllers can be incorporated into any Logix control system solution. The SoftLogix controllers are capable of running the logic programs designed for the standalone Logix controllers on a Windows-based PC. When running a SoftLogix controller on a PC, up to 32 configurable tasks can be run simultaneously. Communication modules can be added to PCs using the PCI cards available from Rockwell Automation. The following table details the three SoftLogix controllers and their distinguishing features:

Bulletin 1789 SoftLogix controller					
Controller	**Memory**	**Rack size**	**1784-SIM**	**PCI cards**	**Third party support**
1789-L10	2 MB	3 slots	1	N/A	No
1789-L30	64 MB	5 slots	5	5	Yes
1789-L60	64 MB	16 slots	16	16	Yes

Components of a SoftLogix solution

In this section, we will explore the software and hardware components that interact with SoftLogix. The following diagram displays a typical SoftLogix solution architecture:



The assets in the preceding diagram are explained as follows:

- **SoftLogix Chassis Monitor**: This application is the virtual software-based rack that runs on the PC. It allows you to manage the virtual rack modules, monitor the status of the controllers and modules (just like a physical rack), and configure your virtual controllers.
- **SoftLogix controller**: This application is a virtual controller process running on a Windows-based PC. The virtual controller runs the RSLogix 5000 / Logix Designer programs such as ladder logic and **Function Block Diagrams** (**FBD**), which would usually run on a physical controller. It is also capable of running the C++-based external routines.
- **RSLinx**: As discussed earlier in the book, RSLinx is a communications gateway, which allows PCs to communicate with the Logix controllers.
- **IOLinx**: This application is an API that allows the SoftLogix controllers to read the I/O data from a physical I/O module.

SoftLogix 5800 versus RSLogix Emulate 5000

Although we do not cover Emulate 5000 in detail in this book, it is important to understand the differences between SoftLogix 5800 and Emulate 5000. These products provide a PC-based virtual rack and controller.

Note that you cannot install SoftLogix on a PC that already has RSLogix Emulate 5000 installed on it. In order to install SoftLogix, you must first uninstall RSLogix Emulate 5000.

RSLogix Emulate 5000 is a virtual Logix controller and rack, designed to allow you to debug your Logix program code using features such as breakpoints and tracepoints. Using Emulate 5000, you can create a virtual test rack using similar modules to a physical rack and even test your project with an HMI. Unlike SoftLogix, Emulate 5000 is not capable of controlling real I/O. Also, the communication modules are not supported by Emulate 5000. A typical RSLogix Emulate 5000 solution consists of modules being configured in a virtual Logix rack to mimic the end solution. The logic is downloaded and monitored using Logix Designer / RSLogix 5000 in order to troubleshoot it. The following table outlines the most significant differences between these two products:

SoftLogix 5800	**RSLogix Emulate 5000**
Full-featured PC-based Logix controller	Emulated controller for Logix debugging
Full network module support	No networking modules supported
Normal Logix controller features	Advanced debugging features

For simulation and testing, one can easily use either SoftLogix or Emulate 5000. However, if you are looking for advanced debugging/logging features, Emulate 5000 would be a better option. The user interface and configuration of SoftLogix and Emulate 5000 are so similar that you can complete the SoftLogix exercises in this chapter using Emulate 5000.

Working with SoftLogix

In the following exercises, we will perform the following tasks:

- Setting up our SoftLogix 5800 virtual rack and controller
- Configuring our RSLinx driver and establishing communications to our SoftLogix controller

- Building a new Logix Designer / RSLogix 5000 project that connects to the SoftLogix controller
- Simulating I/O with the 1784-SIM modules

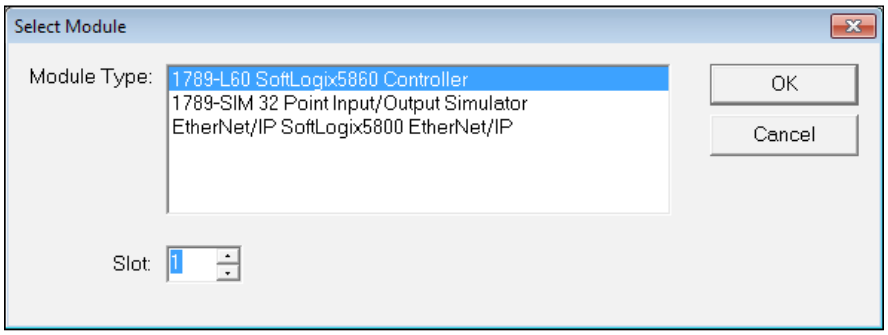
SoftLogix 5800 Chassis Monitor

In this exercise, we will configure a SoftLogix controller in the SoftLogix Chassis Monitor.

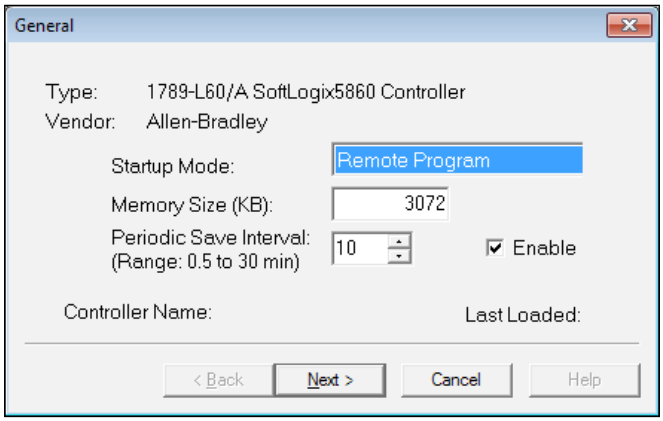
SoftLogix 5800 is included in the annual release of the Rockwell Automation toolkit. It can also be found on the Rockwell website for approved customers and system integrators. The SoftLogix controllers need to be licensed in order to run, and Rockwell Activation Manager will also need to be installed.

To configure a SoftLogix controller, perform the following steps:

1. Start up the SoftLogix 5800 Chassis Monitor software.
2. Open the drop-down menu option by navigating to **Slot** | **Create Module…**.
3. The **Select Module** dialog box appears and allows us to add the communication modules, 1789-SIM I/O simulation modules, and the SoftLogix 5800 controller modules to the slot we specify. Select **1789-L60 SoftLogix5860 Controller**, and **1** for the **Slot** number field using the numeric selector. Then, click on the **OK** button, as shown in the following screenshot:



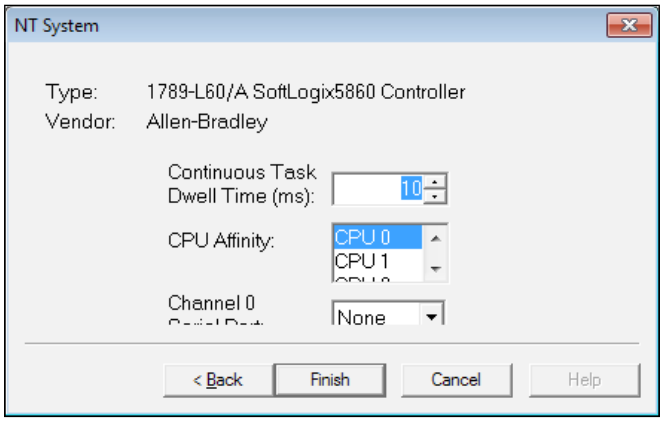
4. The **General** dialog box appears. Next, we will configure the SoftLogix controller we added to slot 1 in our virtual chassis. The **General** dialog box allows us to specify the following options:
    - **Startup Mode**: This option allows you to select the SoftLogix run mode state when the controller first starts up (equivalent to the key position on a physical controller). The options are **Remote Program** or **Last Controller State**.
    - **Memory Size (KB)**: This option is the amount of memory (RAM) the controller is allowed to use on the PC.
    - **Periodic Save Interval**: This option stores all the current tag values in the SoftLogix controller to the PC's hard disk drive. This value runs at a higher processor priority than other tasks on your PC and can impact the performance of other processes. Having a modern multicore processor helps to reduce this risk.
5. We are going to keep the default options in the **General** dialog box, so click on the **Next >** button, as shown in the following screenshot:



6. Now, the **NT System** dialog box appears and provides us with the following options:
    - **Continuous Task Dwell Time (ms)**: This option is the breathing room you offer to the PC to handle the other tasks running on the system. It is essential to provide the CPU with a slice of time to handle Windows-related system tasks; otherwise, you may find your CPU pegged at 100 percent utilization and your PC system unresponsive.

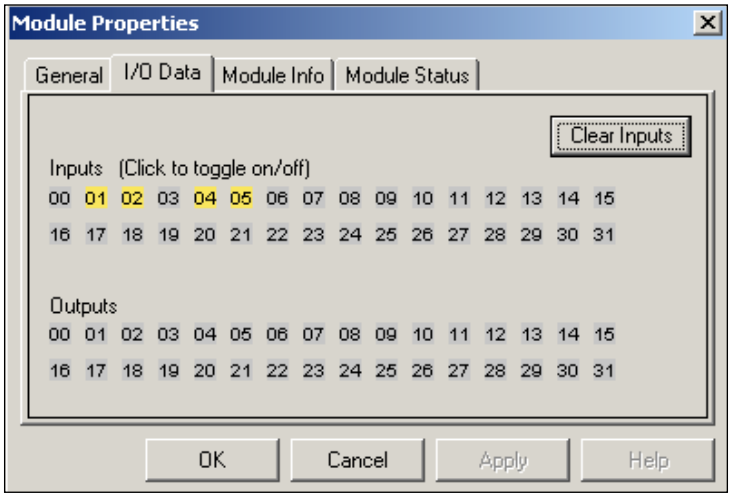
- **CPU Affinity**: This option is the CPU core used for the SoftLogix controller.
- **Channel 0 Serial Port**: This option is the COM port used for serial communication by the SoftLogix controller.

7. We are going to keep the default options in the **General** dialog box, so click on the **Finish** button, as shown in the following screenshot:

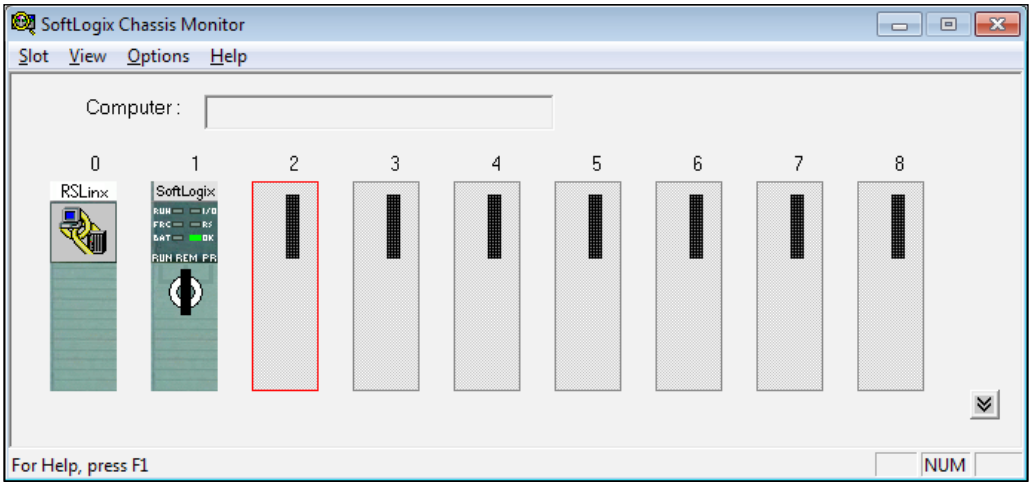


8. Adding a 1789-SIM I/O module will allow us to simulate input values and monitor output values. We can add the 1789-SIM module using the same process we used to add the SoftLogix controller module. Open the drop-down menu option by navigating to **Slot** | **Create Module…**.
9. In the **Select Module** dialog box, select **1789-SIM 32 Point Input/Output Simulator** in the **Module Type** field, select **2** in the **Slot** field, set the **Label** option to **Simulated Points**, and click on the **OK** button.

10. Now that we have a 1789-SIM module added to our virtual backplane, we can toggle inputs and monitor the outputs. You can toggle the digital points by right-clicking on the 1789-SIM module and selecting **Properties**.



11. You can toggle the digital input values by selecting the **I/O Data** tab and toggling the input boxes. You can also see the output values by clicking on the module on the virtual chassis to open the module door.

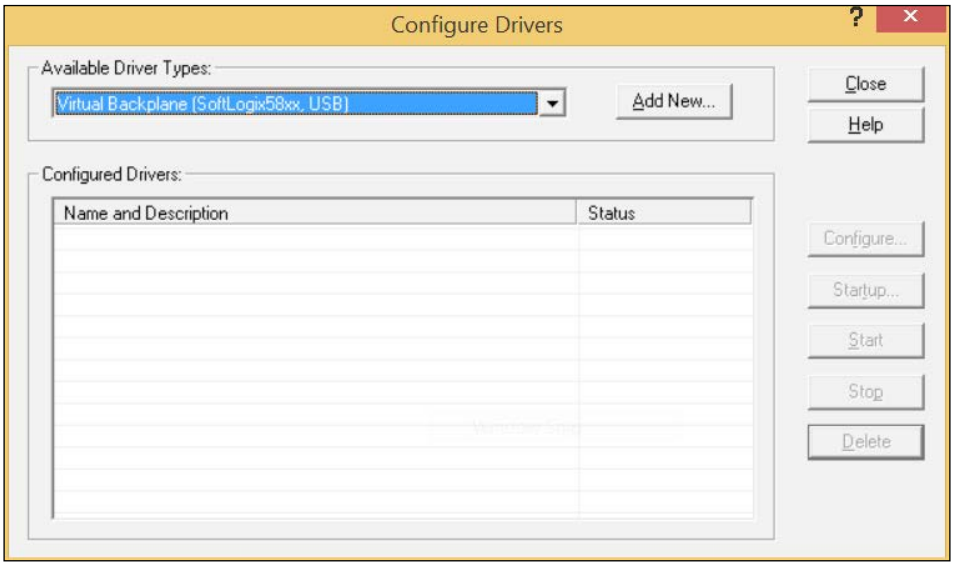


We have now completed the configuration of our SoftLogix controller, Chassis Monitor. Next, we will set up our RSLinx driver to allow Logix Designer / RSLogix 5000 to communicate with the SoftLogix controller.

Configuring the RSLinx virtual-backplane driver

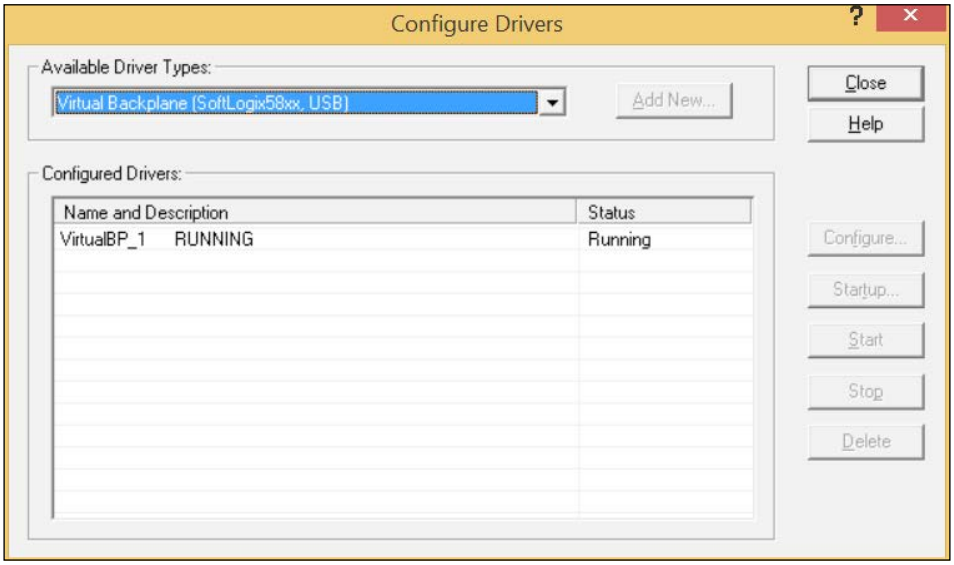
The RSLinx virtual-backplane driver is the communication gateway used by Logix Designer / RSLogix 5000 to program and monitor the SoftLogix controllers. Follow these steps:

1. Run RSLinx Classic.
2. Navigate to **Communications | Configure Drivers**.
3. In the **Configure Drivers** window, click on the **Available Driver Types** drop-down menu and select **Virtual Backplane (SoftLogix58xx, USB)**. Then, click on the **Add New…** button, as shown in the following screenshot:



4. A Windows dialog box will appear with the title **Add New RSLinx Driver**. Enter the title, `VirtualBackplane_1`.

5. Next, the **Configure Virtual Backplane** dialog box will allow us to select the slot number where the RSLinx module will reside. By default, the module will be positioned in slot 0. Only in Logix Designer Version 2.1 and higher, we are allowed to select a slot position other than 0. Our virtual backplane drive is now configured and running, and we can close the dialog box by clicking on the **Close** button.



Now that we have our virtual backplane configured, we can start to develop programs and download them to our SoftLogix controller.

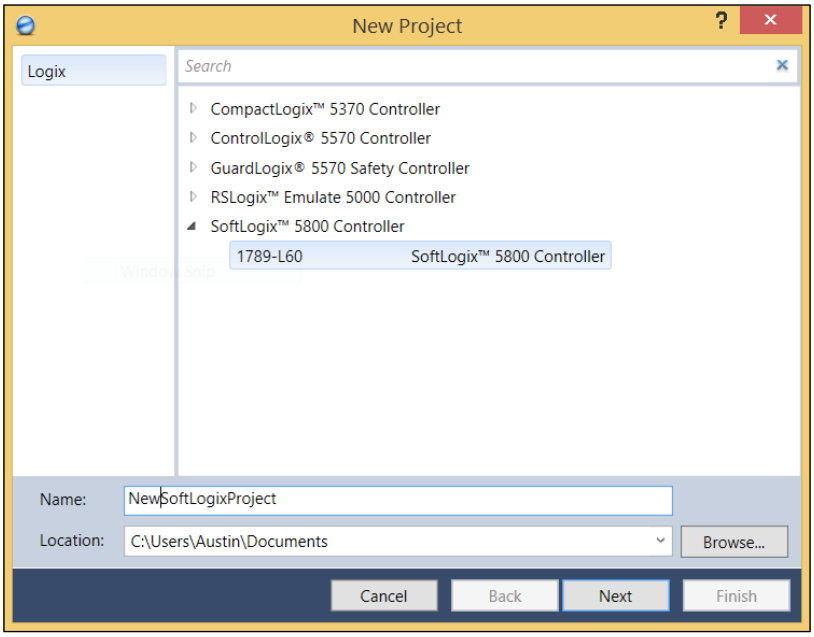
Creating a Logix Designer SoftLogix project

In order to download programs and run your SoftLogix controller, you will need to create and configure a new Logix project by performing the following steps:

1. First, open Logix Designer / RSLogix 5000 and create a new project by selecting **New Project** from the **Studio 5000** splash screen in Studio 5000 or opening the drop-down menu option on navigating to **File | New...** in RSLogix5000.

In Studio 5000 Logix Designer, there is another **New Project** wizard step, but in RSLogix 5000, there is only a single dialog box for creating a new project.

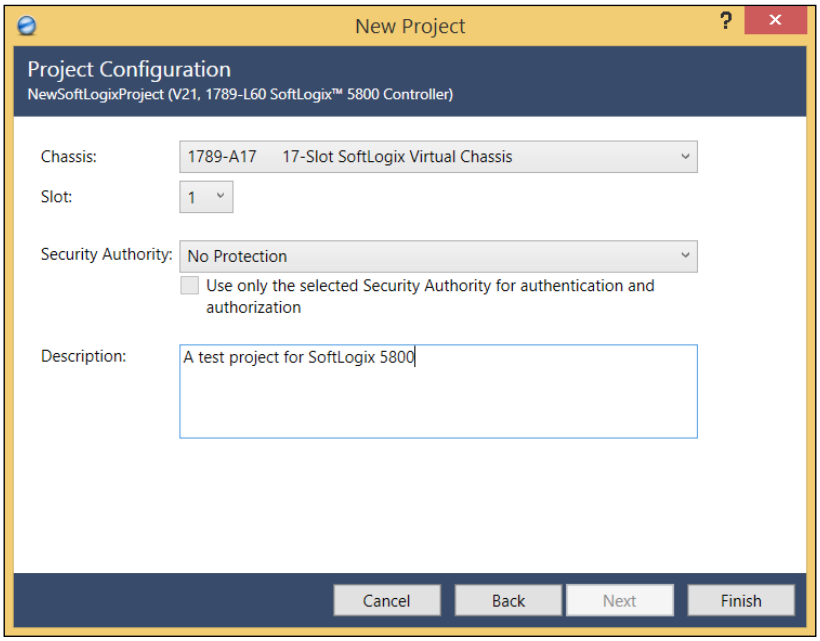
2. In the **New Project** dialog box, navigate to **SoftLogix™ 5800 Controller | 1789-L60 SoftLogix™ 5800 Controller** and set the **Name** field to `NewSoftLogixProject`. In Logix Designer, click on the **Next** button.



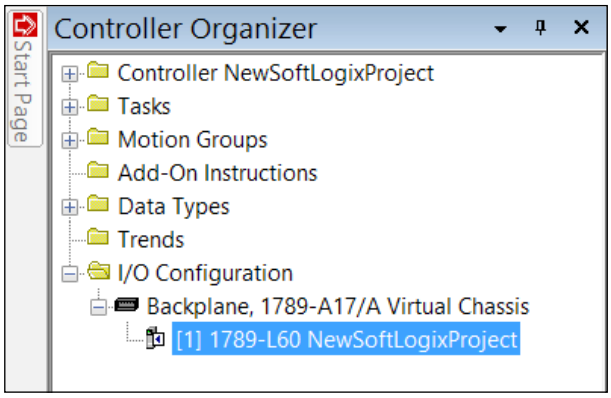
3. Set the **Chassis** field to **1789-A17 17-Slot SoftLogix Virtual Chassis** and the **Slot** field to **1**. Set the **Security Authority** field to **No Protection**.

In later versions of Logix, you can specify a value in the **Security Authority** field, which must be present on a FactoryTalk Network Directory in order to go online with a controller. Projects that are secured with a specific security authority cannot be recovered if that security authority is lost.

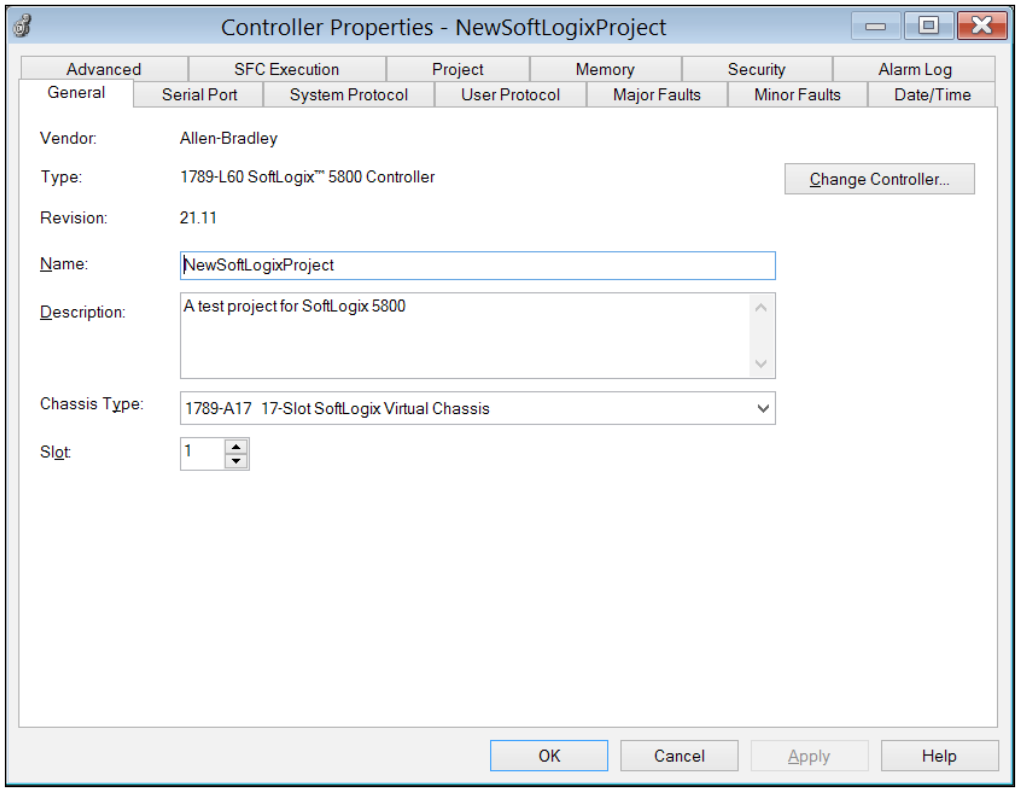
4. Specify a useful project description. Then, in Logix Designer, click on the **Finish** button or in RSLogix 5000, click on **OK**.



5. In the **Control Organizer** pane, we can see our SoftLogix controller listed.



6. Next, right-click on our newly added SoftLogix controller and select the **Properties** menu option.



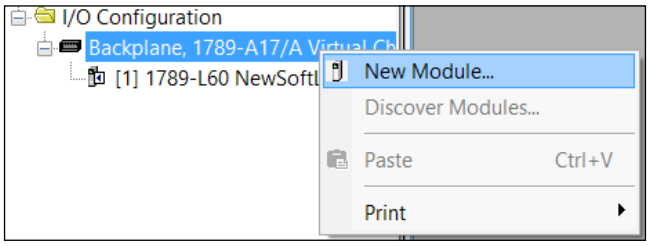
7. The **Controller Properties** window will allow us to configure the controller to suit the needs of our application.

We have now configured our SoftLogix controller and can add modules and begin downloading programs to it.

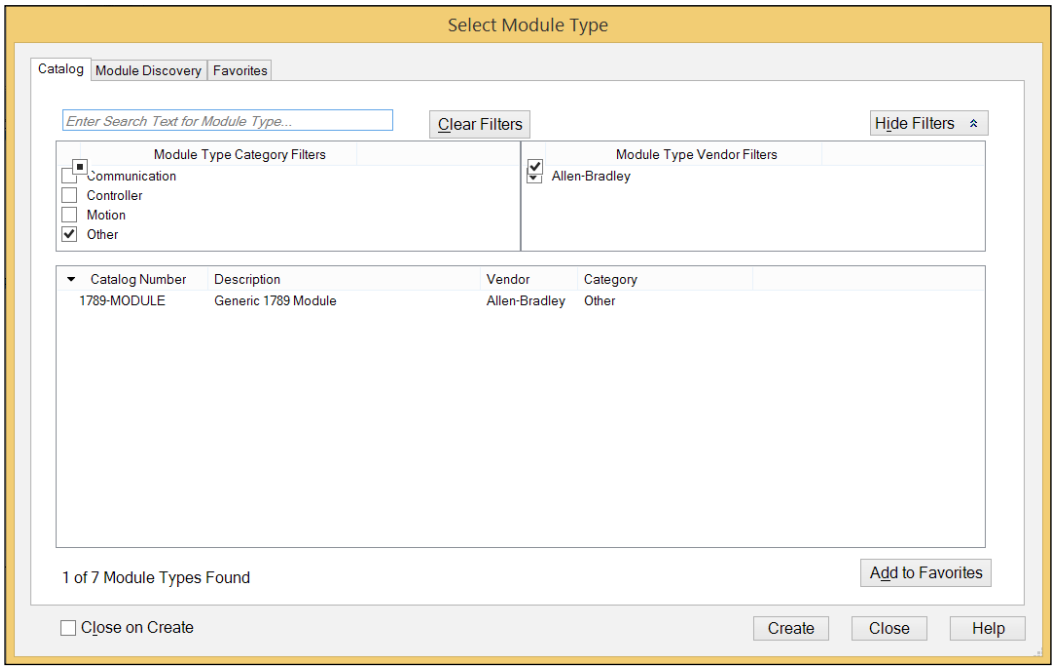
Configuring the 1789-SIM module in the Logix Designer project

In order to use our 1789-SIM module within our program, we will need to load and configure the module in our Logix Designer project:

1. Open Logix Designer / RSLogix 5000 and open the project we previously added our SoftLogix controller to.
2. In Logix, right-click on the **I/O Configuration** folder and select **New Module**.



3. The **Select Module Type** dialog box will appear. Use the dialog box to find the **1789-MODULE Generic 1789 Module** module (under **Other** in Logix Designer), and then click on the **Create** button.

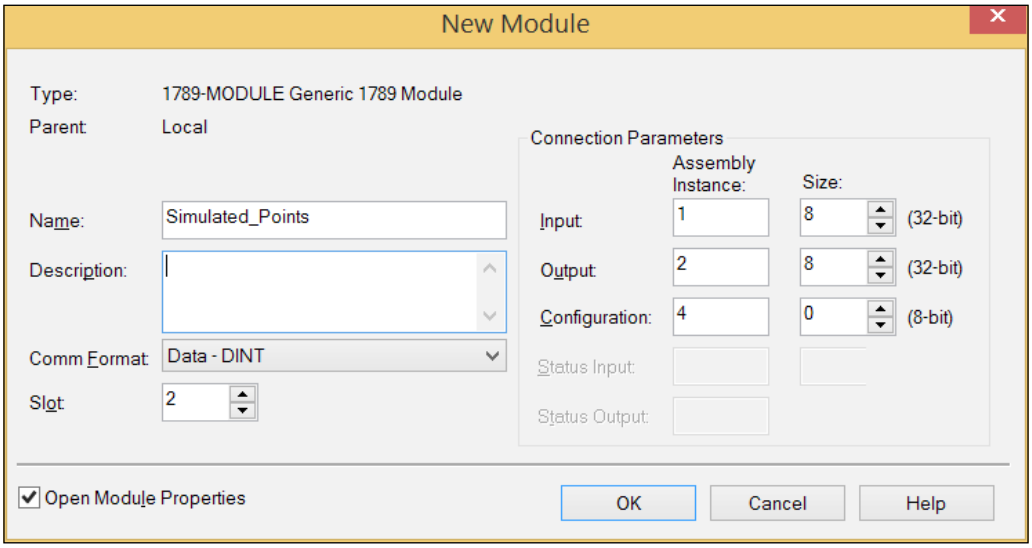


4. The **New Module** dialog box will appear and allow us to configure the 1789-SIM module's general properties.
5. The **Assembly Instance** field values represent the configuration modes for the 1789-SIM module. The **Status Input** and **Status Output** values are disabled when the **Comm Format** field that does not support the status information is selected. I have only ever seen two different configuration modes here, which are detailed in the following table:

Assembly instance	Read/write	Listen only
Input	1	1
Output	2	3
Configuration	4	4
Status input	5	5
Status output	6	6

As you can see from the table, the **Assembly Instance** values will always stay the same, except in the case of a **Listen Only** mode, where the **Output** field value changes to **3**.

The **Size** property for each **Assembly Instance** field value is used to specify the number of channels for **Input**, **Output**, **Status Input**, and **Status Output**. The **Configuration** field size is always 400 regardless of the value entered in the **Size** property. Set the properties of the 1789-SIM module, as shown in the following screenshot:



6. The preceding configuration will create the following input, output, and configuration controller tags:

Name	Value	Force Mask	Style	Data Type
Local:2:I	{...}	{...}		AB:1789_MODULE_DIN...
Local:2:I.Data	{...}	{...}	Decimal	DINT[8]
Local:2:O	{...}	{...}		AB:1789_MODULE_DIN...
Local:2:O.Data	{...}	{...}	Decimal	DINT[8]
Local:2:C	{...}	{...}		AB:1789_MODULE:C:0
Local:2:C.Data	{...}	{...}	Hex	SINT[400]

7. Had we selected a **Comm Format** field with **Status**, we would also have **Input Status** and **Output Status** controller tags appear in the **Controller Tag** list.
8. Click on the **OK** button.
9. The **Module Properties** window will appear and allow us to modify the module connection configuration.
10. Set the **Requested Packet Interval (RPI)** value for the 1789-SIM module to **50.0 ms** and click on the **OK** button.

Setting an RPI value to less than 50 ms can cause the 1789-SIM module to fail.

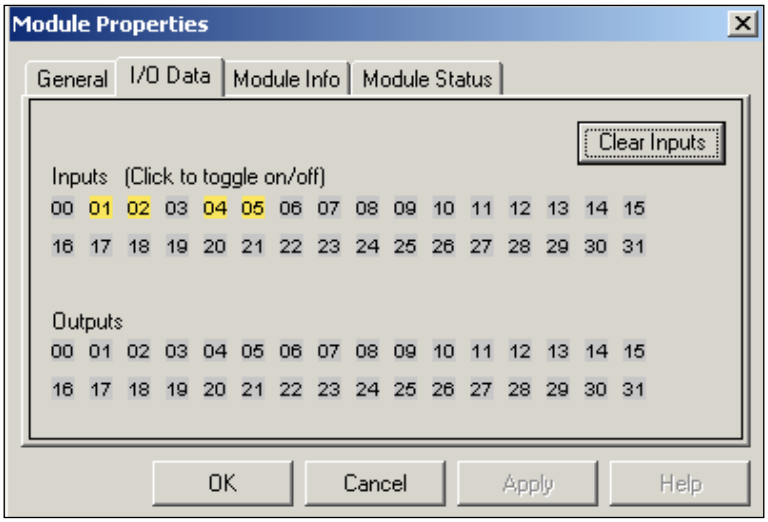
11. Now, we can connect and download our program to the SoftLogix controller just like a regular physical controller. Open **Who Active** by navigating to **Communications** | **Who Active**, and then navigate to our newly created SoftLogix controller and click on the **Download** button.
12. Finally, ensure that you are online with your SoftLogix controller, and in the next exercise, we can start to simulate some values.

Simulating values using the 1789-SIM module

Now that we have configured our virtual chassis, RSLinx, our controller, the 1789-SIM module, and downloaded our program, we can start to simulate input and monitor output values by performing these steps:

1. Open SoftLogix Chassis Monitor, right-click on the 1789-SIM module, and select **Properties**.
2. Click on the **Module Properties** tab, which is labeled as **I/O Data**.

3. The **I/O Data** tab will allow us to toggle digital inputs by clicking on them and monitor digital outputs.



When you toggle a digital input and return to your online program in RSLogix / Logix Designer, you will see the corresponding value has changed in the **Controller Tags** monitor.

Summary

In this chapter, we introduced the virtual controller options in the Rockwell Integrated Architecture platform, including SoftLogix 5800 and Emulate 5000. We learned how to configure our SoftLogix controller's virtual chassis and RSLinx application, and how to create a new project based on the SoftLogix platform. Finally, we learned the power of the 1789-SIM I/O cards and their use with the SoftLogix controller.

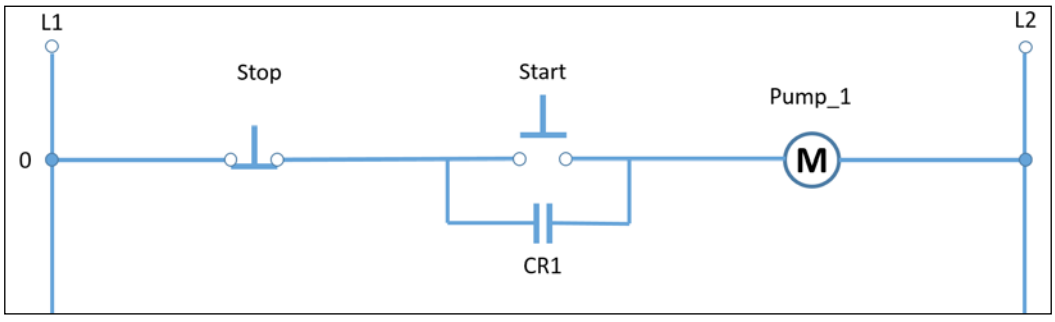
In the next chapter, we will introduce the ladder logic programming and its use with the Logix family of physical and virtual controllers.

5
Writing Ladder Logic

In this chapter, we will look at the history of ladder logic and the development of the IEC standard programming languages. Then, we will jump into the ladder logic programming by creating a simple pump control program. We will demonstrate how to buffer inputs and outputs in our ladder logic code and discuss the importance of this process. Finally, we will explore the new program parameter features in Logix Designer Version 24 and how that can be used to buffer values and greatly reduce the amount of ladder logic required for a program.

Ladder logic overview

Ladder logic was originally a written method for capturing the wiring of relay circuits also known as relay logic. The name ladder logic can be attributed to the diagrams resembling a ladder. Two vertical lines (often known as L1 and L2) represent the voltage of the circuit, and the horizontal lines and symbols represent the devices (buttons, motors, and breakers) connected to the circuit. Each horizontal line in the circuit is known as a rung. Once microprocessors enabled programmable logic in control systems, ladder logic evolved into a programming language rather than an engineering diagram. The following diagram is an example of a single relay logic rung:



Today, ladder logic is still the most popular industrial automation programming language.

IEC 61131-3

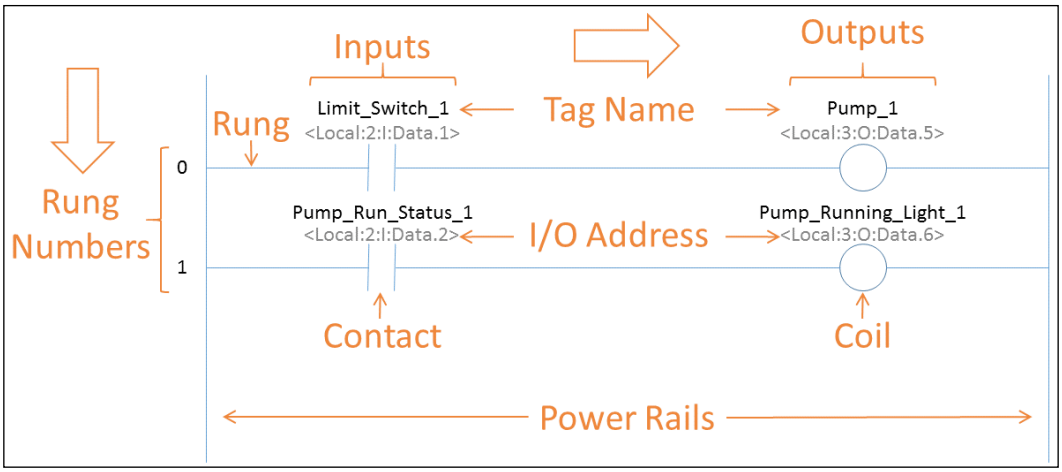
Ladder logic is one the five IEC 61131-3 Compliant languages available in Logix. IEC 61131-3 defines a common set real-time automation programming language structure, which is shared across multiple vendor's software products:

- Naming conventions
- Data types
- Task structure, scheduling, and execution control
- Execution flow control
- Program execution
- Triggers
- Scheduling

IEC, which was first published in December 1993, enables you to transit between programming platforms designed by different vendors. Furthermore, it improves the safety and reliability of automation applications by making them easily understood by a wider audience.

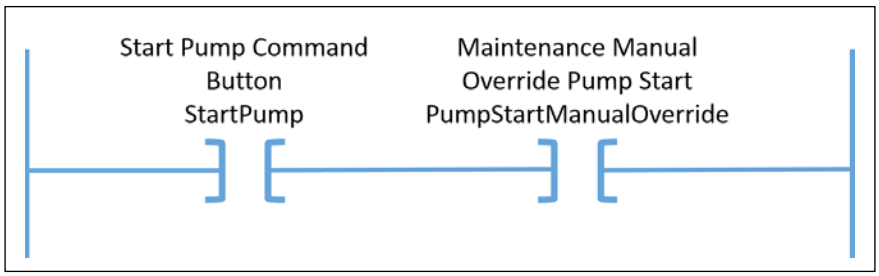
Understanding programming logic

Each rung of ladder logic is an equation solved by the PAC as **True** (**1**) or **False** (**0**), also known as energized (1) or de-energized (0). Ladders are executed one rung at a time from top to bottom, and each rung executes one instruction (also known as an element in Logix) at a time from left to right. The following diagram details the anatomy and terminology of a simple ladder logic program:



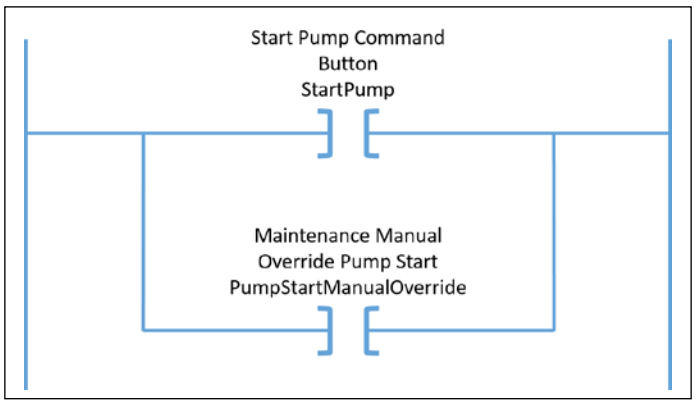
AND logic in ladder

When the ladder logic instructions are positioned side by side, known as the **AND** logic, both instructions need to evaluate as true, in order for the output to energize. The following diagram illustrates a simple example of the AND logic:



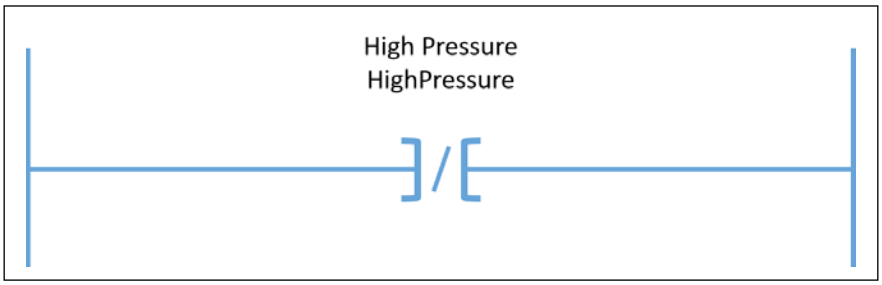
OR logic in ladder

When the ladder logic instructions are stacked on top of each other, known as the **OR** logic, either of the instructions can evaluate as true in order to energize the output. The following diagram demonstrates the OR logic in a ladder logic rung:



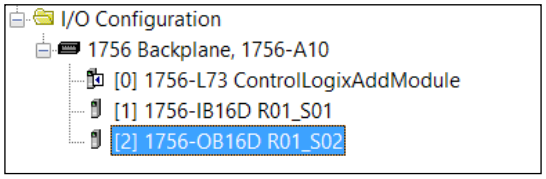
NOT logic in ladder

The ladder logic contact instructions, by default, are considered to be normally open and when closed, they evaluate as true. There is another form of ladder logic contact instruction that is normally closed, and can be identified by the diagonal line that passes through it. The normally closed contacts evaluate as true when their value is actually false and are considered to be the **NOT** logic in ladder logic. The following diagram shows an example of the NOT logic:



How to write ladder logic

In this section, we will create a ladder logic program using Studio 5000's Logix Designer (previously known as RSLogix 5000). Typically, a control system will read inputs from sensors, and equipment use the inputs in logic routines and then write to outputs. This activates the equipment in order to manipulate a process. To provide an example that aligns with a typical control system, we will need to ensure that our project contains both a digital input module and a digital output module. In order to complete the following exercise, you will need to load the project created in *Chapter 3*, *Configuring Logix Modules*, or simply create a new project and add a controller and a 16 channel digital input module to it (1756-IB16D 16 points 10V-30V DC diagnostic input). Then, add a digital output module (1756-OB16D 16 points 19.2V-30V DC diagnostic output module) by following the process we detailed in *Chapter 3*, *Configuring Logix Modules*.



Buffering I/O data

One important issue we must address prior to starting the development of our program is the buffering of module I/O data. In *Chapter 1*, *ControlLogix and CompactLogix Overview and Firmware*, we briefly looked at the Logix operating cycle and the differences between asynchronous and synchronous execution. In modern asynchronous operating cycles, there are many activities happening at the same time. The input and output values can change in the middle of a program scan and put the program in an unpredictable state. Imagine a program starting a pump in one line of code, and then closing a valve directly in front of that pump in the next line of code because it detected a change in process conditions. In order to address this issue, we use a technique called buffering, and depending on the version of Logix you are developing on, there are a few different methods of achieving this. Buffering is a technique where the program code does not directly access the real input or output tags on the modules during the execution of a program. Instead, the input and output module tags are copied at the beginning of a program's scan to a set of base tags that will not change the state during the program's execution. Think of buffering as taking a snapshot of the process conditions and making decisions on those static values rather than the live values that are fluctuating every millisecond. The two widely accepted methods of buffering are as follows:

- Buffering to base tags
- Program parameter buffering (only available in Logix Version 24 and higher)

Do not underestimate the importance of buffering a program's I/O. I worked on an expansion project for a process control system where the original programmers had failed to implement buffering. Once a month, the process would land in a strange state that the program could not recover from. The operators had attributed these problem to "Gremlins" for years until I identified and corrected the issue.

Defining tags

In Logix, a tag allows you to allocate and reference data stored in the controller. When working with legacy PLCs, a programmer would often use registers to store data and reference them using their addresses in memory. Modern PACs, such as the Logix family, use name-based tags to store and manipulate data. Tags can be a simple, single element, or an array, or structure. There are four types of tags in Logix:

- **Base**: These tags are the default variable tag type in Logix. Base tags allow you to specify a unique name-based tag to store and manipulate data in your program.
- **Alias**: These tags allow you to assign your own unique name to a module channel, existing tag, or structure tag member. When you create an alias tag, you will also select what the alias tag is an alias for. They are most frequently assigned to module channels in order to improve code readability and ease of maintenance. For example, if you consistently refer to a digital input module's channel as an alias tag and the channel wiring changes to a new location, you can easily update your code by only changing the configuration of your single alias tag.
- **Produced**: This tag allows you to pass a tag's value to a remote Logix controller at a predictable (real-time) frequency. A produced tag is always paired with a consumed tag on the controller that reads the tag's value.
- **Consumed**: This tag allows you to receive a tag's value from a remote Logix controller at a predictable (real-time) frequency. A consumed tag is paired with produced tag on the controller that is sending the tag value.

Each tag that is created in Logix is also assigned with a scope. The scope defines the area within our Logix project that the tag can be accessed from. The scope can be configured as either of the following levels:

- **Controller level**: This level is globally accessible across all the routines.
- **Program Level**: This level is accessible only within a single program. The program scope is selected during the tag configuration.

It is important to note that the tag scope cannot be easily changed once it is configured. So, care should be taken prior to selecting the tag's scope level.

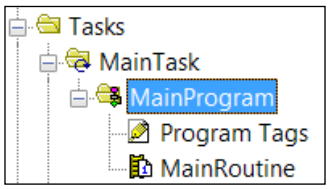
Buffering base tags

As we will learn in *Chapter 9, Using Tasks and Programs for Project Organization*, Logic can be organized into manageable pieces and executed based on different intervals and conditions. The practice of buffering base tags takes advantage of Logix's ability to organize code into routines. The default ladder logic routine that is created in every new Logix project is called `MainRoutine`. In the following exercise, we will be editing the `MainRoutine` ladder logic program and adding these three routines that will be called by it:

- One for reading the input values
- One for executing logic
- One for writing the output values

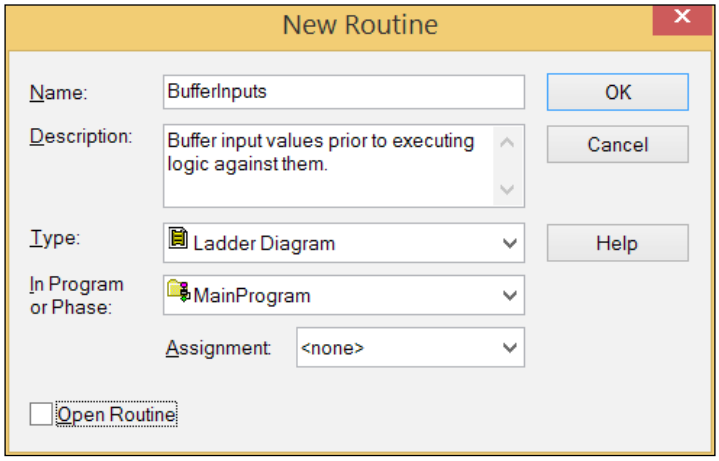
Open the Logix project we created earlier in this chapter with the digital input and output modules and follow these steps:

1. Using the **Control Organizer** pane, open the `MainProgram` folder, which can be found by navigating to **Tasks** | **MainTask**:



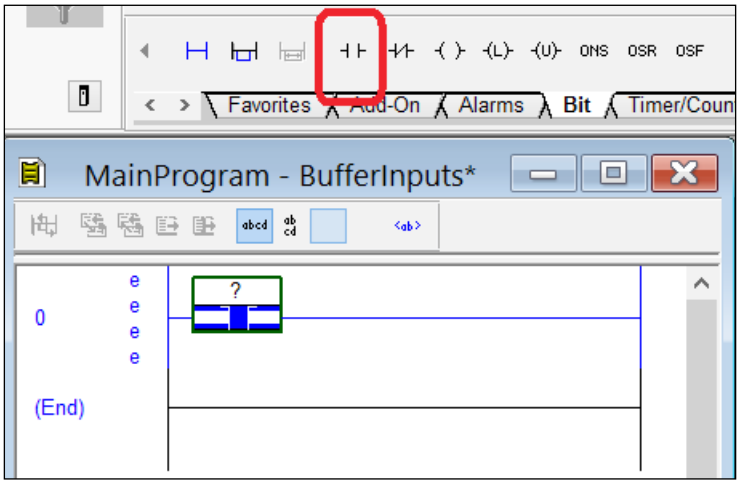
2. Right-click on **MainProgram** and select the **New Routine…** option.

3. In the **New Routine** window that appears, set the following field values:
    - **Name**: `BufferInputs`
    - **Description**: `Buffer input values prior to executing logic against them.`
    - **Type**: **Ladder Diagram**
    - **In Program or Phase**: **MainProgram**
    - **Assignment**: **<None>**

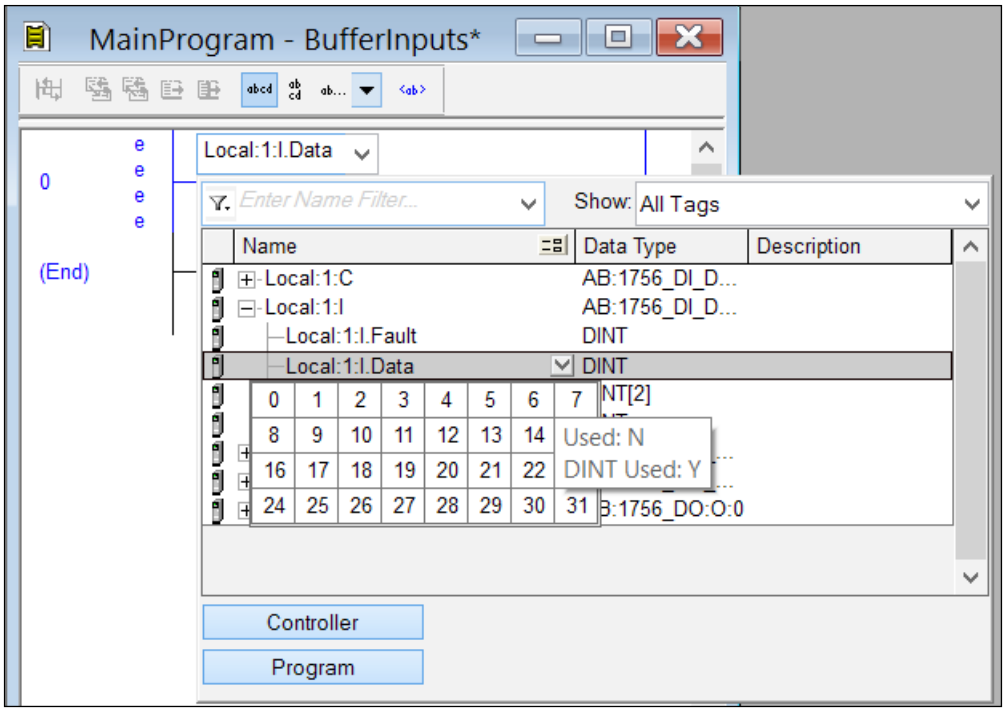


4. Create two more new routines. Use the following configurations for the first one:
    - **Name**: `PumpControl`
    - **Description**: `Pump control routine.`
    - **Type**: **Ladder Diagram**
    - **In Program or Phase**: **MainProgram**
    - **Assignment**: **<None>**
5. Use the following configurations for the second one:
    - **Name**: `BufferOutputs`
    - **Description**: `Buffer write output values after executing logic.`
    - **Type**: **Ladder Diagram**
    - **In Program or Phase**: **MainProgram**
    - **Assignment**: **<None>**

6. Now, let's start some ladder logic programming! Open up the **BufferInputs** routine.
7. Ladder logic programs are primarily created through drag and drop (although if you prefer good old coding, you can always right-click on a rung and select **Edit Code** to type in the logic).
8. Now, let's configure a contact from our digital input module to write a value (buffer it) to our coil base tag value.
9. First, we will add a ladder rung for buffering a digital input module channel for a start pump button signal to a base tag.
10. Above our ladder logic routine, you will find the ladder logic element groups and elements. These can be dragged and dropped into our ladder logic routines. Under the **Bit** element group, you will see our contact element (known as **Examine On** in Logix) and our coil element (known as **Output Energize** in Logix). Drag the **Examine On** element to the left of our **BufferInputs** ladder rung 0, as shown in the following screenshot:

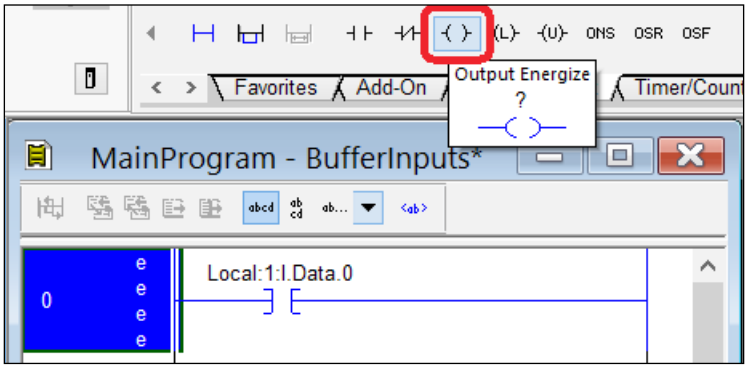


11. Now, we can assign the **Examine On** value to a channel of our digital input module. Double-click on the question mark above the **Examine On** element and select or type **Local:1:I.Data.0** in the module and channel, as shown in the following screenshot:

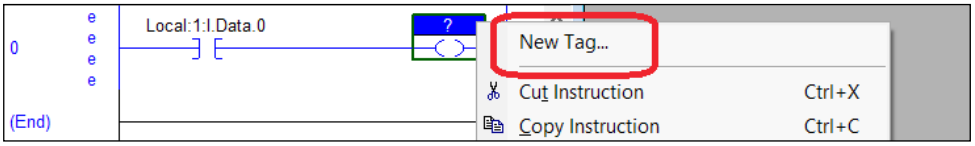


A common practice for handling the module inputs and outputs in a Logix program is to assign them to aliases and reference only the aliases throughout your program. This allows you to change the location of the module value easily in the future if required. However, as long as we only reference our module inputs and outputs in our buffering routines, we might not clutter up our tag list with module alias tags.

12. Next, we will add the contact, known as **Output Energize** in Logix, to rung 0. Find the **Output Energize** element in the **Bit** element group and drag and drop it into our rung.

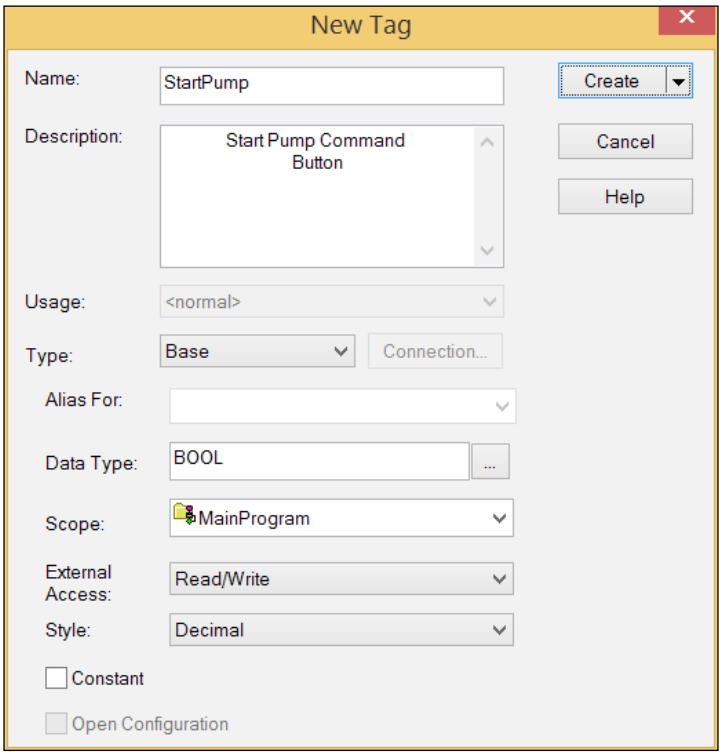


13. Now, we will assign the value of our input module channel **0** to a base variable using the **Output Energize** element. Right-click on the question mark above the **Output Energize** element and select **New Tag…**, as shown in the following screenshot:

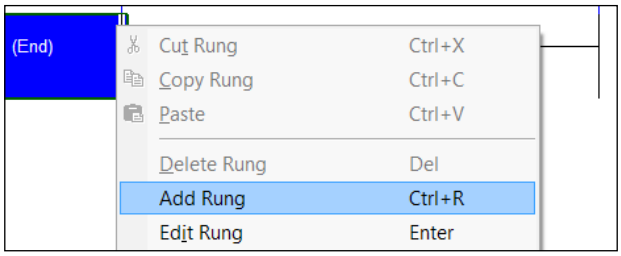


It is possible to interlace **Examine On** (input contacts) and **Output Energize** (coils) across a ladder logic rung. The **Output Energize** elements will evaluate as the value that is being written to it in the logic. However, mixing coils and contacts in the middle of a rung can create ladder logic that is difficult to follow.

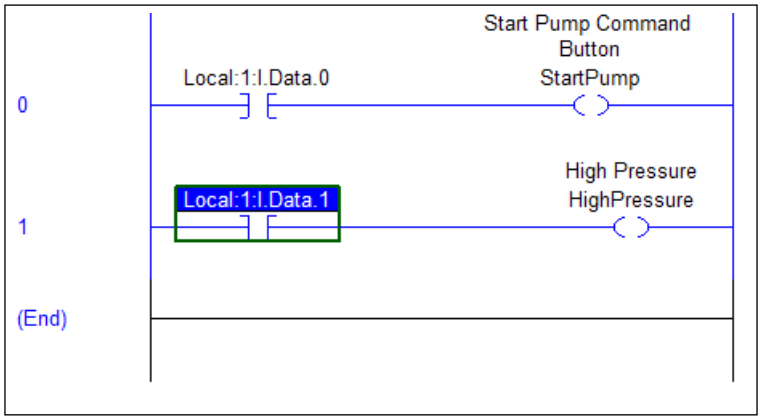
14. The **New Tag** window will appear and allow you to set the following parameters for a newly created tag:
    - **Name**: `StartPump`
    - **Description**: `Start Pump Command Button`
    - **Data Type**: **BOOL**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**



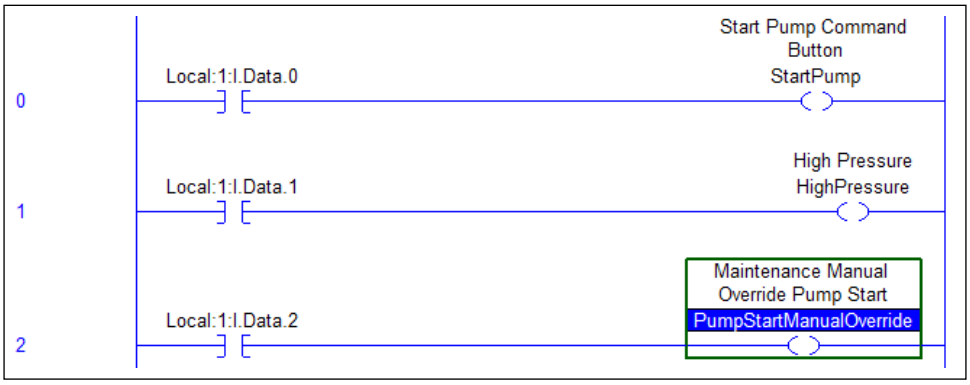
15. Next, we will be adding another rung to our **BufferInput** routine in order to buffer a **High Pressure** digital input module channel signal to a base tag. Right-click on the ladder logic rungs and select **Add Rung**, as shown in the following screenshot:



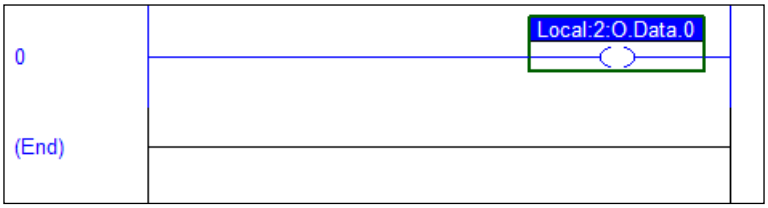
16. Drag an **Examine On** element into our newly added rung 1, and set the value to **Local:1:I.Data.1**.
17. Drag an **Output Energize** element into rung 1, and create a new base tag with the following parameters:
    - **Name**: `HighPressure`
    - **Description**: `High Pressure`
    - **Data Type**: **BOOL**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**



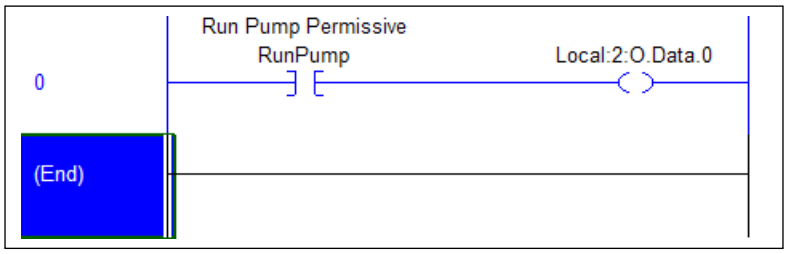
18. We will add one more ladder logic rung for **Maintenance Manual Override Pump Start**. Add a new ladder rung (rung 2) and add **Local:1:I.Data.2** as the **Examine On** element. Then, add the **Output Energize** element and base tag with the following parameters:
    - **Name**: `PumpStartManualOverride`
    - **Description**: `Maintenance Manual Override Pump Start`
    - **Data Type**: **BOOL**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**



19. Now that we have buffered the digital inputs for our simple ladder logic program, we will now buffer the outputs. Our output will be **Run Pump Permissive** from our program. The pump will only run if the **Start** button is being clicked on and the **High Pressure** value is not present. Open the **BufferOutputs** ladder logic routine we created earlier and drag an **Output Energize** element into rung 0. Now, set the **Local:2:O.Data.0** output tag value to channel 0 of our digital output module.

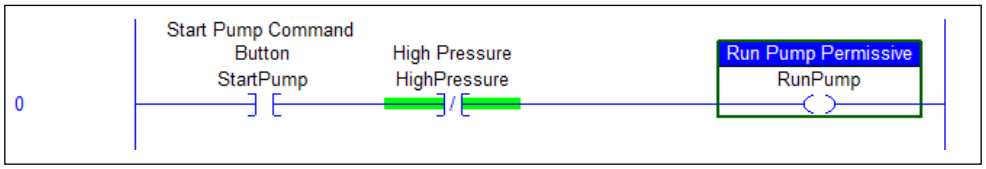


20. Next, we will add our **Examine On** element and create a new base tag with the following parameters:
    - **Name**: `RunPump`
    - **Description**: `Run Pump Permissive`
    - **Data Type**: **BOOL**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**



21. We have now buffered our module inputs and module outputs in order to ensure they do not change in the middle of a program's execution and potentially put our process into an undesired state. We will use the AND and NOT logic to control our pump using our buffered base tags. Open the **PumpControl** routine we created.
22. Drag an **Examine On** element (contact) into ladder rung 0 in the **PumpControl** routine and assign the element to our **StartPump** base tag.
23. Drag an **Examine Off** element (looks similar to the **Examine On** element with a line through it) into ladder rung 0. You will notice that the **Examine Off** element appears beside the **Examine On** element, creating an AND logical expression. Assign the **Examine Off** element to the **HighPressure** base tag we created in our buffering routine. The **High Pressure** Examine Off element will protect our process from being over pressurized by shutting down the pump if the pressure is too high.

24. We will complete our ladder logic rung by adding the **Output Energize** element to ladder rung 0 and assigning it to our buffered output base tag, **RunPump**.



The **HighPressure** Examine Off element uses a NOT logic expression, which is illustrated by the slash across the element symbol. This element will evaluate as true when the **HighPressure** digital input reads false. With the two elements side by side, our ladder logic rung now utilizes the AND logic expression. Both **StartPump** and **HighPressure** must evaluate as true in order to energize the **RunPump** output coil. We created the following logic expression in our rung:

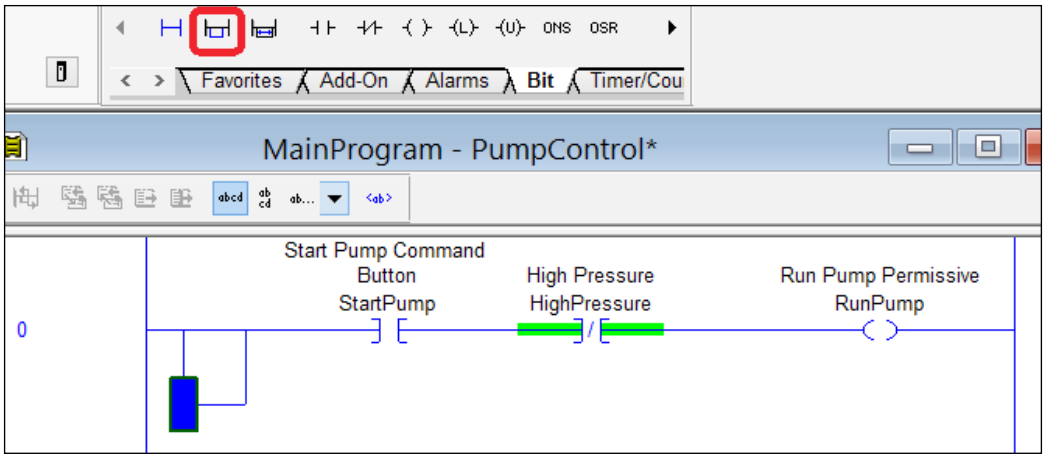
```
IF StartPump = True AND HighPressure = False THEN RunPump
```

When creating logic expressions, the truth tables can be a helpful tool to understand all the possible input combinations and their outputs. The following truth table shows all the input and output combinations of our ladder logic rung:

StartPump	HighPressure	RunPump
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	False (0)
True (1)	True (1)	False (0)

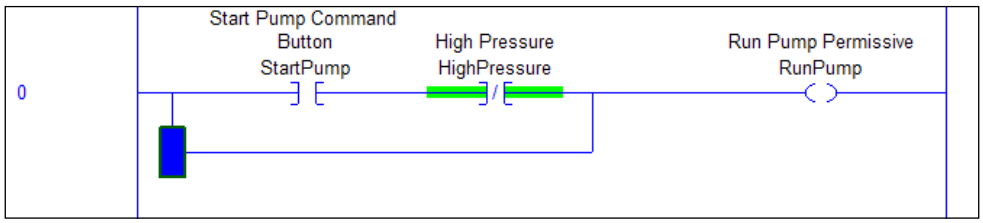
We still need to add our **Maintenance Manual Override Start** element for our pump control logic. This feature will allow the operator to run the pump regardless of the process conditions. In order to implement this feature, we will add a branch to our ladder logic rung, which evaluates as an OR logic expression by performing the following steps:

1. The branch element is the second element in all of the element groups. Drag the branch element on to rung 0.

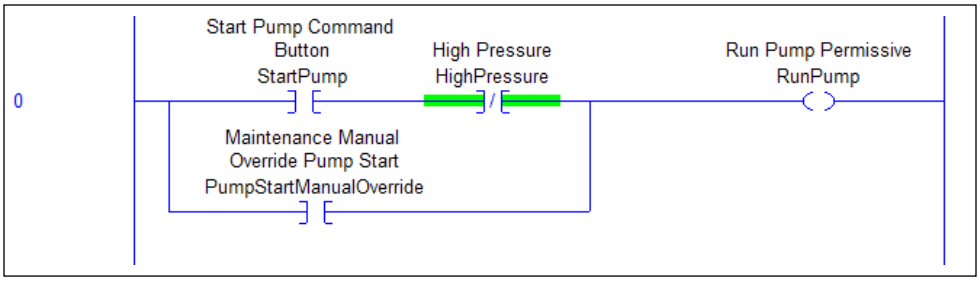


It is possible to nest multiple branch elements to create a very complex OR logic expression.

2. In order for the OR logic expression to override all our other module inputs, we will need to drag and drop the **StartPump** and **HighPressure** elements to the top line of the branch element.



3. Adding **Maintenance Manual Override Pump Start** is now just a matter of dragging an **Examine On** element into the bottom of the branch. Then, associate the element with the **PumpStartManualOverride** base tag.



We have updated our rung to override the **Start Pump** and **High Pressure** inputs with **Maintenance Manual Override Pump Start** and created the following logic expression:

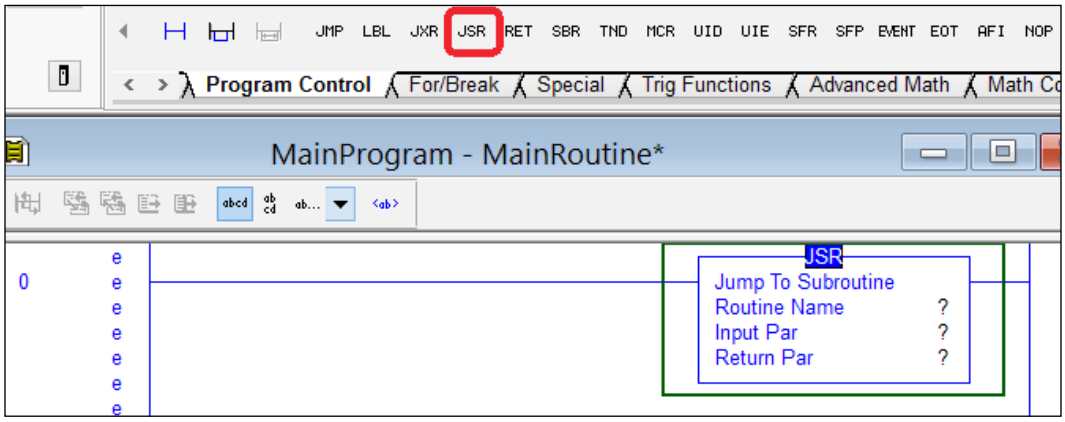
```
IF (StartPump = True AND HighPressure = False)
OR
PumpStartManualOverride = True
THEN RunPump
```

The following truth table shows all the input and output combinations of our updated ladder logic rung:

StartPump	HighPressure	PumpStartManualOverride	RunPump
False (0)	False (0)	False (0)	False (0)
True (1)	False (0)	False (0)	True (1)
False (0)	True (1)	False (0)	False (0)
False (0)	False (0)	True (1)	True (1)
True (1)	True (1)	False (0)	False (0)
False (0)	True (1)	True (1)	True (1)
True (1)	False (0)	True (1)	True (1)
True (1)	True (1)	True (1)	True (1)

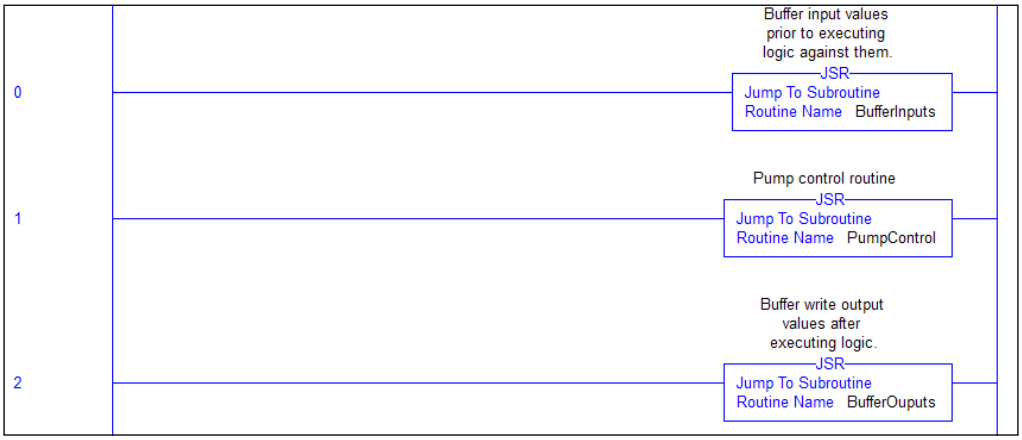
Currently, our routine will not be executed in the program on our Logix controller because **MainProgram** (which contains our ladder logic routines) is configured to run the **MainRoutine** program. In order for our routines to execute, we must reference them in the **MainRoutine** program. The **Jump To Subroutine** ladder logic element will allow us to execute the routines we created from the **MainRoutine** program:

1. Open the **MainRoutine** program and find the **JSR** element in **Program Control** element group and drag it into rung 0.



2. Now, we can associate the **JSR** element with our **BufferInputs** routine by setting the **Routine Name** parameter to `BufferInputs`.
3. We will need to remove the **Input Par** and **Return Par** parameters from the **JSR** element by right-clicking on the **JSR** routine and selecting **Remove Instruction Parameter** on both the **Input Par** and **Return Par** parameters.

4. In order to add the **JSR** elements for our other routines, we will need to add two more ladder rungs and add the **JSR** elements with the following parameters:
    - **JSR Rung 1**: `PumpControl`
    - **JSR Rung 2**: `BufferOutputs`



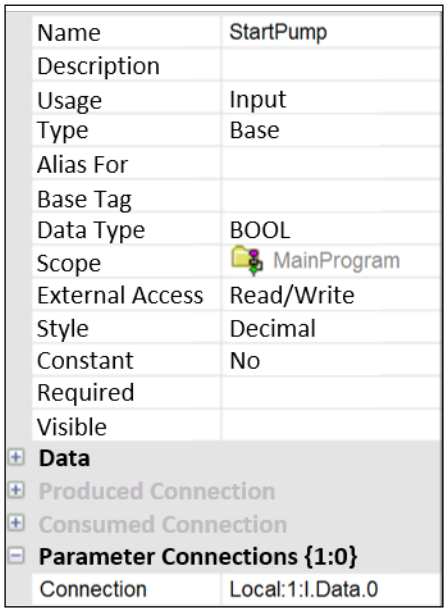
We completed our first ladder logic program. Ladder logic can be easily created by dragging and dropping elements into a rung. There are far too many elements available in Logix to cover in this book, but you can discover them yourself by dragging them into a rung and right-clicking on them and selecting **Instruction Help**. In addition to it, Rockwell Automation has a number of resources on ladder logic programming in their Literature Library Resources (see the appendix of this book for some useful links).

Buffering using program parameters

A program parameter is a powerful new feature in Logix that allows the association of dynamic values to tags and programs as parameters. The importance of program parameters is clear by the way they permeate the user interface in newer versions of Logix Designer (Version 24 and higher). Program parameters are extremely powerful, but the key benefit to us for using them is that they are automatically buffered. This means we could have effectively created the same result in one ladder logic rung rather than the eight we created in the previous section. There are the following four types of program parameters:

- **Input**: This program parameter is automatically buffered and passed into a program on each scan cycle.
- **Output**: This program parameter is automatically updated at the end of a program (as a result of executing that program) on each scan cycle, similar to the way we buffered our output module value in the previous section.
- **InOut**: This program parameter is updated at the start and the end of the program scan. It is also important to note that unlike the Input and Output parameters, the InOut parameter is passed as a pointer in memory. A pointer shares a piece of memory with other processes rather than creating a copy of it. This means that it is possible for an InOut parameter to change its value in the middle of a program scan. This makes InOut program parameters unsuitable for buffering when used on their own.
- **Public**: This program parameter behaves like a normal controller tag and can be connected to Input, Output, and InOut parameters. Similar to the InOut parameter, Public parameters are updated globally as their values are changed. This makes program parameters unsuitable for buffering when used on their own. Primarily, Public program parameters are used for passing large data structures between programs on a controller.

In Logix Designer Version 24 and higher, a program parameter can be associated with a local tag using **Parameters and Local Tags** in the **Control Organizer** pane (formally called program tags). The module input channel can be associated with a base tag within your program scope using the **Parameter Connections** option. Add the module input value as a parameter connection.



The preceding screenshot demonstrates how we will associate the input module channel with our **StartPump** base tag using the parameter connection value.

Summary

In this chapter, we explored the genesis of ladder logic programming and the IEC programming language standards. We learned how to create ladder logic by dragging and dropping elements into a ladder rung in a routine. We also learned the importance of buffering inputs and outputs and some techniques for accomplishing this.

In the next chapter, we will introduce another IEC language called **Function Block Programing** and its use with the Logix family.

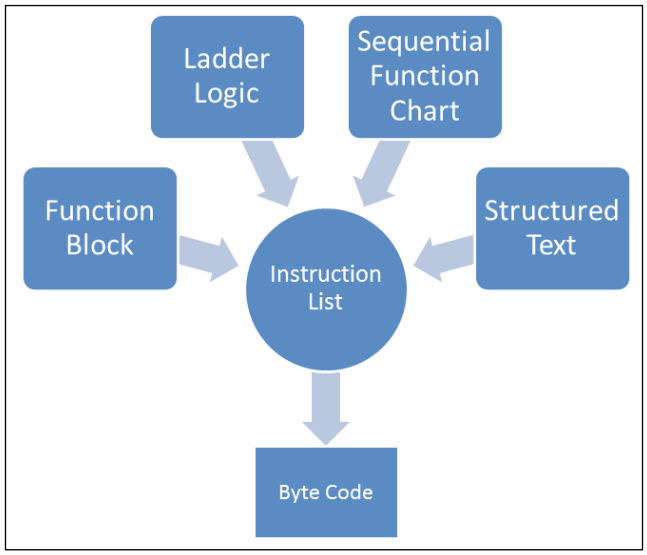
6
Writing Function Block

In this chapter, we will explore the merits of function block programming by building a small sample application. We will also provide instructions for modifying the function block properties and performing online edits. The following function block topics will be covered in detail in this chapter:

- Function block versus ladder logic
- Function block sheets
- Function block elements
- Function block wiring
- Function block logic
- Online monitoring and editing function blocks
- Function block textboxes
- Assigning constant values

Language compilation overview in Logix

Logix Designer, like most IEC 61131-3 compliant applications, will take any program you create in any IEC-compatible language, convert it to **instruction list** (**IL**)—a low-level language that resembles Assembly), and compile it down to bytecode (the binary language used internally by the controller) in order for the controller to execute it. The following diagram illustrates the way various languages are all compiled down into the same bytecode language:

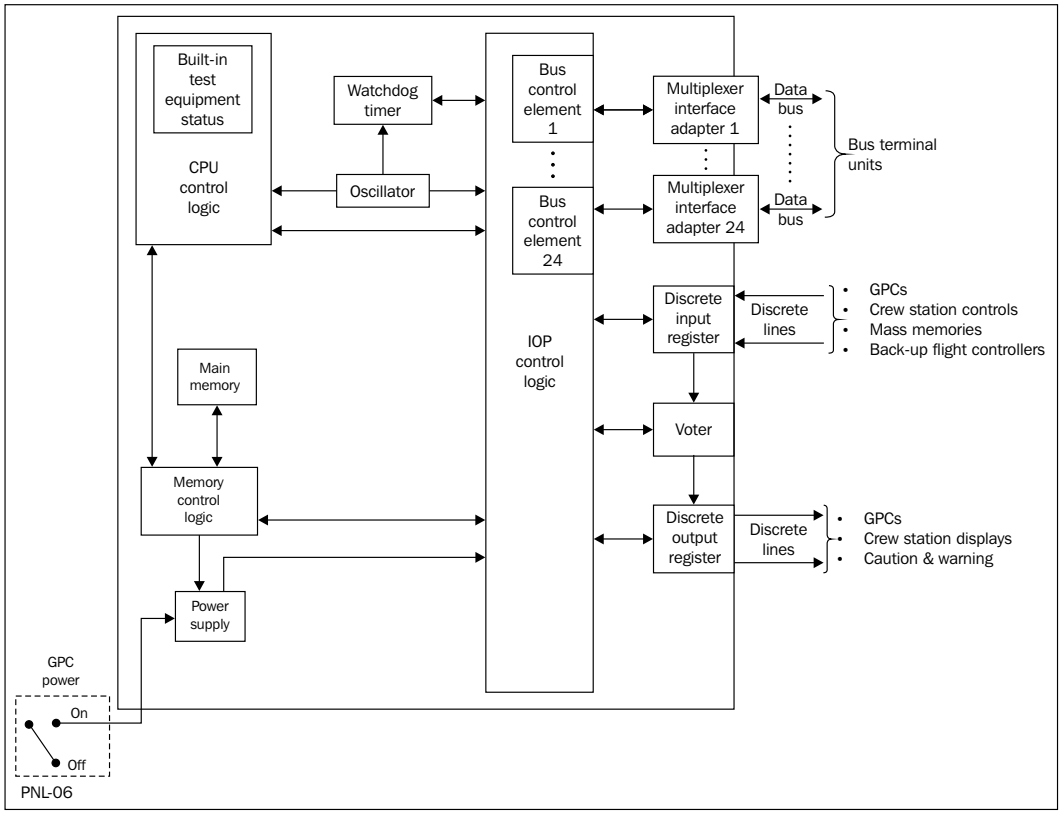


Language compilation to bytecode in Logix

Ultimately, the controller is not aware of which language you created your program in (**Ladder Logic**, **Function Block**, or **Sequential Function Chart**); it always ends up as the same **Bytecode** language that is executed on the controller. Within Logix Designer, you can create the exact same program using ladder logic that you would create using **function block diagram** (**FBD**), and it would (in theory) translate down to the same **Structured Text** (**ST**) commands and compile down to the same Bytecode for the controller. It is important to understand the direct relationship between ladder logic, function block, and other IEC languages.

The function block overview

The FBD programming language, which I refer to as function block, is a graphical flow diagram language where program instructions appear as blocks (function blocks). Each function block has inputs and outputs that can be wired to other function blocks to create a visual representation of data flow. In the same way that ladder logic is based on the Relay Logic Engineering drawings, FBDs are also derived from engineering discipline standards. Before being used as a programming language, FBDs were used in system / software engineering to describe the interrelationships between electronic systems. The following FBD is from NASA's Space Shuttle program and describes the electronic system relationships inside an IBM AP-101S general purpose computer.



Within Logix, it is important to note that editing FBDs is not supported by all editions of the Logix programming software. Only Professional Edition, Full Edition, and Lite Edition allow you to write programs using FBD; however, all the editions will allow you to upload and download the existing FBD programs. It is important to understand the version and edition of Logix you are using and the capabilities that have been enabled. Refer to the links in the appendix of this book or Rockwell Automation Literature Library for more information on the editions of Logix that are available.

Understanding FBD

In this section, we will explore the basic elements of an FBD within the Logix platform.

Function block versus ladder logic

As mentioned earlier, both FBD and ladder logic eventually will compile down to the same controller bytecode language. The available functions and development interface in these two programming languages are vastly different, and it is important to highlight these differences. Ladder logic, as you will recall from the preceding chapter, is executed from the top of the ladder to the bottom and from the left-hand side of the rung to the right-hand side. Function block also executes from left to right, so if you want a particular function block to execute prior to other, position it more to the left than the other blocks on the page. The inputs and outputs also help Logix to determine the order of execution for the function blocks on a sheet.

The following table lists some of the notable differences between ladder logic and function block:

Ladder logic	**Function block**
This language is executed top to bottom and left to right.	This language's execution order is determined by the input and output connections and the horizontal position on the sheet (left to right).
In this language, the input values can change during execution, so buffering the input data is recommended.	Here, input values are read only once at the start of the execution, so buffering the input values is not required.
This language uses normally open or normally closed contacts.	This doesn't use normally open or normally closed contacts.
This language is organized by ladders, which execute in a sequential order.	This language is organized by sheets. All the sheets execute at the same time.

Ladder logic	Function block
This language is a low-level language. Ladder elements have very basic functionality and consume fewer processor cycles. It produces more code to maintain.	This language is a high-level language. Function block elements have very powerful functionality and consume more processor cycles. This language produces less code to maintain with more powerful functions.
This language uses the controller and program-scoped tags.	This language uses the controller and program-scoped tags.

Ultimately, when choosing weather to use ladder logic or function block to write a routine, it comes down to selecting the right tool for the job. Some problems are better suited for the powerful features of function block and others problems would benefit from the low-level control provided by ladder logic. When selecting a language to solve a problem, you should try to select a language that will allow you to strike the balance between:

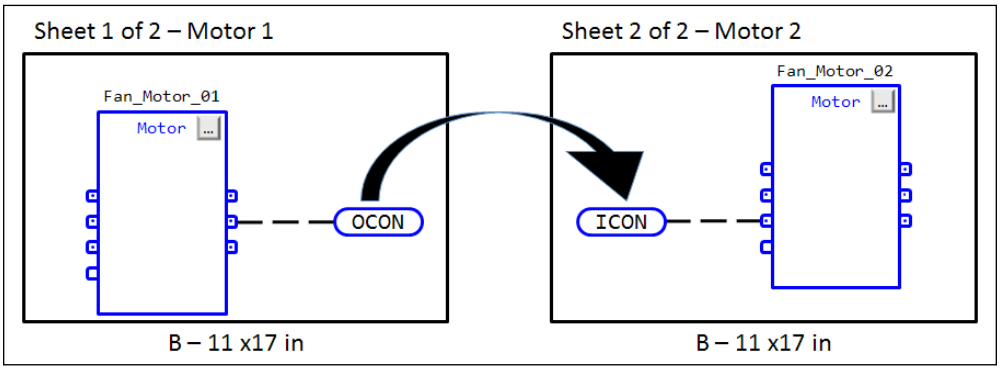
- Ease of development
- Ease of maintenance
- Efficient use of processor cycles

The function block sheets

Within a routine, FBDs are created in an area called a sheet, and the sizes of the sheets directly correspond to standard metric or English printer page sizes (we use ledger / 11 x 17 inch in our following example). This allows the sheets to be easily printed, presented, and even signed like a typical engineering drawing. It is helpful to think of each sheet as a drawing for a single device.

You can adjust the size of a sheet by right-clicking on an empty area of the function block routine and selecting **Properties**.

You can connect multiple sheets together using the input wire connectors and output wire connectors; we will discuss connectors in more detail later in the chapter. Each sheet has **Sheet Number** and **Sheet Name** to identify it. A routine can contain an unlimited number of sheets, but it is important to understand that all the sheets are executed at the same time (not one sheet at a time). Here is a diagram displaying two sheets, each containing a function block wired together using the input and output wire connectors:

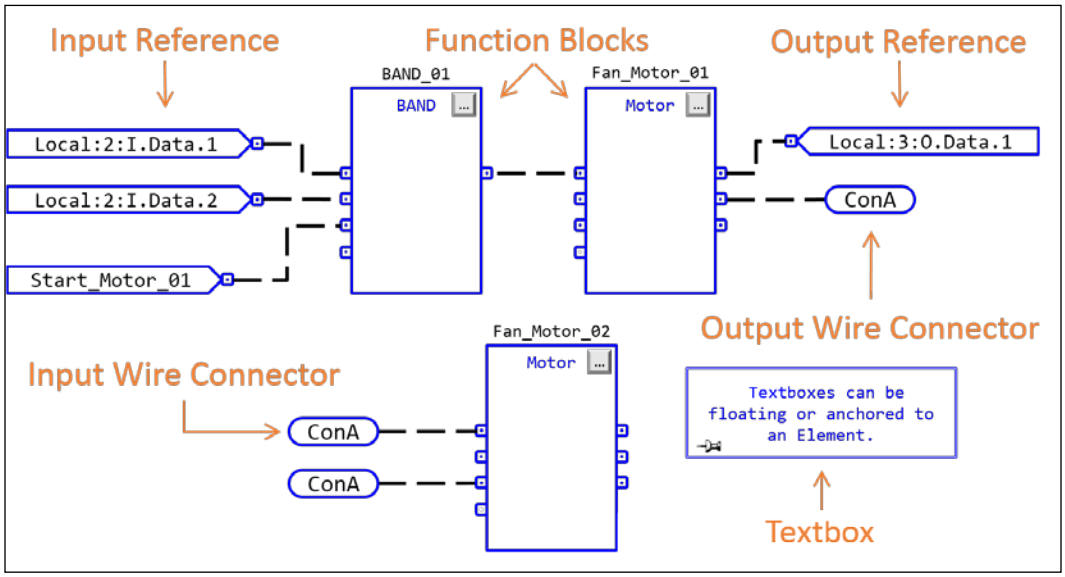


The function block elements

There are the following six elements used within an FBD:

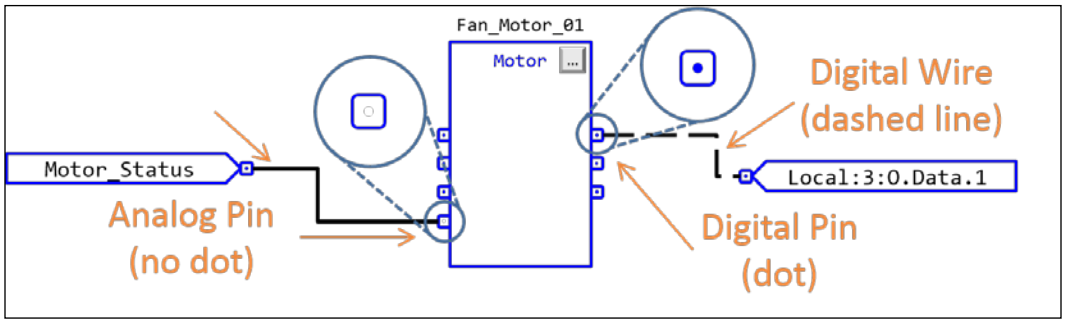
- **Input reference** (**IREF**): These elements are the input tag values that are read into a function block routine before it executes.
- **Output references** (**OREF**): These elements are the output tag values that are written to once the function block routine completes its execution.
- **Input wire connectors** (**ICON**): This element receives data from another function block that is on a different sheet within the same routine or far apart on the same sheet.
- **Output wire connectors** (**OCON**): This element sends data to another function block that is on a different sheet within the same routine or far apart on the same sheet. A single output connector can be wired to multiple input connectors.
- **Function block** (**FB**): This element executes an operation based on the values of its input pins and then provides results to its output pins.
- **Textbox**: This element is used to provide the code comments to the function block routines.

The following function block routine identifies these six elements and how they are connected:



Function block wiring

Within a single sheet, the function blocks are wired together in a similar fashion to an electronic circuit board. Wires are used to connect input tags to function blocks, function blocks to function blocks, and function blocks to output tags. Pins are used to connect wires to the function block elements. The following diagram demonstrates the use of wires and pins in a simple function block routine:



Function block wiring

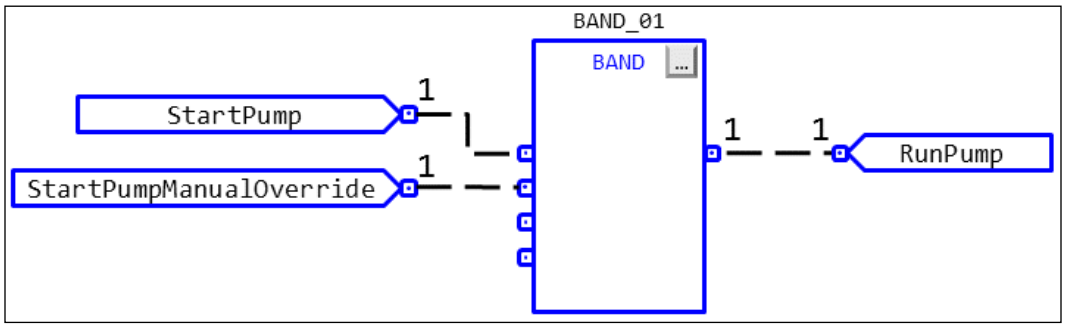
Note that there is a visible difference between the analog wires (solid lines), digital wires (dashed lines), analog pins (no dot in the center), and digital pins (dot in the center).

Function block logic

Where ladder logic uses the element position on a rung to create logic expressions, function block logic expressions are handled by dedicated function block elements. In the next sections, we will explore some of the logical function block elements that are available and see how they are used.

The AND logic function block

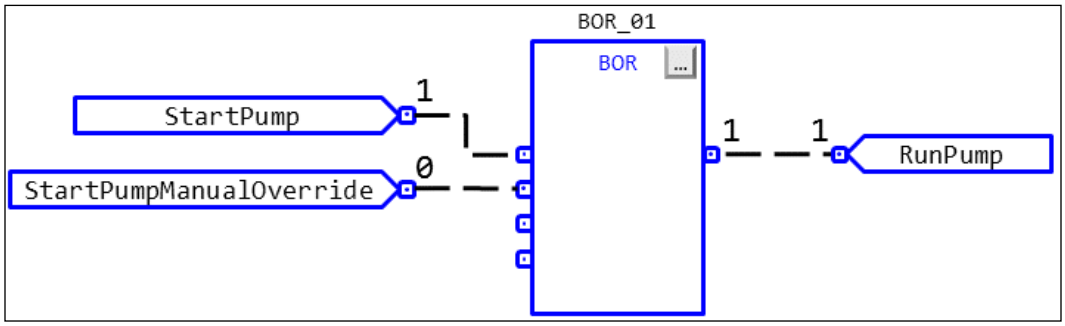
Within a function block routine, you can evaluate an AND logical expression using the **BAND** function block. This block will evaluate an AND logical expression using all the input references passed into and provide the solution to its output reference pin. The following diagram shows the same, simple AND logical expression in function block:



When the **StartPump** and **StartPumpManualOverride** input references are both true, the **RunPump** output reference will energize. Note that the input reference values to the right on the input reference pins indicate the current value of the references. Also, the output reference pin on the BAND function block displays the current value of the AND expression. Although we are only displaying four digital input references in this diagram, you can add up to eight digital input references to a **BAND** block.

The OR logic function block

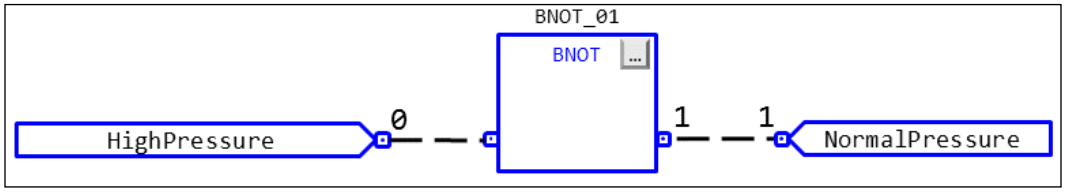
The **BOR** function block will accept and evaluate an OR logical expression against its input reference pins and provide output to its output reference pin.



The OR logic function block

The NOT logic function block

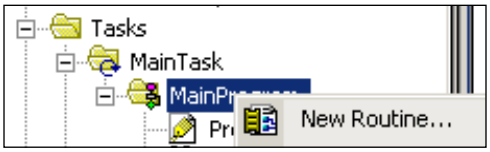
The **BNOT** function block will invert the value of the input reference provided. Passing a 1 value into the input reference pin will result in a 0 value in the output reference pin and passing 0 into the input reference pin will result in a 1 in the output reference pin. The following diagram illustrates a simple example of a BNOT function block:



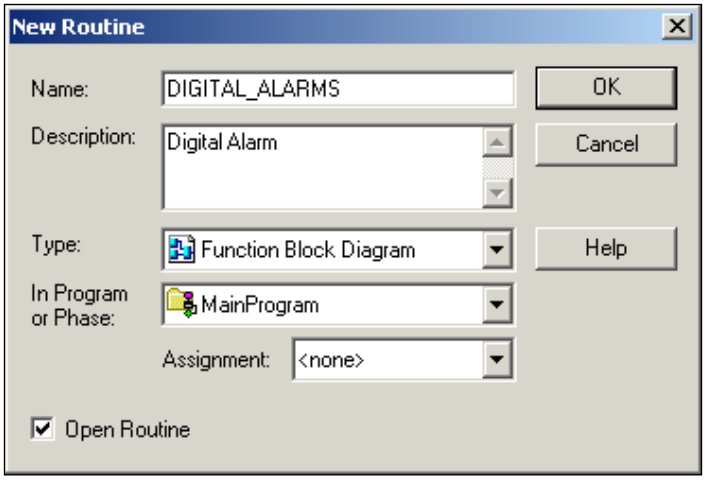
Writing a function block program

Now that we have covered the basic elements and logic functionality of FBDs, let's start to build our first function block routine. The following exercise will create a simple digital alarm routine using the **Alarm Digital** (**ALMD**) function block. Follow these steps:

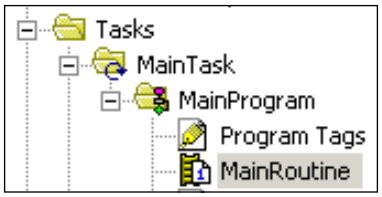
1. Open the **Controller Organizer** pane and expand the tree by navigating to **Tasks** | **Main Tasks** | **Main Program** and right-clicking and selecting **New Routine**:



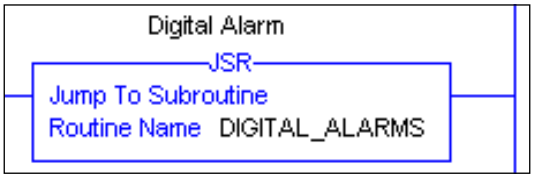
2. Configure a new functional block diagram routine by setting the following values:
    - **Name**: `DIGITAL_ALARMS`
    - **Description**: `Digital Alarms`
    - **Type**: **Function Block Diagrams**



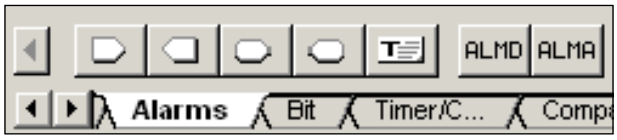
3. In order for our newly created routine to be executed with each scan of the PLC, we will need to add a reference to it in the **MainRoutine** program, which is executed with each scan of **MainTask**. Double-click on our **MainRoutine** program to display the **MainRoutine** ladder logic.



4. In the preceding chapter, we added **JSR** for our ladder logic diagram. We can simply copy and paste this ladder rung and change the value to point at our routine **DIGITAL_ALARMS**. Right-click on the left-hand side of the first ladder rung (where **0** is displayed) and select **Copy** (or press *Ctrl* + *C*).
5. Right-click below the first rung and select **Paste** (or press *Ctrl* + *V*).
6. Now double-click on the **Routine Name** parameter of the **JSR** element and select our newly added **DIGITAL_ALARMS** routine.

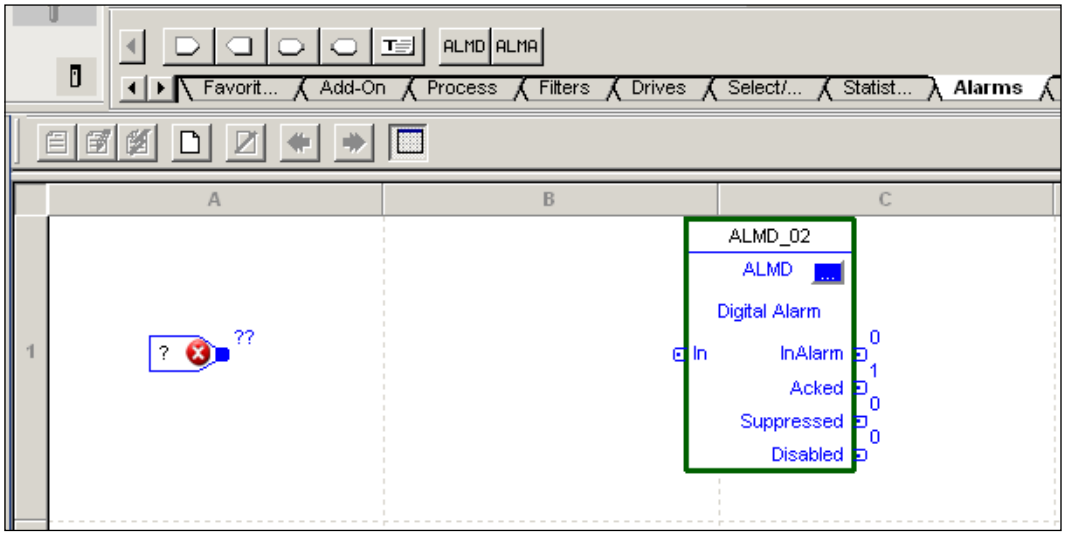


7. Now, we will return to our **DIGITAL_ALARMS** FBD by double-clicking on it in the **Controller Organizer** pane.
8. Next, we are going to add our digital alarm FBD, which we will use to manage our valve alarm fault. Select the **Alarms** element group just above the function block diagram sheet and click on **ALMD**.



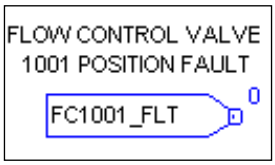
9. We need to connect the ALMD block to our valve fault alarm using an input reference, so let's add one to our FBD. The input reference element looks like an arrow (with a square corner), which is pointing to the right. It can be found at the top-left part of the element group selector above the FBD. Click on the input reference object icon to add it to the diagram.

10. Right-click on the question mark inside the input reference and select **New Tag**.

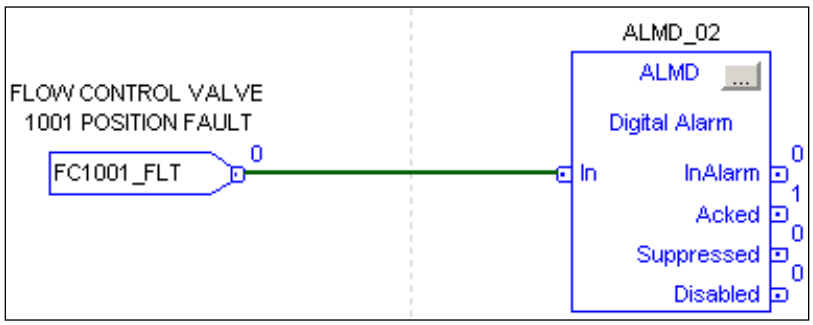


11. The **New Tag** window will appear and allow you to set the following parameters for a newly created tag:
    - **Name**: `FC1001_FLT`
    - **Description**: `FLOW CONTROL VALVE 1001 POSITION FAULT`
    - **Data Type**: **BOOL**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**

12. Now, our input reference is pointing to our fault base tag, **FC1001_FLT**:

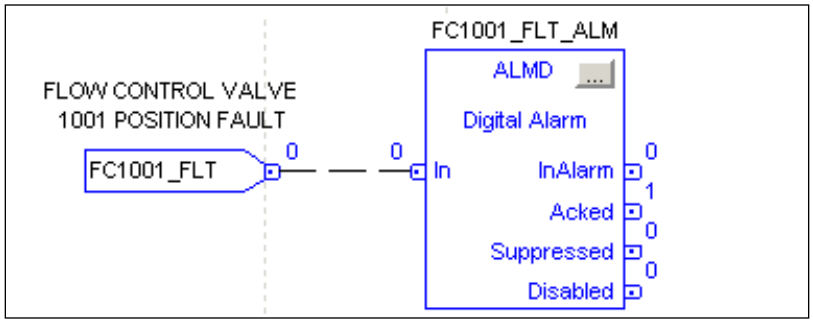


13. Now, we will need to reposition our blocks so that they fit properly on our FBD sheet. Click and drag the **ALMD** object a few inches to the right.
14. Now, we will connect the **FC1001_FLT** input reference to the **ALMD** block. Click and drag the point of the input reference (you will see the mouse pointer change to a connector mouse icon) and release the mouse button over the input digital pin.



15. The **ALMD** function block we added was automatically created as a base type object in our program-scoped tag list (program tags). We will now change the name of the **ALMD** object to follow our existing tag naming convention. Right-click on the top title of the **ALMD** object and select the **Edit ALMD_01** element.

16. Change the **Name** field of the element properties to `FC1001_FLT_ALM` and click on **OK**. The scope of our FBD base tag was set to the **MainProgram** scope automatically when we added it to our routine.

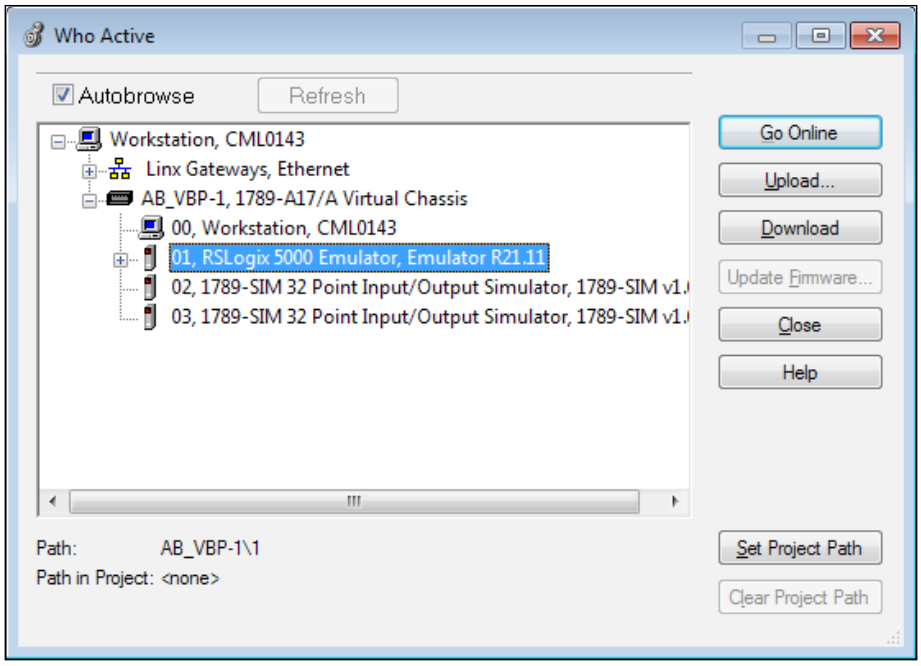


Online monitoring and editing

After completing a routine in any language, the next step in development is to thoroughly test it. Logix Designer incorporates powerful monitoring and debugging features, which can be used to test our routine. Perform the following steps:

1. First, we need to ensure that the communication path to our controller (physical or virtual) has been established. Open **Who Active** by navigating to **Communications** | **Who Active** or by clicking on the **Who Active** icon.

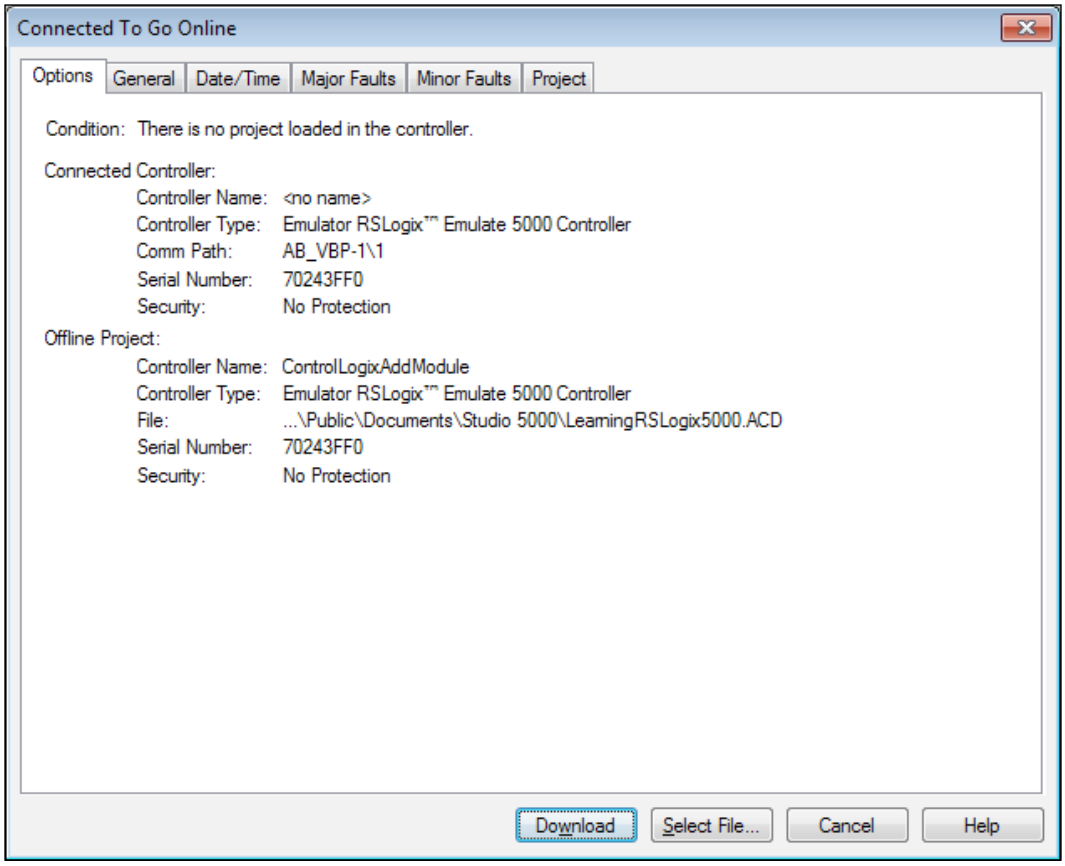
2. The **Who Active** window allows us to browse to the controller, which will run our program. Expand the **RSLinx Driver** tree and navigate to your virtual or physical controller. The following screenshot shows the selection of Virtual RSLogix 5000 Emulator in the tree:



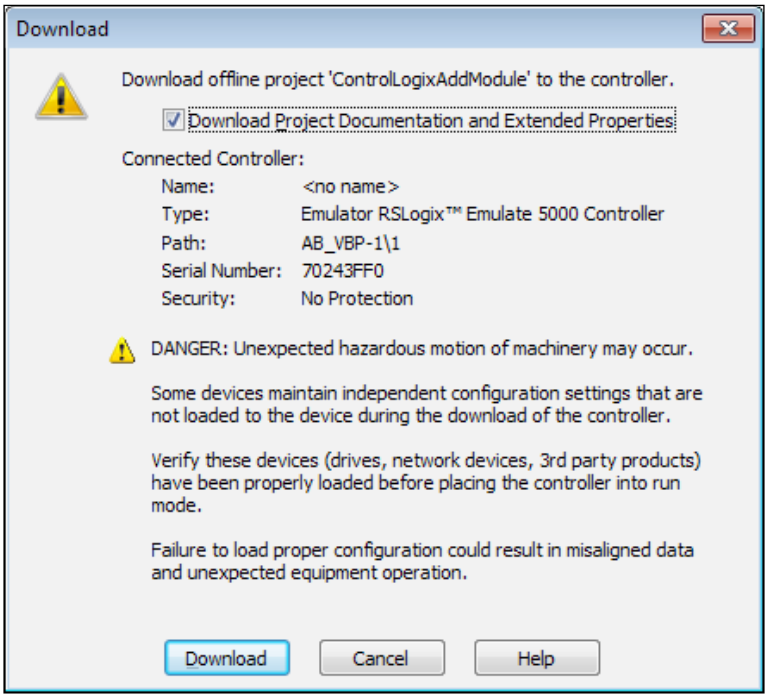
If you are having difficulties in finding your controller in the tree or the tree is empty, you may want to refer to the *RSLinx* section in *Chapter 2, Industrial Network Communications*.

3. After selecting a valid controller, the buttons on the right-hand side (as explained in the following list) will become enabled:
    - **Go Online**: This button will try to connect to the controller and start monitoring the execution of whichever program is currently running on it.
    - **Upload**: This button will upload the program currently running on the controller to your local computer.
    - **Download**: This button will download the current program you have open in Logix Designer to the controller.
    - **Update Firmware**: This button will allow you to upgrade the firmware on your controller.
    - **Set Project Path**: This button will update the path stored in your project file, which will automatically set this as the communication path the next time you open your project.

4. Click on the **Go Online** button. The **Connected To Go Online** window appears, providing the current status information of the controller and the options to **Download**, **Select File…**, or **Cancel**.



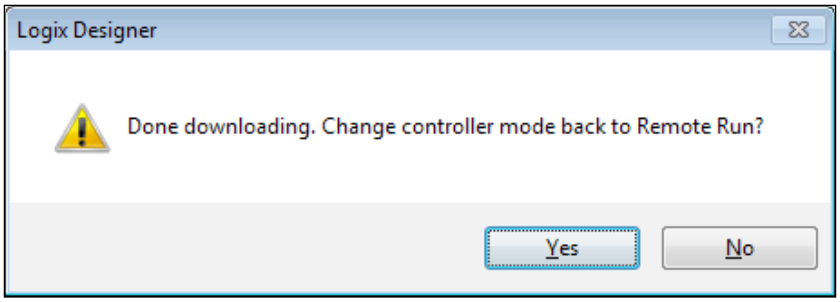
5. Click on the **Download** button. A **Download** window containing a safety message appears.



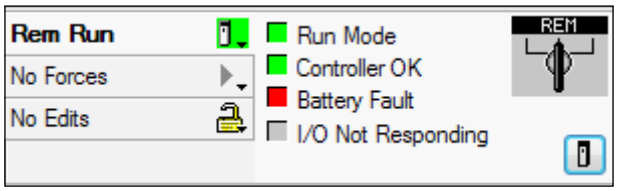
Downloading a program to a controller can cause a process to lose its state or trip. Always take proper safety measures to ensure that you will not put people or the facility at risk prior to working on operating equipment.

6. Click on the **Download** button on this window to copy your program into the controller memory.

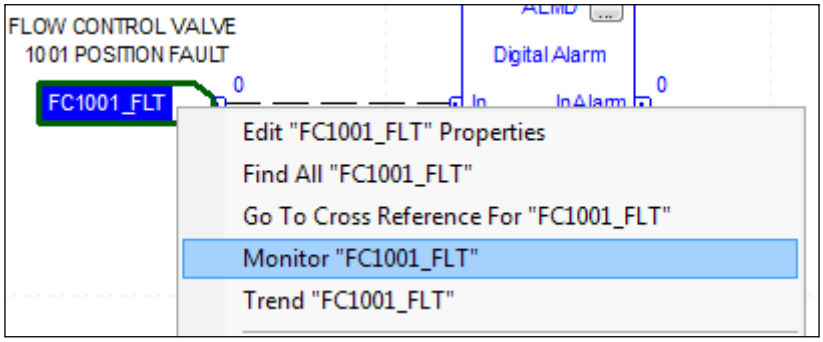
7. Once the **Download** process is complete, you may see a dialog box asking if you want to put the controller back into remote run. Click on **Yes**:



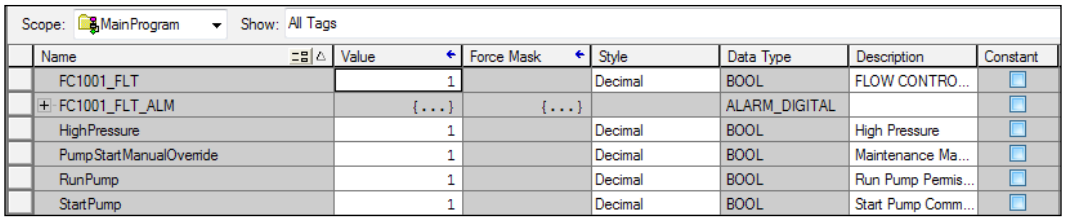
8. You will want to ensure that your controller is in **Run Mode** so that you can see live changes in your program. Now, we can start to monitor our tags and test our routines. Logix Designer should now show that we are in the **Rem Run** mode.



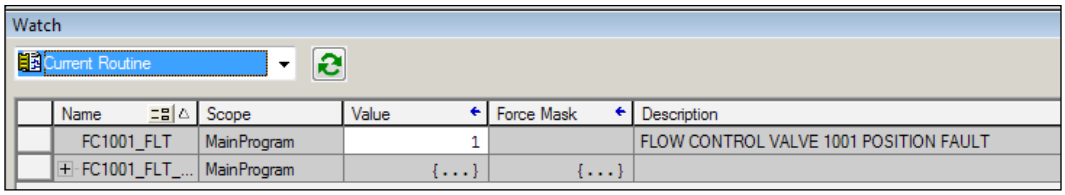
9. Now, return to our function block routine called **DIGITAL_ALARMS**.
10. Let's activate our digital alarm and observe the changes within our function block outputs. Right-click on the **FC1001_FLT** input reference and select **Monitor "FC1001_FLT"**.



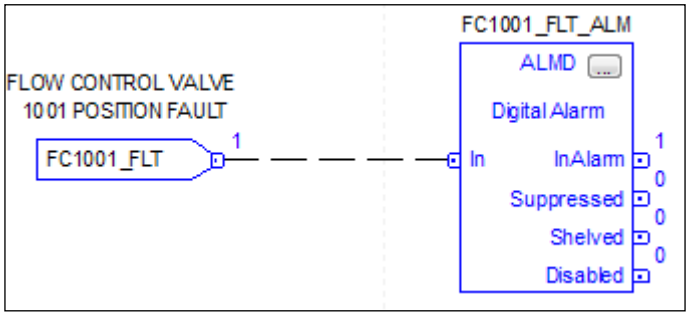
11. The **Monitor Tags** panel now appears in the foreground and provides us with a list of all the tags in the **MainProgram** scope. We can manually set the value of the **FC1001_FLT** tag to trigger our digital alarm function block. Enter **1** into the **Value** column of the **FC1001_FLT** tag and press *Enter*.



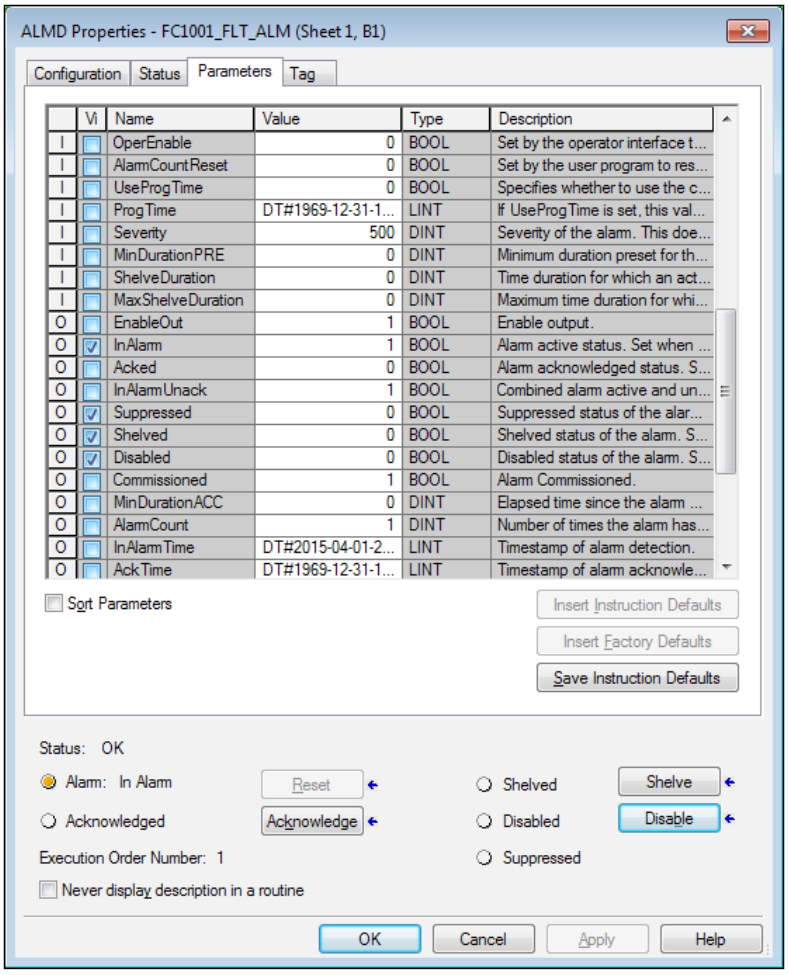
12. Alternatively, from the **DIGITAL_ALARMS** routine you can open the **Watch** panel by navigating to **View | Watch**. The **Watch** panel will appear directly under your FBD sheet and display a monitoring list of all the tags in the current routine.



13. Return to the **DIGITAL_ALARMS** routine by navigating to **Window | MainProgram – DIGITAL_ALARMS**.



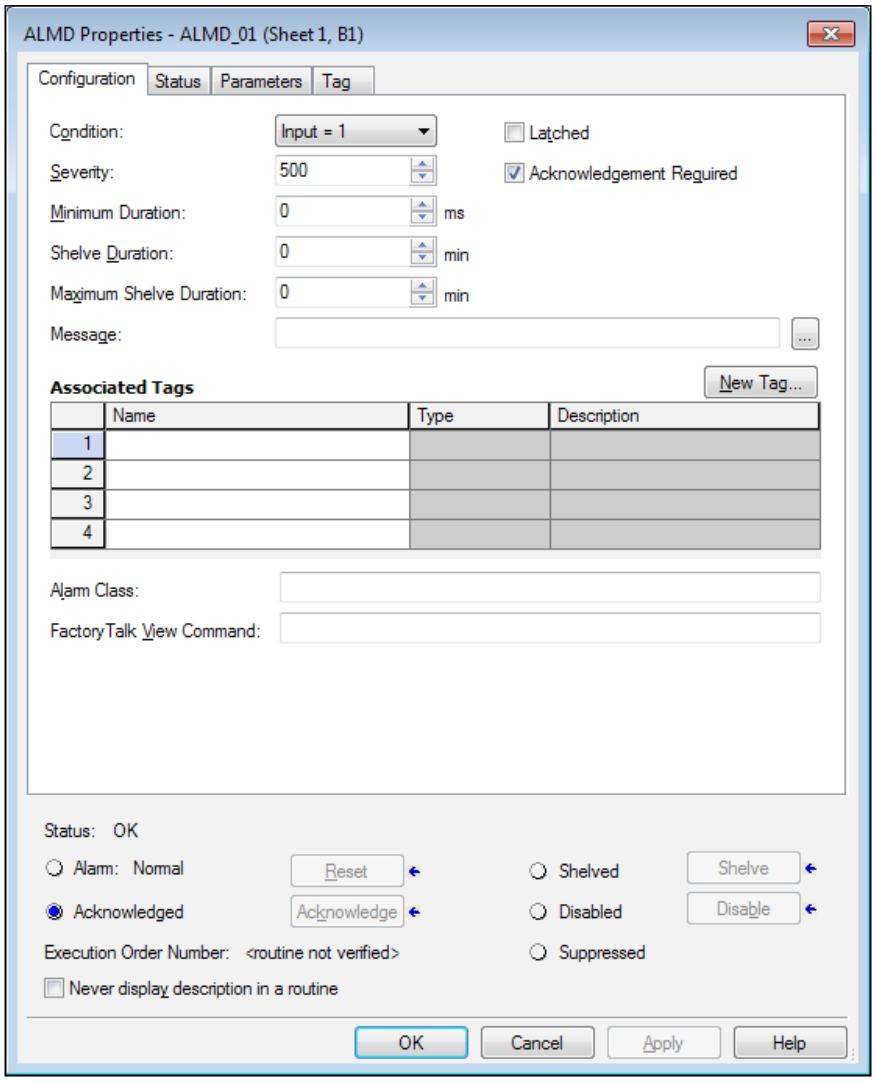
14. Now, we can see that the **InAlarm** output pin on **FC1001_FLT_ALM** has a value of **1**, indicating that the digital alarm function block is now in alarm. If you right-click on the digital alarm function block and select **Properties…**, you can see all the current **Parameter** values. These values can be referenced throughout your program in the various languages and also by the HMI/SCADA system that is providing the graphical user interface for your program.



Function blocks provide a high-level functionality and are relatively easy to configure and maintain. Function blocks especially work well with the Rockwell's FactoryTalk HMI program. FactoryTalk provides graphical faceplates, which align with the functionality of the Logix function blocks. Faceplates make it easy to provide a feature-rich control system user interface for operators.

The FBD properties

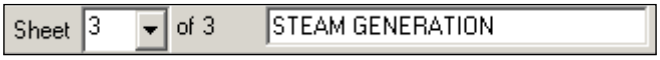
Double-clicking on an FBD block will open its properties. Each FBD block contains a unique set of properties and a detailed help documentation is provided (by pressing the *F1* key). Many of these properties allow you to more tightly integrate your PLC controller with your HMI computer. Using FBD can allow you to configure many properties, such as alarm names, in the PLC rather than in the HMI. Many SCADA system vendors are moving to a more DCS style, single database configuration. Rockwell Automation's PlantPAx automation system takes this type of DCS functionality to the next level, but that is a subject for a separate book perhaps.



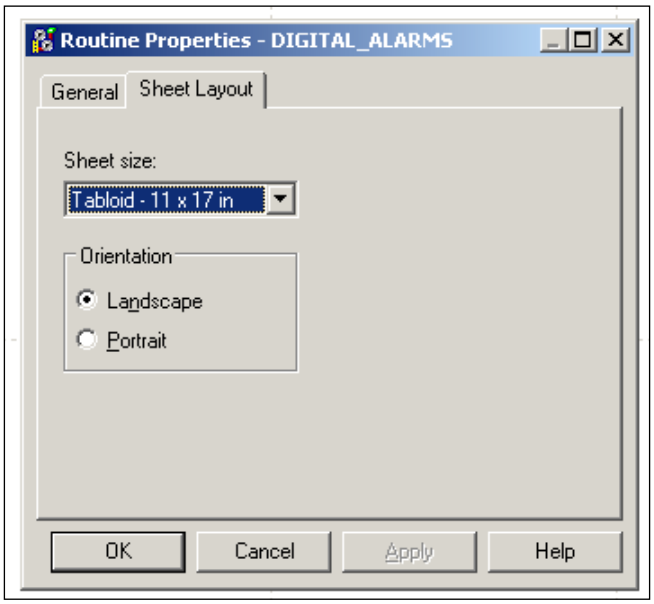
Adding and naming sheets to a routine

You can add sheets to your FBD by clicking on the **New Sheet** icon above your FBD routine.

You can also provide a helpful name for each sheet by editing the **Sheet** text field.



The **Sheet size** and **Orientation** fields can also be modified (using standard English and Metric page sizes) by right-clicking on the white space of a function block routine and selecting **Properties**.

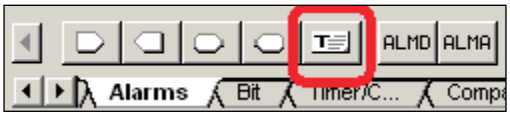


The layout flexibility and sheet organization that function block routines provide make them more suitable for printing than ladder logic.

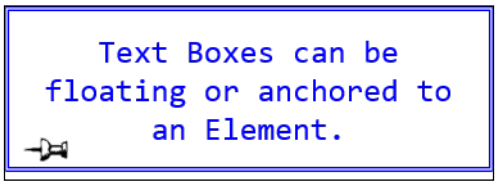
Adding a textbox to a function block routine

Textboxes can be added to a function block routine to provide a floating box of documentation. Adding documentation to a function block routine can improve the readability of the code and make it easier to maintain in the future. Just keep in mind that the textbox documentation will also need to be maintained and updated as the routine evolves over time, so keep the documentation clear and concise.

A textbox can be added to a function block routine by dragging the textbox icon from the language element toolbar into the target routine.



You can double-click on the textbox and enter your code notes and press *Ctrl + Enter* when finished. The textbox can be attached to a function block element by clicking on the pin symbol on the textbox, and then clicking on the function block element you want to attach it to. Once the textbox is linked with a function block, it will move with the function block when it is moved around the sheet.

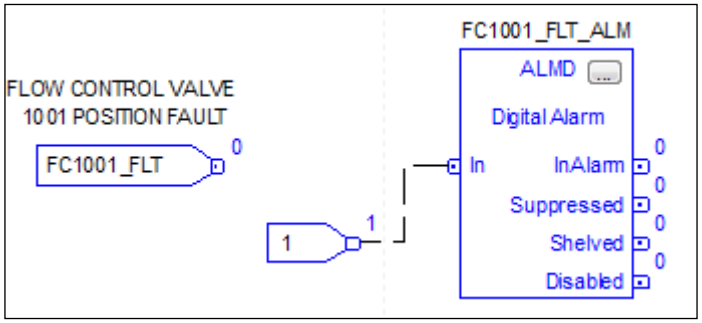


Hiding and showing function block pins

Function blocks often have some of their input or output pins hidden from view. You can view the available pins of a function block by clicking on the function block properties box or right-clicking on the function block and selecting **Properties**. The available input and output pins are listed under the **Parameters** tab. You can control the visibility by checking or unchecking the checkboxes under the **Vis** column.

Assigning a constant value to a function block

Rather than assigning a tag to an input reference of a function block, you can assign a hard coded value. You can specify a constant value by double-clicking on the input reference element and entering the desired value.



Summary

In this chapter, we explored the FBD origins in systems engineering and introduced the basic concepts of IEC FBD programming. We learned how to create FBDs by dragging and dropping elements into a sheet in a routine. We also learned how to wire input and output references to function block pins and identify digital and analog connections and monitor their values online.

In the next chapter, we will introduce another IEC language called **Structured Text Programming** and its use in the Logix family.

7
Writing Structured Text

In this chapter, we will explore the strengths and weaknesses of structured text programming by exploring the typical uses for this language and building several small sample applications. This chapter will cover the following structured text topics:

- Writing structured text routines
- Structured text operators
- Structured text expressions
- Structured text instructions
- Structured text constructs

An introduction to structured text programming

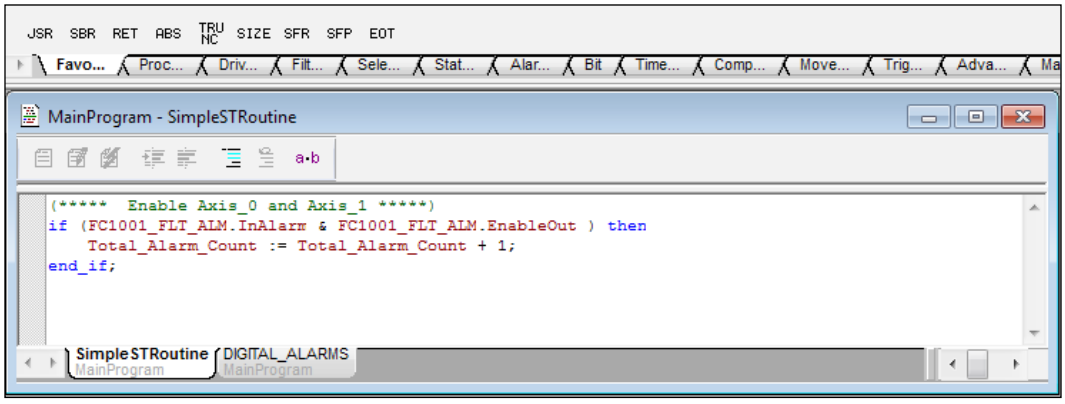
Structured text (**ST**) is another IEC 61131-3 language, which can be used in your Logix applications. As the name implies, structured text is a text-based programming language with a syntax that resembles Pascal (on which it was based) or **Visual Basic for Applications** (**VBA**). Like other IEC-based languages, it can share IEC common elements and reference tags and objects created in other languages with your Logix program.

Typical usage of structured text

Structured text is extremely robust and capable of matching the functionality of any other IEC-based language. However, you will find that structured text is used sparely in most automation projects. I have seen automation projects built entirely in the structured text code and they do work fine. But, the practice of developing entirely in structured text is frowned upon by most automation professionals. Often, you will see this from an engineer who is freshly out of school and is accustomed to modern programming text-based programming languages. Since its inception in 1968, ladder logic has been, and continues to be, the primary automation programming language. Selecting an IEC language for your routine is about selecting the right tool for the job. Ladder logic is a powerful language for simple sequential process control problems; it is easy to write and easy for other automation professionals to understand. Function block provides a high-level object-based language, which can create powerful routines with minimal effort that are easy to maintain. Structured text is a relatively low-level language that excels at complex algorithms, complex decisions (logical statements), and text manipulation.

The structured text editor

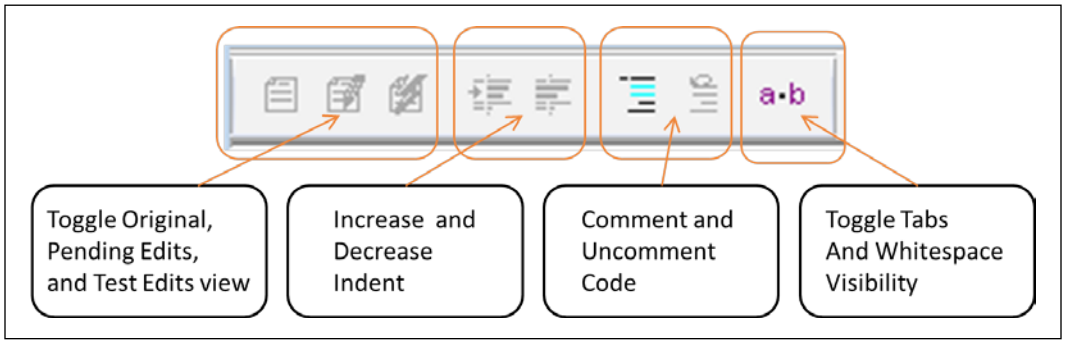
The structured text editor appears in the **Routine** window of Logix Designer and is the development environment for writing structured text code. The structured text editor window allows you to type in structured text code or drag and drop code elements from the structured text element toolbar.



The structured text editor window

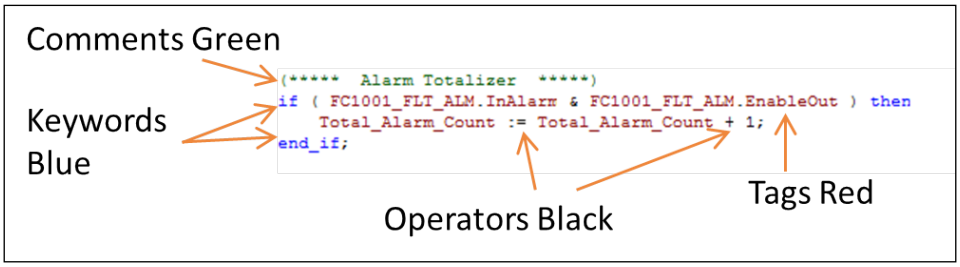
The structured text editor features a toolbar which allows you to perform the following tasks:

- Toggle the code view between original, pending edits, and test edits while making online changes to your structured text code
- Increase and decrease the indent of a selected piece of structured text code (it can also be accomplished using the hotkeys, *Tab* to increase and *Shift* + *Tab* to decrease the indent)
- Comment and uncomment out a selected piece of structured text code
- Show and hide the whitespace and tab characters



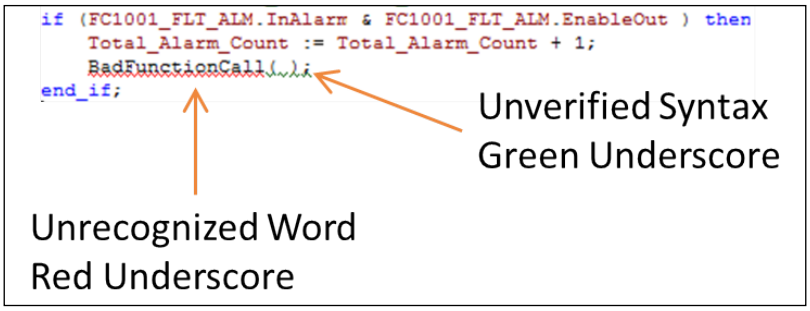
The structured text editor toolbar

The code area of the structured text editor provides context-sensitive code coloring to improve the readability of code. Color is used to signify that a word is recognized by the structured text editor and helps to ensure that the syntax is valid at a glance. The code colors can be adjusted by navigating to **Tools** | **Options…**.



Context-sensitive code coloring

The structured text editor also provides a syntax checker function as you edit. Any words that are unrecognized are indicated by a red wavy underscore line beneath the unrecognized word. Any syntax that is unrecognized or unverified is marked with a green wavy underscore line beneath the unverified word.



The structured text editor word and syntax checker functionality

The structured text editor also provides you with context-sensitive help for the functions in your routine. If you want to learn more about a specific function in your routine, select it and press *F1*. The **Help** documentation will open and automatically browse to the specific chapter of the function you have selected.

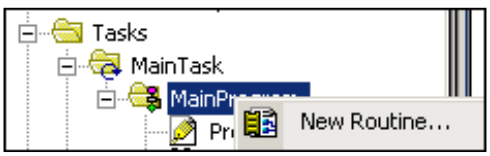
Writing structured routines

To get started with structure text, let's write a simple routine and break down the components.

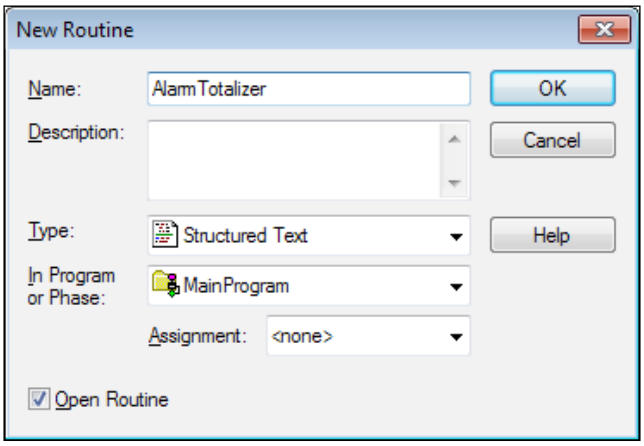
Simple routine

We have taken a brief look at the structured text editor, and now we will write a simple structured text routine to introduce some of the syntax and conventions. Our first structured text routine will provide an alarm count totalizer. The first version of this we create will have a few small errors, which we will correct as we learn more about structured text. Perform the following steps:

1. Open the **Controller Organizer** pane and navigate to **Tasks** | **Main Tasks** | **Main Program**. Right-click on **Main Program** and select **New Routine**, as shown in the following screenshot:



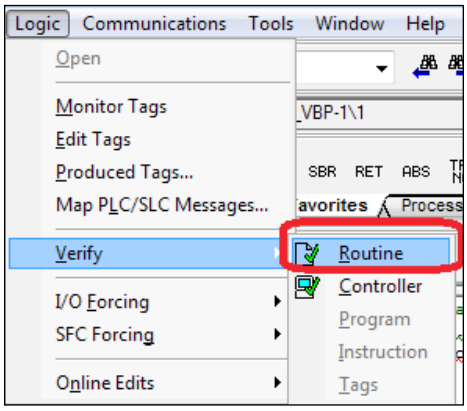
2. Configure a new structured text routine by setting the following values:
    - **Name**: `AlarmTotalizer`
    - **Description**: `Digital Alarms`
    - **Type**: **Structured Text**



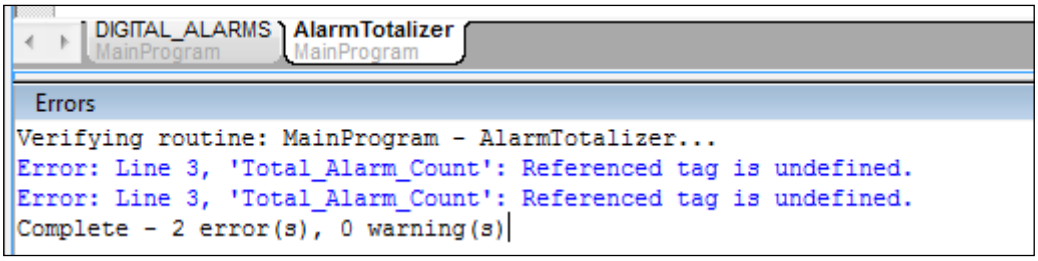
3. In the structured text editor, enter the following code:

```
(*****  Alarm Totalizer  *****)
if ( FC1001_FLT_ALM.InAlarm AND NOT FC1001_FLT_ALM.Disabled
  ) then
  Total_Alarm_Count := Total_Alarm_Count + 1;
end_if;
```

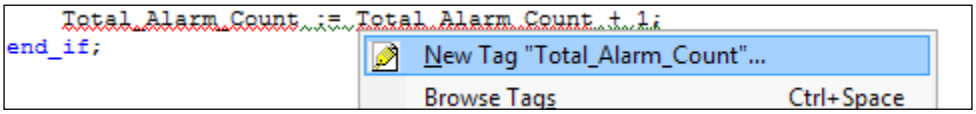
4. Next, we will ensure that we have not made any syntax errors in our code by verifying the routine. Navigate to **Logic** | **Verify** | **Routine**, as shown in this screenshot:



5. Verifying routine will determine if there are any errors or warnings in our code. An error is any problem in your routine that will prevent it from running on the controller. A warning is an issue that will still allow the routine to run on the controller. If you made any syntax errors, you will see them listed in the **Errors** panel at the bottom pane of the screen. If you entered the structured text correctly, you should see two errors, as shown in the following screenshot:

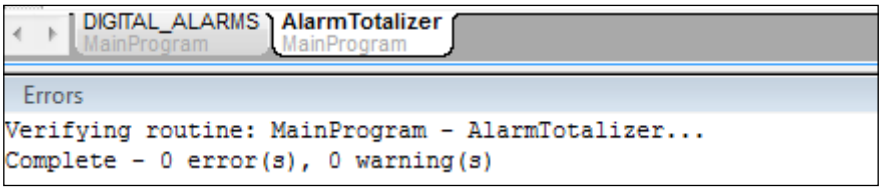


6. The errors identified by the **Verify** option for **Routine** are due to a tag we referenced in our structured text that has not been declared. You can see that the structured text editor also highlights this for us using the red wavy underscore. To resolve this error, you can simply right-click on the `Total_Alarm_Count` tag and select **New Tag "Total_Alarm_Count"...**:

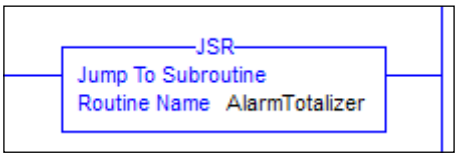


7. Use the **New Tag** form to create a new **Total_Alarm_Count** tag of the **DINT** type by setting the following values:
    - **Name**: `Total_Alarm_Count`
    - **Description**: `Total Alarm Count of Process`
    - **Data Type**: **DINT**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**

8. After adding the tag, we can run our **Verify** option for **Routine** again, and we will see that our routine is now error-free:



9. Finally, we cannot forget to add the **AlarmTotalizer** routine to **MainRoutine**; otherwise, it will never get executed. We can simply copy and paste one of the ladder rungs in **MainRoutine** and change the value to point at our routine, **AlarmTotalizer**.
10. Right-click on the left-hand side of the first ladder rung (where the **0** value is displayed) and select **Copy** (or press *Ctrl* + *C*).
11. Right-click below the first rung and select **Paste** (or press *Ctrl* + *V*).
12. Now, double-click on the **Routine Name** parameter of the JSR element and select our newly added **AlarmTotalizer** routine.



Now that we have created a simple structured text routine, let's use it to break down some of the components of structured text.

Our simple structured text routine begins with a `(***** Alarm Totalizer *****)` comment. Comments are ignored by the compiler and are used to provide documentation within your structured text.

The second line of our structured text routine is a construct, which is a conditional statement used to trigger other code statements. In our code, we use an `if` construct to determine whether our logical expression is true, and if so, we increase our total alarm count:

```
if ( FC1001_FLT_ALM.InAlarm AND NOT FC1001_FLT_ALM.Disabled ) then
```

Within the conditional statement is the AND (`&`) logical operator, which will be used to evaluate the condition—if the fault tag's condition is InAlarm and the fault tag is not disabled.

The third line increments out the total alarm count by one:

```
Total_Alarm_Count := Total_Alarm_Count + 1;
```

The assignment operator (`:=`) is used to assign the new value of the alarm count to the old value of the alarm count plus one.

Finally, we close out the `IF` construct we added in the second line with the `END_IF` statement:

```
END_IF;
```

The `END_IF` statement tells our program where to stop executing the code related to our original `IF` statement.

If you had to run and test our program, you would immediately see an issue with our alarm totalizer counter. Once the digital alarm is triggered, the `Total_Alarm_Count` value will continuously increase with each scan cycle. Certainly, this was not the intent of our alarm counter routine. Later in this chapter, we will investigate instructions and develop a solution to the bug in our routine.

Structured text syntax

Structured text is not case-sensitive, so uppercase and lowercase letters are considered the same character by the compiler. There are best practices for using uppercase and lowercase letters, which we will demonstrate in our routines. Also, the structured text compiler ignores whitespace, which is any space or tab character. This allows you to space out your code in order to make it easier to read.

Operators

Now that you have seen some structured text in action, let's take a look at the available operators in structured text.

Assignment operator

The assignment operator will change a value stored in a tag. An assignment operation is comprised of three parts: the tag, operator (`:=`), expression, and a semicolon at the end. The tag will maintain the assigned value until it is changed by another assignment value. Even after a power cycle, the value will be retained. In this respect, the non-retentive assignment operator is similar to an output latch in ladder logic.

The following code snippet provides some simple assignment operator examples:

```
// Assignment Operator Examples
tag := expression;

PumpSpeed := 100;
ValvePosition := OpenPosition;
FourValue := 2 + 2;
```

Non-retentive assignment operator

The non-retentive assignment operator will change a value stored in a tag. A non-retentive assignment operation is comprised of three parts: the tag, operator (`[:=]`), expression, and a semicolon at the end. The tag will maintain the assigned value until it is changed by another assignment value. Non-retentive assignment operations differ from an assignment operation in that its value will be reset after a power cycle of the controller or when the controller is switched back to the run mode. In this respect, the non-retentive assignment operator is similar to **Output Energize** in ladder logic.

The following code snippet provides some simple assignment operator examples:

```
// Non-retentive Assignment Operator Examples
tag [:=] expression;

PumpSpeed [:=] 100;
ValvePosition [:=] OpenPosition;
FourValue [:=] 2 + 2;
```

Retentive versus non-retentive assignment operators

When developing an industrial automation program, it is important to consider how the program will recover from a sudden loss of power or when the controller is switched to the run mode. After a power cycle or change to the run mode, you may want some values to reset to zero and other to maintain their last known value. For example, you may not want the speed and run condition of a **variable frequency drive** (**VFD**) to assume the last known value after a process comes back online. After a power failure or a process upset, you may want to manually start-up your VFD from your HMI. A veteran automation professional will always consider the values that should be retentive (or latched) and the values that should be non-retentive (or energized) in their program.

Buffering structured text I/O module values

Just like ladder logic, the structured text I/O module values should be buffered at the beginning of a routine or prior to executing a routine in order to prevent the values from changing mid-execution and putting the process into a state you could not have predicted.

The following code is an example of the ladder logic buffering routine we wrote earlier in structured text and using the non-retentive assignment operator:

```
(*
I/O Buffering in Structured Text
Input Buffering
*)
StartPump [:=]  Local:2:I.Data[0].0;
HighPressure [:=] Local:2:I.Data[0].1;
PumpStartManualOverride [:=] Local:2:I.Data[0].2;
(*
I/O Buffering in Structured Text
Output Buffering
*)
Local:3:O.Data[0].0 [:=] RunPump;
```

Relational operators

Relational operators compare two values and provide a `BOOL` (Boolean—true or false) value in return. The following table lists the available relational operators in structured text:

Relational type	Operator
Equal	=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Not equal	<>

Relational operators are typically used in a construct like the `IF` statement we used in our simple example:

```
(* Relational Operator Example *)
if ( TankLevel >= 100 AND MotorRunStatus = 0) then
  StartMotor [:=] 1;
end_if;
```

Logical operators

We looked at the logical operators for the ladder logic and function block programming languages in this book already. Structured text has the same capabilities for the logical statements as the other IEC languages:

Logical type	Operator
And	`&`, `AND`
Or	`OR`
Not	`NOT`
Exclusive or	`XOR`

Logical operators are typically used in a construct like the `IF` statement we used in our simple example:

```
(* Logical OR Operator Example *)
if ( Tank1Level >= 100 OR Tank2Level >= 100) then
  StartMotor [:=] 1;
end_if;
(* Logical AND Operator Example *)
if ( TankLevel >= 100 AND MotorRunStatus = 0) then
  StartMotor [:=] 1;
end_if;
(* Logical NOT Operator Example *)
if ( NOT MotorRunPermissive) then
  StartMotor [:=] 0;
end_if;
```

Arithmetic operators

Arithmetic operators are used in calculations, which is the real strength of structured text when compared to other IEC languages. The following table lists the available arithmetic operators:

Calculation type	Operator
Add	`+`
Subtract/negate	`-`
Multiple	`*`
Exponent	`**`
Divide	`/`
Modulus (division remainder)	`MOD`

Logical operators are typically used in an expression, which we will detail in the next section.

Expressions

Expressions are a combination of values, constants, tags, operators, and functions that are interpreted according to the particular rules of precedence and produce a value. Expressions in Logix will either evaluate to a number (numerical expression) or to a true or false state (BOOL expression). Parenthesis "`( )`" can be used to control the order of operation, just like they would in any mathematical equation. The following table lists the order of operation in Logix:

Operation	Order
Parenthesis "`( )`"	1
Instructions "`function(…)`"	2
`**`	3
negate (`-`)	4
`NOT`	5
`*`, `/`, `MOD`	6
`+` and `-`	7
`<`, `<=`, `>`, and `>=`	8
`=` and `<>`	9
`&` and `AND`	10
`XOR`	11
`OR`	12

Here is an example that uses these operators:

```
(* Expression Example *)
if ( (Tank1Level + Tank2Level) * 10 >= 1000) then
  StopMotor [:=] 1;
end_if;
(* Numeric Algorithm Example *)
TankVolume = 3.14*(TankRadius**)*TankLength
```

Instructions

Instructions are the built-in functions, which will make up the building blocks of most structured text routines. Logix Designer provides a rich set of instructions to utilize within a structured text. All the available structured text instructions are listed in the element groups above the structured text editor and can be dragged into your routine. Alternatively, you can right-click in the structured text editor and select **Add ST Element…** (or press *Alt* + *Ins*), and then select **Add Structured Text Element** from the pop-up window that appears.

The instructions in structured text are equivalent to the ladder element instructions in the ladder logic and function block element instructions in the function block diagrams. However, each language executes the IEC instructions slightly differently. The execution of the function block instructions is triggered using the **EnableIn** pin. Structured text instructions execute as if **EnableIn** is always energized. Ladder logic element instructions use energized rungs to trigger the execution of an instruction, while structured text will execute each time they are scanned, unless you put a conditional construct around it (such as an `IF` statement).

Arithmetic instructions

There are many sets of instructions available in structured text in the element groups. One of the most important instruction sets for structured text are the arithmetic instructions. The arithmetic instructions will aid with any complex calculations you are embedding into your structured text routine.

Calculation type	Instruction syntax
Absolute value	`ABS( numeric expression)`
Arc Cosine	`ACOS( numeric expression)`
Arc Sine	`ASIN( numeric expression)`
Arc Tangent	`ATAN( numeric expression)`
Cosine	`COS( numeric expression)`
Radian to Degrees	`DEG( numeric expression)`
Natural Log	`LN( numeric expression)`
Log base 10	`LOG( numeric expression)`
Degrees to radians	`RAD( numeric expression)`
Sine	`SIN( numeric expression)`
Square root	`SQRT( numeric expression)`
Tangent	`TAN( numeric expression)`
Truncate	`TRUNC( numeric expression)`

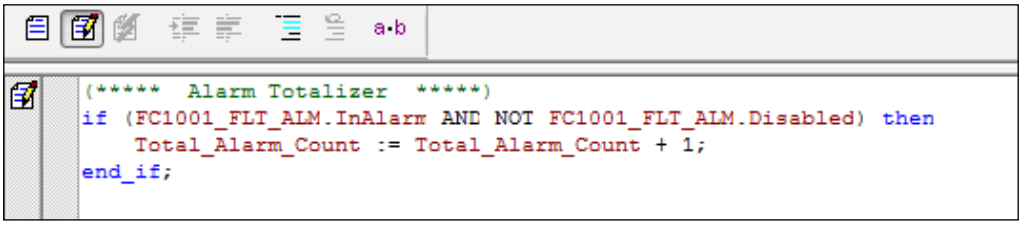
Here is an example that uses these instructions:

```
(* Arithmetic Instructions Example *)
DriveSpeed [:=] ABS((FrequencyValue + Offset)/2);
PolygonArea [:=] 0.5*SideCount*SIN(360/SideCount)*Length**;
```

ORSI instruction

In the simple structured text sample we created in the previous exercise, we encountered an issue where our alarm's total count was continuing to increase with each scan cycle while the the digital alarm was active. In order to adjust our code to count only new alarms as we had intended, we will need to only count a new alarm on a change of alarm state from off to on. This is known as detecting the rising edge of the signal. We can accomplish this in structured text using the **One Shot Rising with Input** (**OSRI**) instruction. In the following exercise, we will add the ORSI instruction to the routine we started earlier in the chapter to resolve our alarm counting issue by performing these steps:

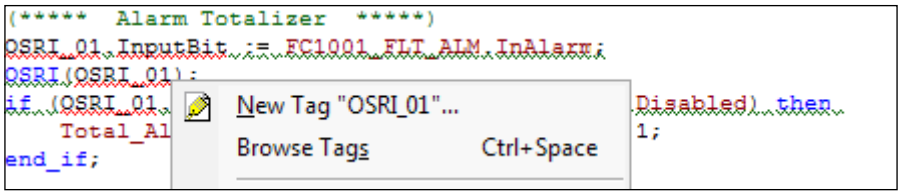
1. Open up the structured text routine, we created earlier, called **Alarm Totalizer**, as shown in the following screenshot:



2. Add the following structured text code after the comment and change the `IF` construct value:

```
(***** Alarm Totalizer *****)
OSRI_01.InputBit := FC1001_FLT_ALM.InAlarm;
OSRI(OSRI_01);
if (OSRI_01.OutputBit AND NOT FC1001_FLT_ALM.Disabled) then
  Total_Alarm_Count := Total_Alarm_Count + 1;
end_if;
```

3. Now, highlight the OSRI instruction in the structured text editor and press *F1* to take a closer look at the way it is used within structured text. In the **Help** documentation, we can see that the OSRI instruction has passed an `FBD_ONESHOT` data type. So, we will need to add our missing data type value, `ORSI_01`, to our program tags by right-clicking on the missing tag and selecting **New Tag "OSRIP_01"...**.



4. We will need to add the `OSRI_01` tag as an `FBD_ONESHOT` data type in order to support the `.InputBit` and `.OutputBit` properties and pass them into the OSRI instruction.
5. Use the **New Tag** form to create a new `OSRI_01 FBD_ONESHOT` tag by setting the following values:
    - **Name**: `OSRI_01`
    - **Description**: `One Shot for Catching Rising Edge of Digital Alarm`
    - **Type**: **Base**
    - **Data Type**: **FBD_ONESHOT**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
6. Now if we compile, download, and test our routine, we will see that the digital alarm is now incrementing correctly only once per new alarm.

Constructs

Constructs control the flow of our structured text routine and allow us to create decision statements, state machines, and loops.

The IF THEN construct

We are already familiar with the `IF THEN` construct from our simple structured text exercise in this chapter. An `IF` construct will only execute the structured text between the `IF` and `END_IF` loops when its expression evaluates as True (or 1).

The `IF` statements can be nested using the `ELSEIF` statement, and the `ELSE` statement can be added to execute when all other statements do not.

```
(* IF THEN ELSEIF ELSE Example *)
if (TankLevel >= 50) then
  Pump1Permissive [:=] 1;
elseif (TankLevel >= 100) then
```

```
  Pump1Permissive [:=] 1;
Pump2Permissive [:=] 1;
else
  Pump1Permissive [:=] 0;
Pump2Permissive [:=] 0;
end_if
```

The CASE OF construct

A `CASE` statement can be used to execute statements based on a numeric value:

```
(* CASE Example *)
case sequence_number of
  1: StartPump [:=] 1;
    OpenValve [:=] 1;
  2: StartBlower [:=] 1;
  3,4: StartMixer [:=] 1;
  4..10: StartAuger [:=] 1;
else
  StartPump [:=] 0;
end_case;
```

The FOR DO construct

The `FOR DO` construct will loop a specific number of times before continuing to execute the routine. Loops are quite useful when working with arrays of values:

```
FOR count := initial_value
TO final_value
{ BY increment If you don't specify an increment, the loop
  increments by 1. - Optional }
DO
```

Here's an example:

```
For LoopCount := 1 TO 10 By 1
  ALARMS[LoopCount] = 0;
DO
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Summary

In this chapter, we introduced structured text and the best uses for it within an automation program. We created a simple structured text routine and learned about the powerful syntax of structured text code.

In the next chapter, we will introduce the final Logix IEC language called sequential function chart programming.

8
Building Sequential Function Charts

In this chapter, we will implement a sequential function chart routine and break down the steps, actions, transitions, and branches that are used to construct it. We will also work with the online editing capabilities of sequential function chart routines.

Introducing sequential function charts

Sequential Function Charts (**SFC**) is another IEC 61131-3 language, which allows you to visually program using a flow chart construct. The IEC SFC language is based on the GRAFCET language, which was the original industrial automation flow chart programming language. In some regions and within some companies, you will find that the terms SFC and GRAFCET are used interchangeably. Like other IEC-based languages, it can share IEC common elements and reference tags and objects created in other languages with your Logix program. SFC is a powerful high-level language similar to function block. Often, you can create the equivalent functionality of 40 ladder logic rungs in a few SFC steps. An SFC routine (again, like function block) will typically have more computational overhead than the same routine developed using low-level languages such as ladder logic or structured text. SFC routines are very easy to debug and maintain because of their compact visual design. SFC has its strengths and weakness like the other IEC languages. An automation professional will select the right language to meet the particular challenges of the process they are faced with. It is not uncommon for an automation professional to realize that they have selected the wrong language for a particular task and rewrite the routine in a different IEC language.

Typical usage of SFCs

As the name would imply, SFC is well-suited for sequential step processes. You will also find that SFC is popular within batch processes as it closely aligns with the step-by-step requirements of batching. It is particularly useful when paired with other IEC languages. Within SFC routines, you will use structured text for any logical expressions, assignments, or other constructs. By using the structured text function, JSR (or similar functions), you can reference ladder logic, function block, structured text, or even other SFC routines. SFC can be used as a high-level program flow controller (business logic) and the detailed step logic can be created in separate routines. Structuring a program in this way will make it easy to follow, troubleshoot, and modify in the future.

The SFC editor

The SFC editor appears in the routine window of Logix Designer and is the development environment for writing the SFC routines. Within an SFC routine, the SFC elements are created in an area called a sheet, and the sizes of the sheets directly correspond to the standard metric or English printer page sizes (similar to the function block diagram sheets). This allows the sheets to be easily printed, presented, and even signed like a typical engineering drawing. It is helpful to think of each sheet as a drawing for a single device. Unlike the function block diagram sheets, you cannot divide your SFC into multiple named sheets. However, the SFC routine is not limited in size like a function block routine. You can expand the size of the SFC routine to be up to 175 x 175 cross reference grid blocks. An SFC routine can span multiple sheets horizontally and vertically. The specified sheet size makes it easier to organize the way the SFC routine appears when printed. You can see the sheet size divisions are solid grey lines on your SFC routine. The sequential function chart editor window allows you to add the SFC elements from the SFC element group, drag them around the sheet, and connect the flow of the SFC elements using wires.

Defining the SFC steps

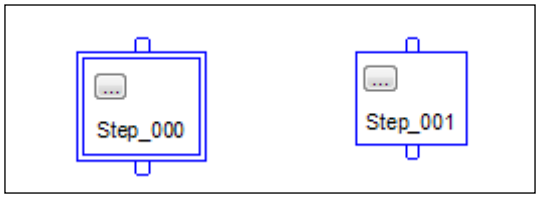
Steps are the main building blocks for the SFC routines. A step is associated with the logic to be executed at a specific point in your process. A step is represented by a rectangle shape in the SFC routine with a pin at the top and bottom. Steps can be referenced just like the function block elements and contain a number of properties as follows:

- `.T`: This property is the timing of how long the step has been active
- `.PRE`: This property is the preset amount of time to run a step for

- `.DN`: This property is the Boolean flag that triggers high, once the timer reaches the preset amount of time
- `.X`: This is a bit that is ON the entire time that the step is executing.

There are also a number of alarm properties that can be configured and triggered if a step runs too long or not long enough. The complete list of step properties can be found in the **Help** documentation by selecting a step element and pressing *F1* on the keyboard.

All SFC routines must have an initial step defined in order to run. The initial step is indicated with a double line around its rectangle shape. You can specify which step you would like to make the initial step by right-clicking on it and checking **Initial Step**. The following screenshot shows two step elements; on the left-hand side is the initial step and on the right-hand side is a normal step:

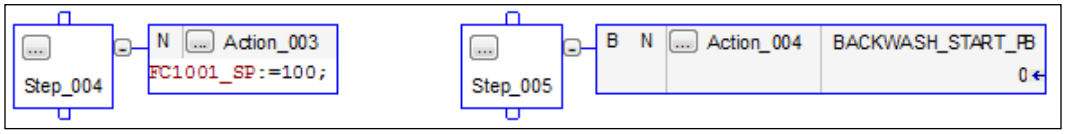


Defining the SFC actions

Actions are associated with a step and contain the structure text code used to perform functions such as starting a pump. You can associate multiple actions with a single step. There are two types of actions:

- **Non-Boolean**: These actions allow you to execute the structured text code or reference other routines using the JSR function.
- **Boolean**: These actions set a `Bool` tag value to true when it is on. This action will require other logic to monitor this tag value in order to execute.

The following diagram shows a non-Boolean action on the left-hand side and a Boolean action on the right:



Each action element and its properties can be referenced and monitored. The action elements have the following properties:

- `.A`: This property means that the Boolean property is **ON** when the action is active
- `.T`: This property is the timing of how long the action has been active
- `.PRE`: This property is the preset amount of time to run an action for
- `.Count`: This property is the number of times the action has become active

There are also a number of other properties that can be monitored on actions. The complete list of action properties can be found in the **Help** documentation by selecting an action element and pressing *F1* on the keyboard.

Actions also use qualifiers to determine when it should start and stop. By default, all actions are set to non-stored qualifiers, which means it will execute when the step is associated with its execution. The following qualifiers are available for actions:

Symbol	Name	Description
`N`	Non-stored	This qualifier starts when the step is activated and stops when the step it is deactivated.
`P1`	Pulse (rising edge)	This qualifier starts when the step is activated and executes only once.
`L`	Time limited	This qualifier starts when the step is activated and stops when the timer runs out or when the step is deactivated.
`S`	Stored	This qualifier starts when the step is activated and stays active until a reset action property is triggered.
`SL`	Stored and time limited	This qualifier starts when the step is activated and stays active until a reset action property is triggered or stops when the timer runs out. It does not deactivate when the step is deactivated.
`D`	Time delayed	This qualifier starts the action for a predefined amount of time after the step is active and while the step is still active. It stops when the step is deactivated.
`DS`	Delayed and stored	This qualifier starts the action for a predefined amount of time after the step is active and while the step is still active. It stays active until an action reset property is triggered.

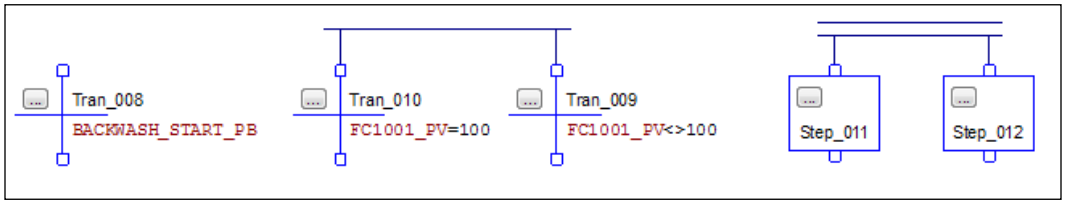
Symbol	Name	Description
SD	Stored and time delayed	This qualifier starts when a specific amount of time has passed after the step is activated, even if the step is later deactivated the action will still execute. It stays active until an action reset property is triggered.
P	Pulse	This qualifier executes the action once when the step is activated and again when the step is deactivated.
P0	Pulse (falling edge)	This qualifier starts when the step is deactivated and executes only once.
R	Reset	This qualifier resets (turns off) an action and can be used to reset some of the other action qualifiers.

Defining the SFC transitions and branches

Transitions are used to bind steps to other steps and specify a structured text logical condition (which could also include a jump to subroutine function) that must be true in order to proceed. A branch is a wire that connects to multiple transitions or steps. Branches and transitions work together to control the flow of an SFC sequence. There are three types of transitions/branches:

- **Sequence transition**: This transition provides a logical expression between each step.
- **Selection branch transition**: This transition provides a logical expression that will direct the step flow to one of many steps (similar to an OR expression). Additional steps can be added to a selection branch by right-clicking on it and selecting **Extend Branch**.
- **Simultaneous branch step**: This transition executes two or more steps at the same time (similar to an AND expression). Additional steps can be added to a simultaneous branch by right-clicking on it and selecting **Extend Branch**.

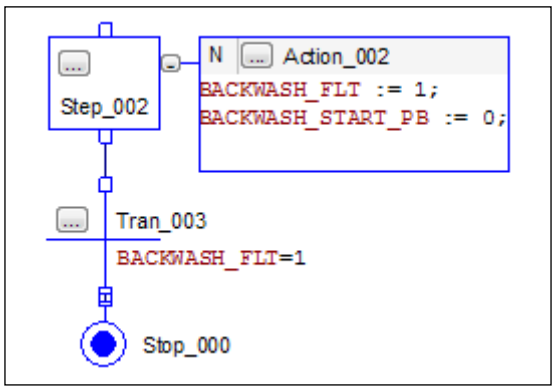
The following diagram illustrates the visual style of each transition and branch type:



Non-Boolean and Boolean actions in SFC

Defining the SFC stop element

The SFC stop element will stop the execution of an entire SFC routine or of a particular branch of an SFC routine and wait for a restart to be triggered. The following diagram demonstrates the use of a stop element in an SFC routine:

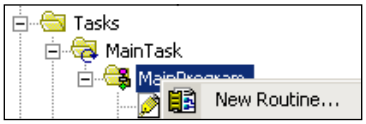


Stop element attached to a transition

A backwash SFC routine

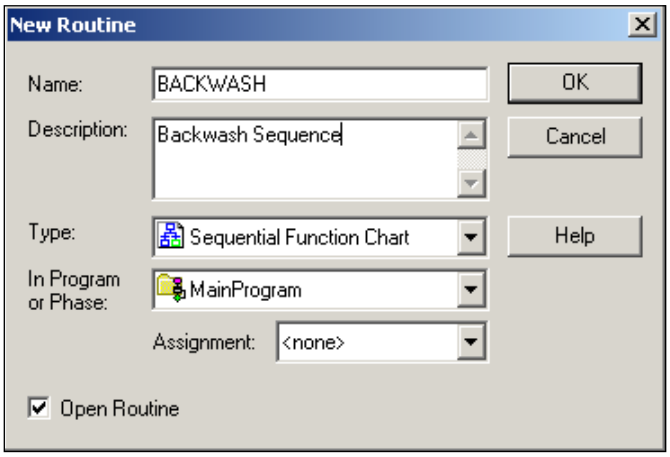
In this section, we will demonstrate the usage of a sequential function chart by building a backwash process using a step-by-step guide. The backwash process will be run to flush out the filters used in our process once they become too dirty. Follow these steps:

1. First, we will need to declare our new routine. Right-click on the **MainProgram** scope in the **Controller Organizer** pane and select **New Routine**.

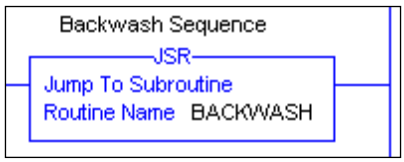


2. In the **New Routine** form that appears, enter or select the following values:
    - **Name**: `BACKWASH`
    - **Description**: `Backwash Sequence`
    - **Type**: **Sequential Function Chart**

3. Then, click on **OK**.

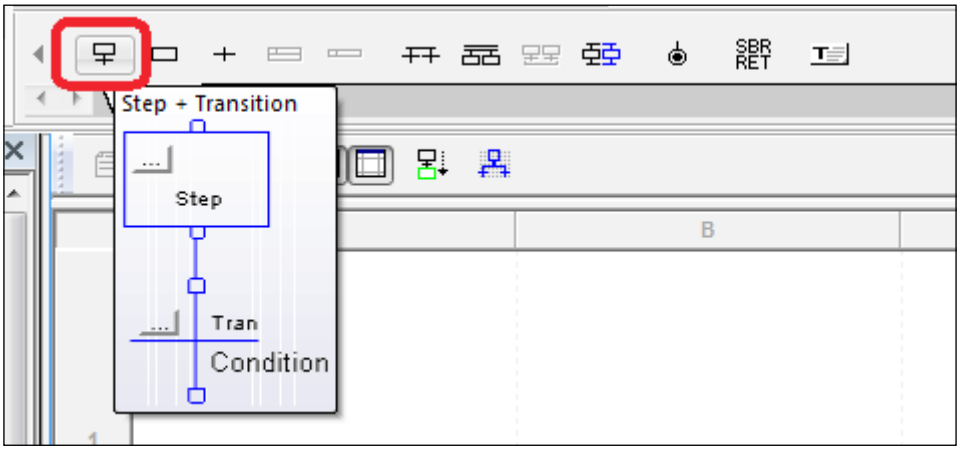


4. In order for our newly created routine to be executed with each scan of the controller, we will need to add a reference to it in the **MainRoutine** program, which is executed with each scan of **MainTask**.
5. Double-click on our **MainRoutine** program to display the **MainRoutine** ladder logic.
6. Now, we can simply copy and paste one of the **JSR** ladder rungs that we created earlier and change the value to point at our **BACKWASH** SFC routine.
7. Right-click on the left-hand side of the last ladder rung (where the rung number is displayed) and select **Copy** (or press *Ctrl* + *C*). Right-click below the last rung and select **Paste** (or press *Ctrl* + *V*).
8. Now, double-click on the **Routine Name** parameter of the **JSR** element and select our newly added **BACKWASH** routine.

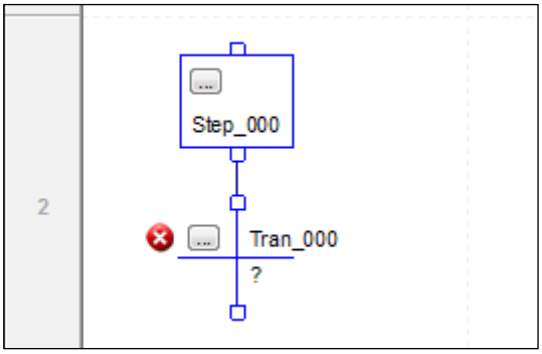


9. Now, we will return to our **BACKWASH** SFC by double-clicking on it in the **Controller Organizer** pane.

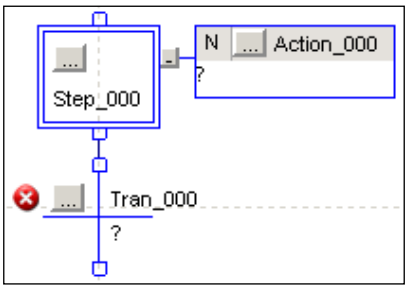
10. As we have learned, the base unit of any SFC routine is a step. Certainly, we are planning to have more than one step in our routine, so it will be prudent to add a transition as well. We can add our first SFC step and transition at the same time by clicking on the **Step + Transition** icon in the SFC element group.



11. You will see **Step_000** and **Tran_000** appear in the SFC editor window.

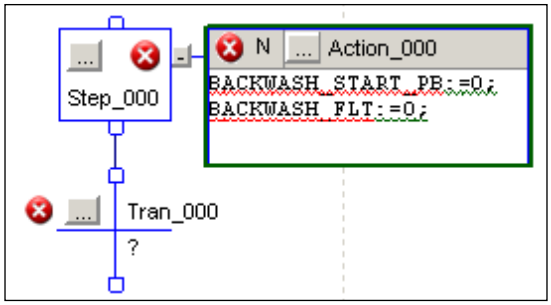


12. Add an action to **Step_000** we created. Right-click on **Step_000** and select **Add Action** (or alternatively, you can select the **Action** element from the SFC element group toolbar above our routine).



13. Now, we will add the initialization values for our SFC routine using the structured text syntax. Double-click on the box with the **?** symbol at the bottom of **Action_000** and enter the following structured text:

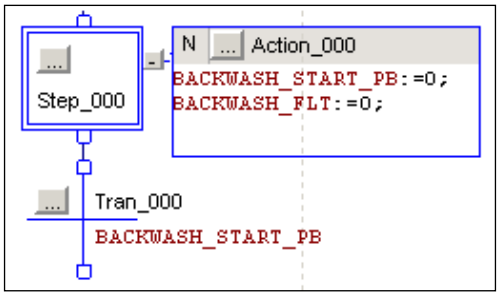
```
BACKWASH_START_PB:=0;
BACKWASH_FLT:=0;
```



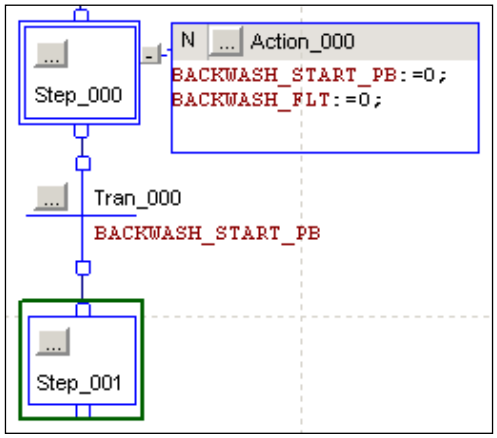
14. An icon with a red **X** symbol will appear on the side of **Action_000** to indicate that there is an error with the structured text we have added. The error is due to the `BACKWASH_START` and `BACKWASH_FLT` tags not yet being declared in our routine. We can easily add these tags by right-clicking on the `BACKWASH_START_PB` tag in the `Action_000` box and selecting the **New Tag "BACKWASH_START_PB"** option.

15. The **New Tag** form will appear, which will allow us to create our new tag as follows:
    - **Name**: `BACKWASH_START_PB`
    - **Description**: `START BACKWASH PUSH BUTTON`
    - **Type**: **BOOL**
    - **Scope**: **MainProgram**
16. Repeat the same process for the second new tag by right-clicking on the `BACKWASH_FLT` tag, and on the **New Tag** form, enter the following properties:
    - **Name**: `BACKWASH_FLT`
    - **Description**: `BACKWASH SEQUENCE FAULT`
    - **Type**: **BOOL**
    - **Scope**: **MainProgram**
17. Next, we will add the transition conditional value, which will start our backwash sequence. Double-click on the **Tran_000** question mark and enter the following structured text logical statement (which is equivalent to `BACKWASH_START_PB=1`):

    ```
    BACKWASH_START_PB
    ```



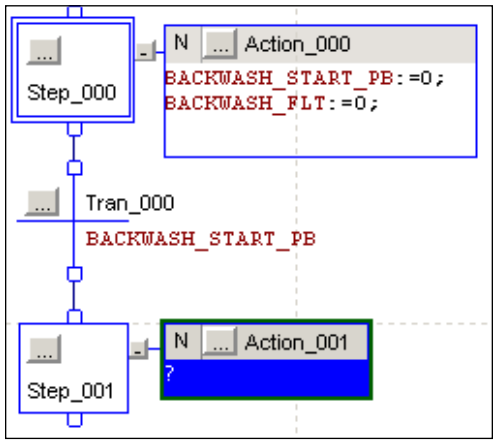
18. The next piece of development work will be to add our backwash sequence steps. Select the **Tran_000** transition box, and then click on the **Step** element icon in the SFC element group above our SFC routine. The **Step_001** step box will be added and automatically connected to **Tran_000** (because we selected the transition before adding our new step).



19. Next, we can add an action to **Step_001**, which will represent our backwash process.

At this stage, we can easily add a structured text JSR function to call a separate ladder logic, function block, structured text, or even another SFC routine. By combining SFC and the JSR function, we could control the flow of our program from SFC. This technique is very helpful when developing and debugging complex batching or sequencing programs.

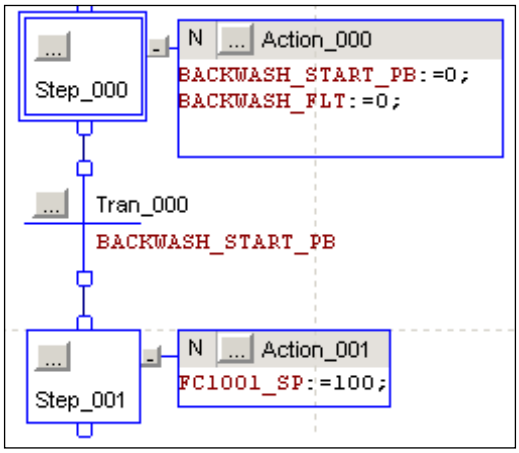
20. Right-click on **Step_001** and select **Add Action**.



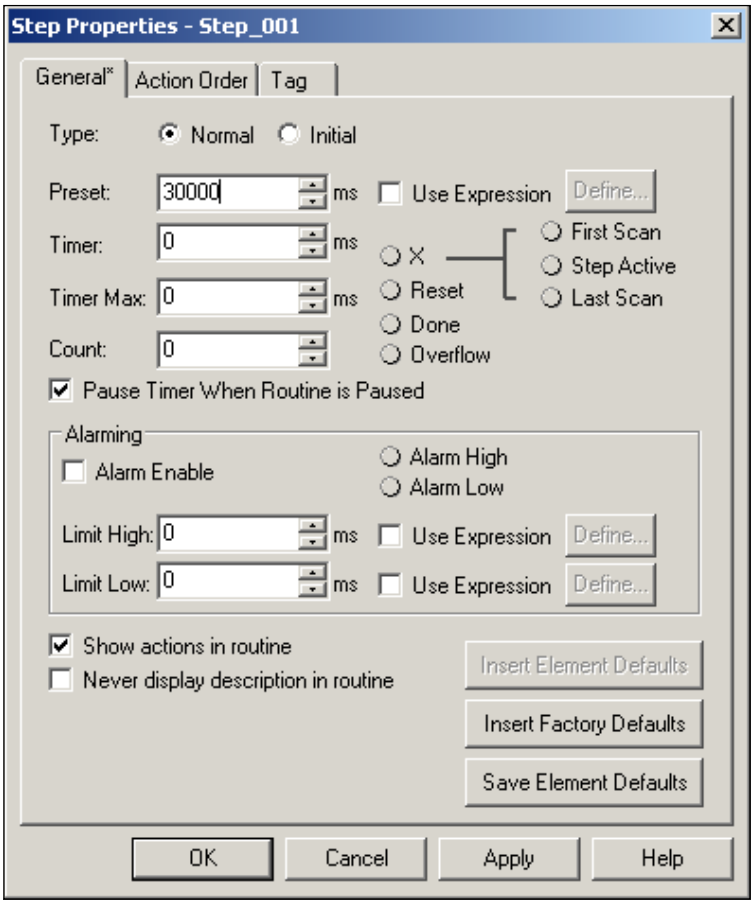
21. Now, we will add our action element's structured text code by double-clicking on the **?** symbol in **Action_001** and entering the following structured text code:

```
FC1001_SP:=100;
```

22. We will also need to add the new `FC1001_SP` tag by right-clicking on it and selecting the **New Tag** option.
23. Configure the new tag with the following properties:
    - **Name**: `FC1001_SP`
    - **Description**: `FLOW CONTROLLER 1001 SETPOINT`
    - **Type**: **DINT**
    - **Scope**: **MainProgram**

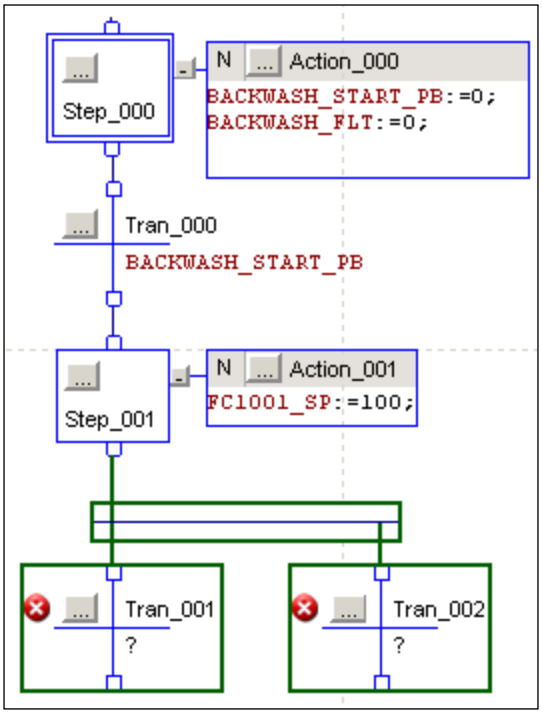


24. Let's add a delay to our **Step_001** box in order to give our backwash time to complete. Right-click on **Step_001** and select the **Step Properties** menu option. In the **Step Properties** form, set the **Preset** field to **30000ms** (30 seconds) and click on **OK**.



25. Next, we will add a selection branch diverge in order to reset our sequence or trigger a fault if there is a problem. Select **Step_001**, and then click on the **selection branch** diverge element icon ( ) just above our sequence chart.

A selection branch diverge executes one sequence or another depending on a logical condition (OR), while simultaneous branch diverge will execute two sequences in parallel (AND).



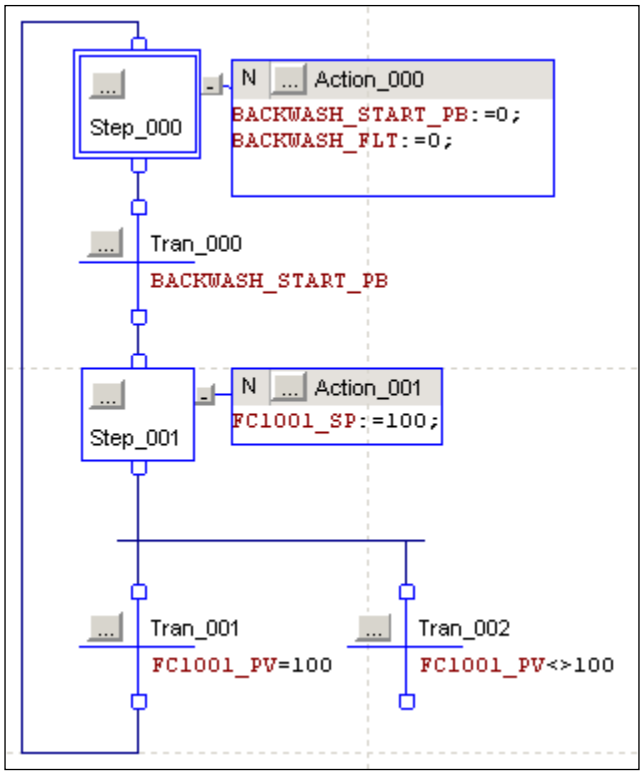
26. Our sequence will automatically reset and await another backwash if the flow controller valve position (`FC1001_PV`) has been 100 percent opened. Select **Tran_001** and click on the **?** icon to set the logical statement, which will execute this selection branch. Type the following structured text logical statement:

    ```
    FC1001_PV=100
    ```

    We will also need to add the `FC1001_PV` tag to our program by right-clicking on it and selecting **New Tag**.

27. In the **New Tag** window, enter the following field values:
    - **Name**: `FC1001_PV`
    - **Description**: `FLOW CONTROLLER 1001 VALVE POSITION VALUE`
    - **Type**: **DINT**
    - **Scope**: **MainProgram**

28. We will want our sequence to reset after it has completed the backwash, so we will connect a wire from our transition, **Tran_001**, to the top SFC element in our sequence, **Step_000**. Click on the connector box under **Trans_001** and drag it to the connector box on top of **Step_000**.



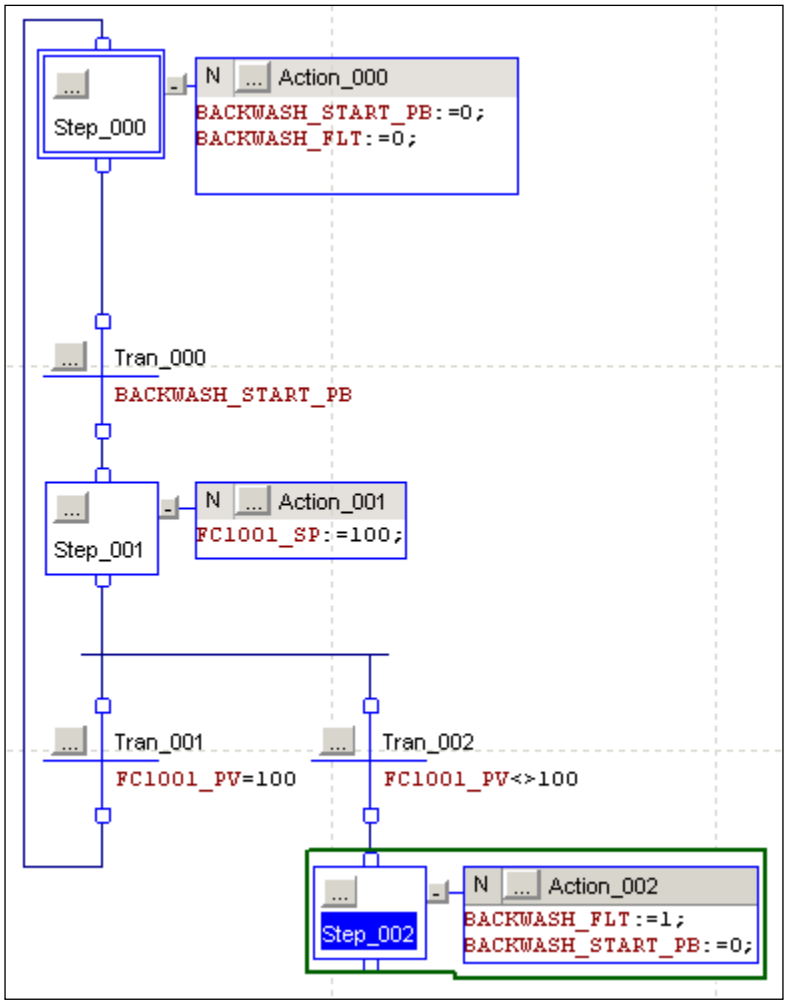
29. If our valve fails to open, we will want to raise a fault before resetting our sequence. Select **Tran_002**, click on the question mark, and enter the following structured text logical statement:

```
FC1001_PV<>100
```

30. In order to raise a fault, we will need to add a step. Select **Tran_002** and click on the **Step** element icon.
31. Add an action to our newly created step, **Step_002**, by right-clicking on it and selecting **Add Action**.

32. Double click on the **?** icon of our newly added action and add the following structured text code:

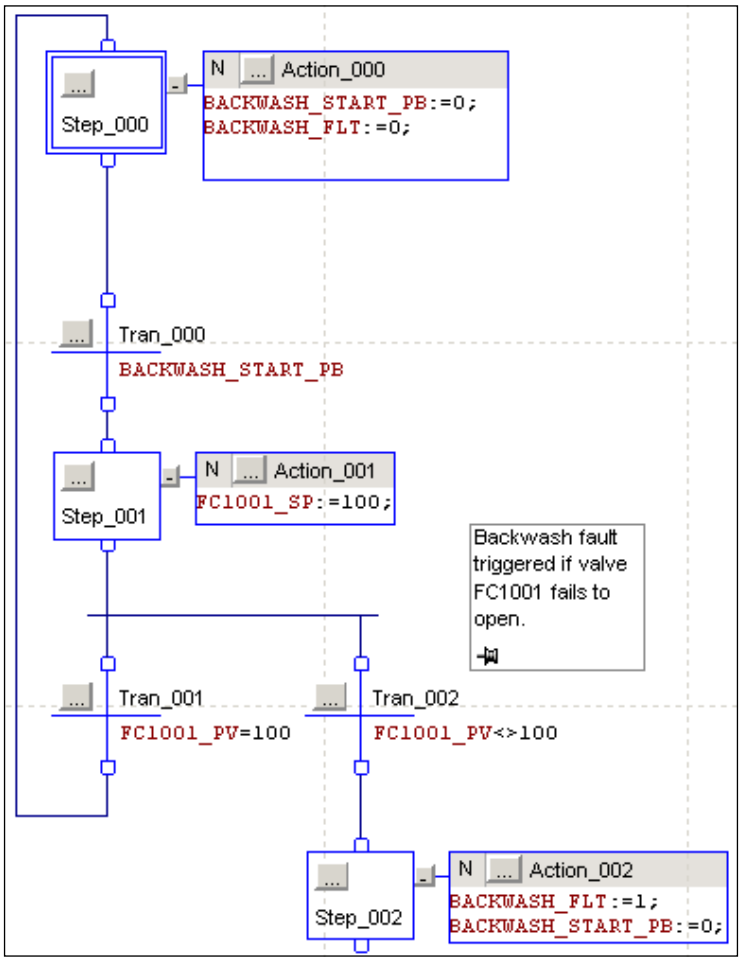
```
BACKWASH_FLT:=1;
BACKWASH_START_PB:=0;
```



The fault bit will prevent the backwash sequence from running again until the problem is investigated and the fault bit is reset.

33. Finally, in order to make our sequence easy to understand, let's add a textbox comment. Click on the **Text Box** element in the SFC element group toolbar to add it to the sequence diagram and drag it to the right of the sequence. Enter the following comment:

```
Backwash fault triggered if valve FC1001 fails to open.O
```



34. The complete SFC routine should be verified to ensure that there are no errors or warnings. From the drop-down menu, navigate to **Logic** | **Verify** | **Routine**.

In the **Errors** panel, you should see the following output:

```
Verifying Routine: MainProgram - BACKWASH…
Complete - 0 error(s), 0 warning(s)
```

If you do encounter errors, double-click on the error message to be taken to the exact location of the problem.

In the last exercise, we left the default names for our steps, actions, and transitions. It is easy to rename the SFC elements to make the sequence easier to read and maintain. You can double-click on the existing name of a step, action, or transition and type a new name to rename it.

Summary

In this chapter, we explored sequential function charts and their typical usages within an automation project. We looked at the few core elements that make up an SFC and created a simple backwash process routine.

In the next chapter, we will identify ways to organize and control the scan frequency of a routine using tasks and programs.

9

Using Tasks and Programs for Project Organization

In this chapter, we will see how to structure a Logix project using the basic organization units (which we have discussed throughout this book)—tasks, programs, and routines. We will also look at the way in which task scheduling and prioritization can be used to balance the processing time of a controller.

This chapter will cover the following topics in detail:

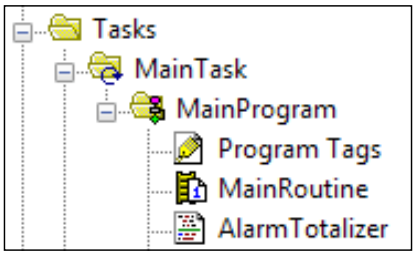
- Logix organizational units
- Controller task types
- Creating tasks
- Inhibiting programs and tasks
- Tuning a Logix controller

Introducing project organization in Logix

As we have already seen, a Logix project is organized into tasks, programs, and routines. When a new project is created, Logix automatically adds one of each basic organizational unit to the project:

- Task
- Program
- Routine

We have seen these units many times in our controller organizer:

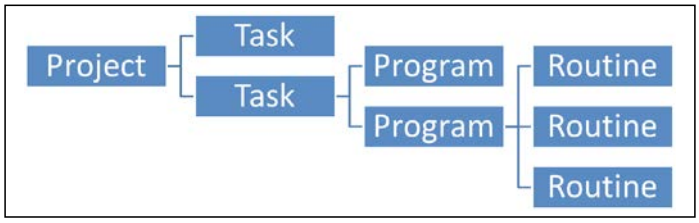


Tasks, programs, and routines in the controller organizer

These units also allow us to select how often and at what priority programs and routines are run. Project organization also allows us to control and optimize the way our projects run in order to reduce the processing load on a controller. In the next section, we will take a look at each organizational unit in more detail.

Organizational units in Logix

As stated in the previous section, the organizational units in Logix are tasks, programs, and routines. The following diagram illustrates the one-to-many relationships within a Logix project organization:



One-to-many relationships between projects, tasks, programs, and routines

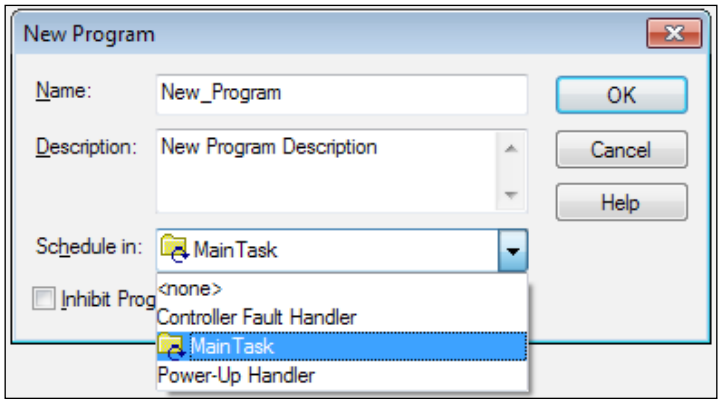
As the preceding diagram illustrates, Logix Designer is only capable of opening a single project at a time. A project can contain multiple tasks. A task can contain multiple programs, and programs can contain multiple routines.

Controller tasks

A task is the most foundational and powerful organizational unit in the Logix platform. When we create a new project in Logix, we are provided with a single task that runs all of our logic. Also, up to this point, we have only used this default single task to develop our application. It is possible to add multiple tasks to your project that can schedule and prioritize programs. However, it should be noted that there is a small controller performance hit, which is switching between tasks (about 1 ms). That said, it is wise to limit the number of tasks in your program to less than a handful. We will talk about different task types and a typical project task configuration in the next section.

Controller programs

Controller programs are associated with tasks or can also be assigned to controller events (such as a controller startup or fault). Programs are a collection of related routines and tags executed by a task. Multiple programs can be scheduled to run within a task and their order in which they execute can be adjusted. Tags can be declared within a program and their scope is limited to a single program. This allows you to use the same tag names across multiple programs. Programs can also be inhibited, moved between tasks, and monitored for performance (execution time). The following screenshot shows the **New Program** dialog in Studio 5000 Logix Designer and the configuration options that are available:



Programs can be scheduled to execute in a variety of ways within a project, as follows:

- Equipment phases
- Unscheduled programs
- Controller fault handler programs
- Power-up handler programs

Controller routines

Routines contain the code that is executed by the controller. A program can contain multiple routines of mixed IEC languages. Each program is assigned a main routine to be executed. Only the routines that are called by the main routine are executed within a program.

Controller task types

Tasks are the most configurable organization unit and are at the trunk of a Logix project. They provide a means of optimizing the execution of our project. It is important to remember that the Logix controllers are only capable of running one task at a time. However, tasks can interrupt the execution of other tasks if they are triggered and have a higher priority value. Tasks that are interrupted will start again where they left off in the execution of their code.

It is important to keep in mind that other tasks can change the data your routine's code is executing on. Care should be taken to buffer tags in order to avoid the code from entering a state that was not accounted for. Buffering has been covered in detail in *Chapter 5*, *Writing Ladder Logic*.

There are three types of tasks in Studio 5000 Logix Designer (RsLogix 5000):

- Continuous
- Periodic
- Event

Continuous tasks

Continuous tasks will execute as quickly as they can and will always be running on the controller when other tasks are not. After a continuous task has completed the execution, it will immediately start running again. Only one continuous task is allowed to be declared per project. The main task, which is added by default when you create a new project, is automatically set up as a continuous task. A continuous task is typically used as the program flow controller in a project. Using a continuous task is not mandatory, but only one can be used in a project.

Periodic tasks

Periodic tasks will run at an interval you can specify in milliseconds (with a default value of 10 ms). Periodic tasks are typically used in projects for values that must be updated at a specific interval.

Event tasks

Event tasks will run when a condition triggers it. The condition that triggers an event task can be as follows:

- A change of a digital input
- A new sample of analog data
- Motion operations such as axis watch and axis registration
- A consumed tag
- An event instruction
- Microsoft Windows events (SoftLogix only)

The capabilities of your controller may limit the event triggers that are available for your project.

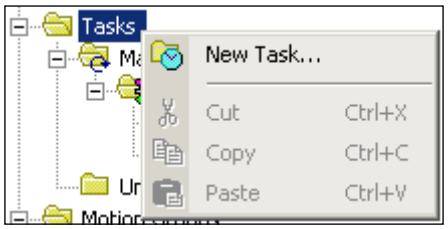
Best practices of Logix task usage

During runtime, there is a small performance hit on the Logix controller when switching between tasks. One should avoid using tasks to organize a project's structure and rely more on programs and tasks to create a logical structure that resembles the process being automated. When many tasks are running on a controller, you increase the risk that some tasks will not have time to complete their program execution before they are triggered again and reset. Often, we hear of programmers who are new to the Logix platform that create dozens of tasks in a project (for example, one per process cell). An experienced automation professional will limit the use of tasks to less than a handful (certainly, there are exceptions). A typical Logix application will have one continuous task to handle the main program execution, and perhaps one or two event tasks to handle special cases.

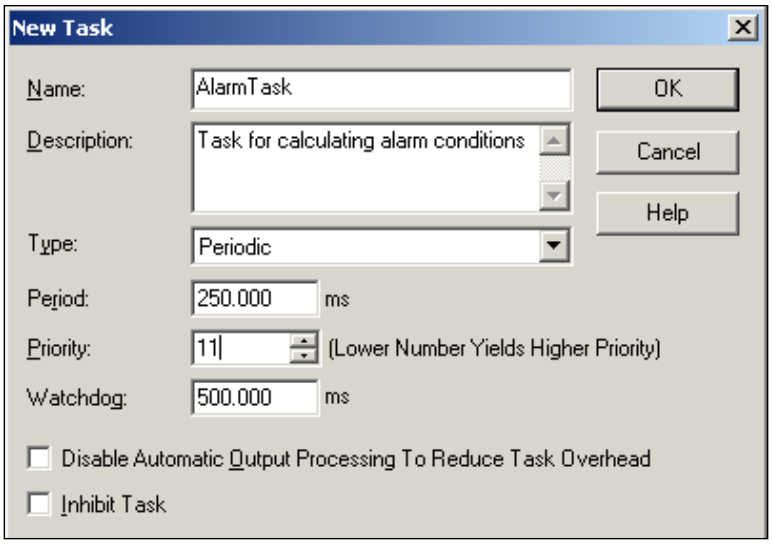
Creating a task

In the following exercise, we will create a simple task by performing these steps:

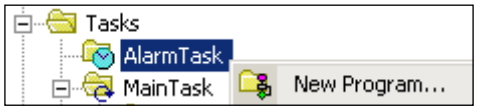
1. In the **Controller Organizer** pane, right-click on the **Tasks** icon, and then click on **New Task…**.



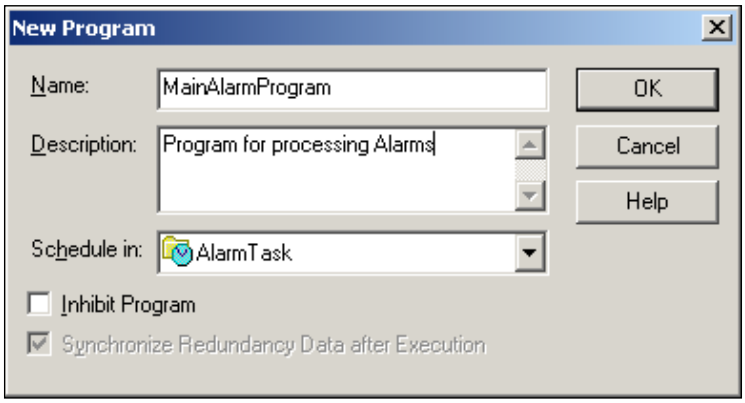
2. In our project, there is no need to check for our non-critical alarms by the default periodic task time of every 10 ms. We will create a new periodic task for processing alarms every 250 ms and give it a low priority in order to reduce the load on our processor.
3. In the **New Task** form that appears, enter the following values:
    - **Name**: `AlarmTask`
    - **Description**: `Task for calculating alarm conditions`
    - **Type**: `Periodic`
    - **Period**: `250.000 ms`
    - **Priority**: `11`
    - **Watch Dog**: `500.000 ms`



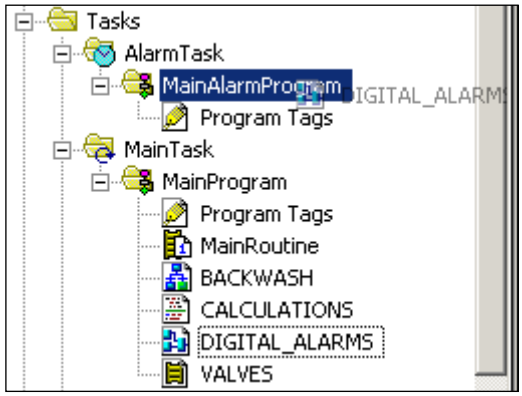
4. Next, we will add our program, which will contain our alarm function and handle the alarm processing for our project. Right-click on our newly created **AlarmTask**, and click on the **New Program…** option.



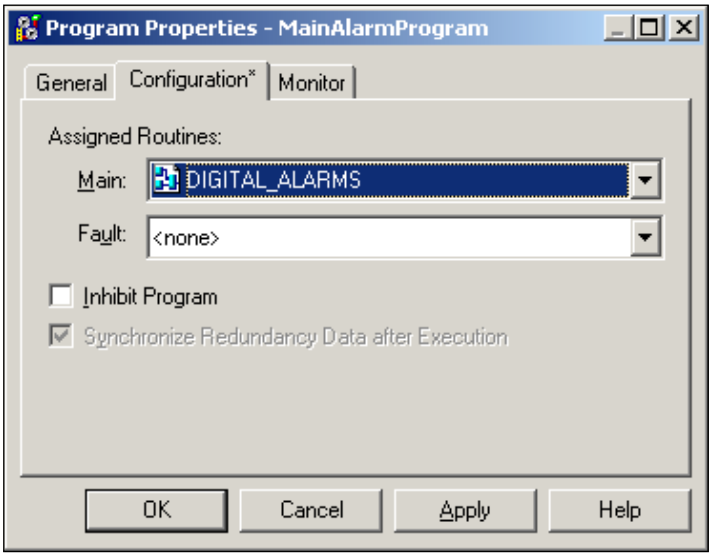
5. In the **New Program** form that appears, enter the following values:
   - **Name**: `MainAlarmProgram`
   - **Description**: `Program for processing Alarms`



6. Now, we can move the **DIGITAL_ALARM** routine that we created in *Chapter 6*, *Writing Function Block*, to our newly created **Alarm** program. Expand the **MainTask** and **MainProgram** scope in the **Controller Organizer** pane's **Tasks** scope, and drag and drop the **DIGITAL_ALARM** routine from **MainProgram** to **MainAlarmProgram**.

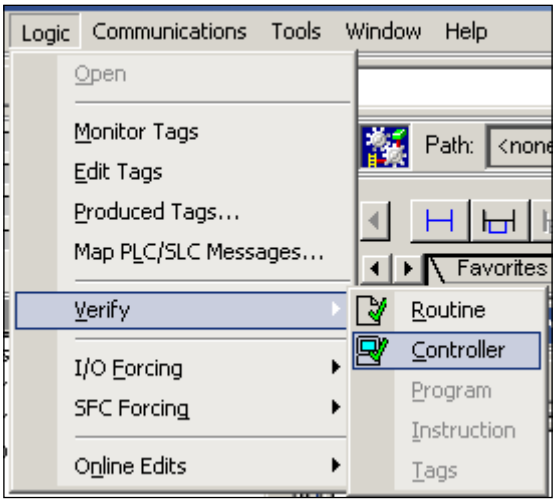


7. We will now set the **DIGITAL_ALARM** routine to be the main routine of **MainAlarmProgram**.
8. Right-click on **MainAlarmProgram** and select **Properties** (or press *Alt* + *Enter*). In the **Program Properties** form that appears, select the **Configuration** tab, and under the **Assign Routines** header, select **DIGITAL_ALARM** from the main drop-down box.



9. You will notice that after assigning the **DIGITAL_ALARM** routine as the main routine for **MainAlarmProgram**, the **DIGITAL_ALARM** icon changes to display a small **1** icon, which indicates that it is the main routine for the program.

10. Next, we will check to see whether we have introduced any errors in our controller with our latest changes. From the drop-down menu at the top of RSLogix 5000, navigate to **Logic** | **Verify** | **Controller**:

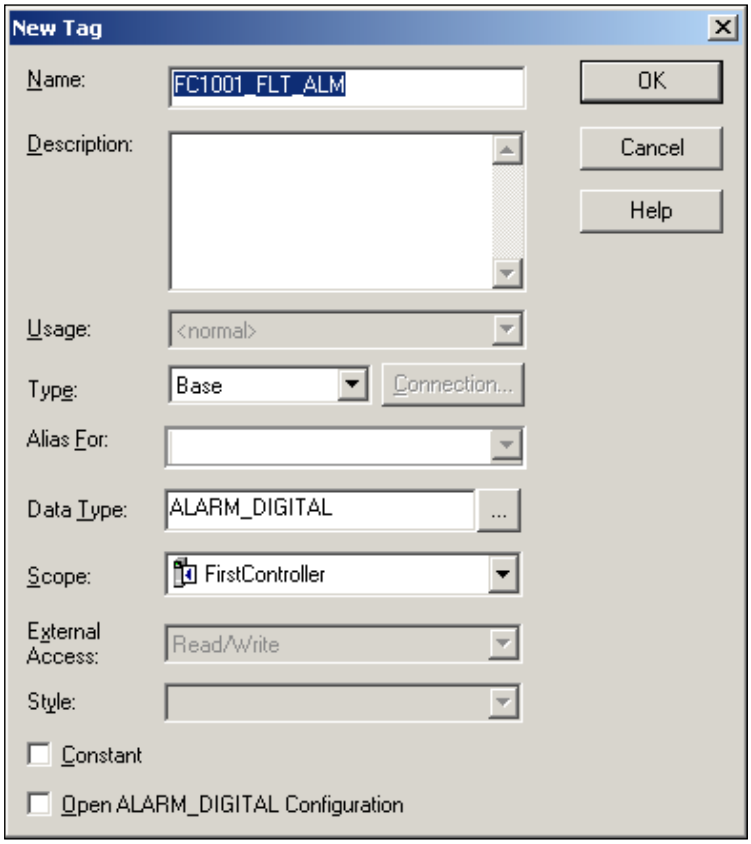


11. You will notice that the **Errors** pane has appeared and the following errors are listed:

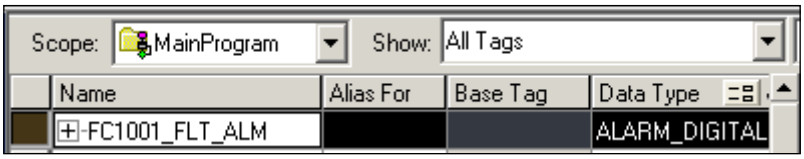
```
Error: Sheet 1, B1, ALMD, FC1001_FLT_ALM: Tag doesn't
  reference valid object or target.
Error: Rung 1, JSR, Operand 0: Invalid reference to unknown
  routine.
```

12. Clicking on the first error message will take you directly to the **DIGITAL_ALARMS** FBD and highlight the **FC1001_FLT_ALM** ALMD element.

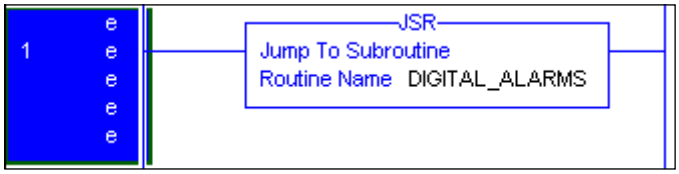
13. The red **X** mark on the FBD element indicates that there is a problem with the block's configuration. The error has occurred because the original **FC1001_FLT_ALM** element was created with a scope of **MainProgram** and it cannot be accessed from the **MainAlarmProgram**. The FBD tags are automatically created with the scope of the current program you are working under when they are added to a routine. In order to declare an FBD at the controller scope (global scope) level, you will need to create it manually using the **New Tag** form.
14. In order to fix this problem, we will need to create the ALMD tag again at a more global scope level. Right-click on the **FC1001_FLT_ALM** tag and select the **New "FC1001_FLT_ALM"** menu option (or press *Ctrl + W*).
15. The **New Tag** form will appear; ensure that the **Scope** field is set to **FirstController** and click on **OK**.



16. Next, we should remove the duplicate **FC1001_FLT_ALM** tag that exists in the **MainProgram** scope. Under the **MainTask** and **MainProgram** folders in the **Controller Organizer** pane, right-click on the **Program Tags** icon and select **Edit Tags**.



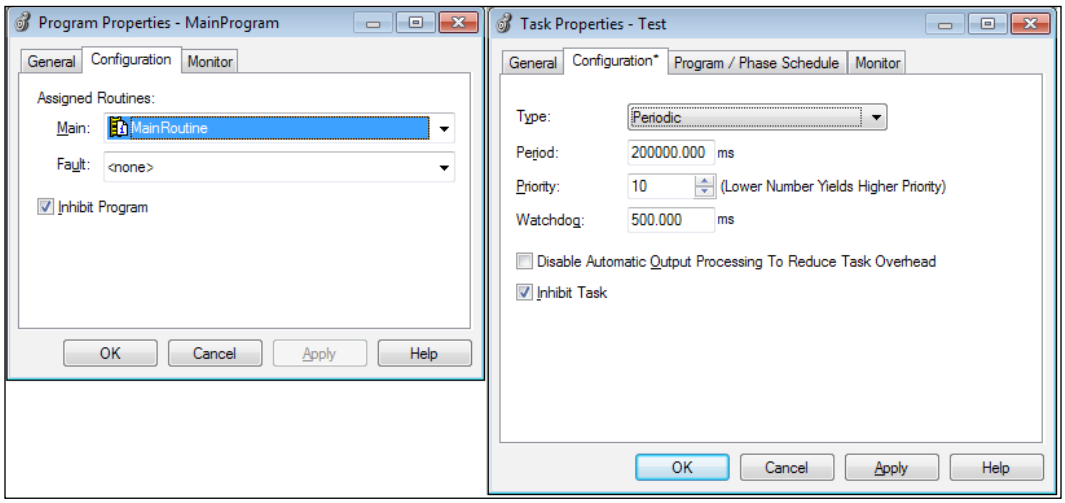
17. The **Edit Tags** table will appear now. Select the **FC1001_FLT_ALM** tag, right-click on the box to the left of the name, and select the **Delete** menu option (or press the *Delete* key).
18. We have now fixed the first error message, now let's resolve the second. Clicking on the second error message in the **Errors** pane will take you directly to the JSR reference within our **MainProgram** scope's **MainRoutine** to the **DIGITAL_ALARM** routine. The error is being displayed because the **DIGITAL_ALARM** routine is no longer in the **MainProgram** scope. Delete this ladder logic rung by right-clicking on it and selecting **Delete** (or by pressing the *Delete* key with the rung selected).



19. Finally, we will verify the program once more to ensure that we no longer have any problems. From the drop-down menu at the top of RSLogix 5000, navigate to **Logic** | **Verify** | **Controller**. The project should now be error-free.

Inhibiting programs and tasks

One advantage of dividing your project into tasks and programs is that you can inhibit them individually if needed and prevent them from executing. To inhibit a task or program, right-click on it and open its properties. The **Inhibit Program** option is in the **Configuration** tab of the **Program Properties** window.



Setting task priorities

Task priorities allow for control over the order in which a task will run on a controller. A Logix controller is only capable of running a single task a time. When multiple tasks happen to occur at the same time, a task's priority setting will allow the controller to select which task it should run first. Within the Logix controllers, there are 15 priority levels (except for SoftLogix, which only has three).

The task's **Priority** option is in the **Configuration** tab of the **Task Properties** window.

Tuning a Logix controller

Logix provides methods for monitoring and tuning the performance of the projects running on a controller. In the next sections, we will explore some of the available options.

System overhead time slice

The system overhead time slice is a controller setting that allows the specification of the percentage of time dedicated to the ongoing background tasks. These on-going background tasks are also known as unscheduled communications and will always require a percentage of the controller's processing power.

Logix unscheduled communications include the following features:

- Programming and device monitoring communication with Logix Designer / RSLogix
- Service communication
- Communication with the HMI devices
- Controller to controller communication
- Synchronize redundant controllers
- I/O connections health monitoring and connection re-establishment

The system overhead time slice only has an impact when a continuous task is configured in a project. When no continuous task is present, the background tasks will take up any extra controller cycles that are available (effectively 100 percent).

A balance must be found between the overhead time slice and the task execution time. When the overhead time slice is set too low, you may encounter the following issues:

- HMIs take a long time to reestablish communications with the controller. With only a short amount of time dedicated to background tasks, it can take the HMI and controller a long time to build the list of tags and initialize communications.
- HMI values are not being updated regularly or stale values are being displayed.
- HMI appears slow to respond to commands and may take longer to switch between page views.
- Logix takes a long time to download or upload a project from a controller.
- Logix takes a long time to connect to the controller or has difficulty connecting.

When the overhead time slice percentage set too high, you may encounter a different set of problems:

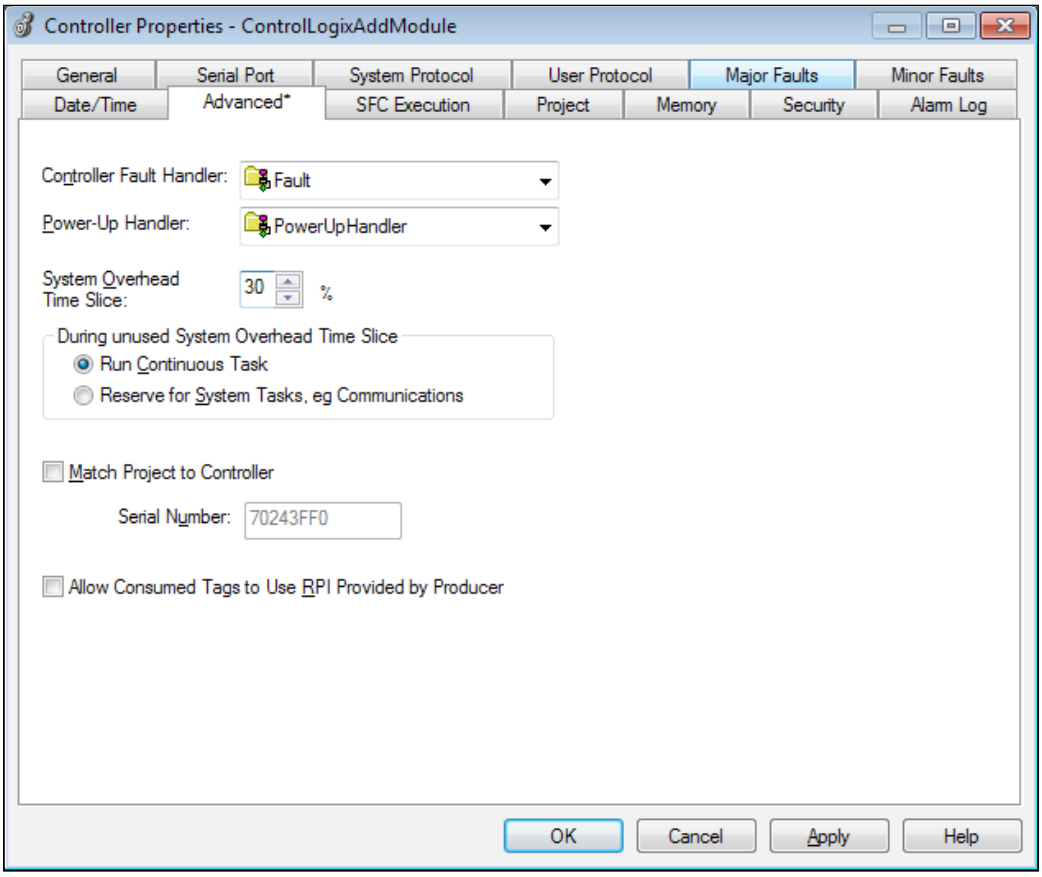
- Programs are not able to complete their execution before they are scheduled to run again (known as overlap)
- Controller CPU utilization is very high

Finding the overhead time slice can be a challenge, particularly on complex programs with resource-constrained controllers. Fortunately, there are tools and techniques provided by Rockwell Automation that provide insights striking the balance.

Setting the system overhead time slice

In the following exercise, we will configure the overhead time slice on the controller of our project by following these steps:

1. In the **Controller Organizer** panel, scroll down to the `I/O Configuration` folder and expand **Backplane** we have configured. Right-click on the controller you have configured and select **Properties**.
2. In the **Controller Properties** dialog box, click on the **Advanced** tab.
3. In the **System Overhead Time Slice** box, raise the overhead time slice to **30**%, and then click on **OK**.



There is also a radio button in the **During unused System Overhead Time Slice** group box that allows us to select **Run Continuous Task** (default) or **Reserve for System Tasks**.

When the **Run Continuous Task** radio button is selected, the controller immediately returns to executing the continuous task after all the unscheduled communications are completed (normal behavior).

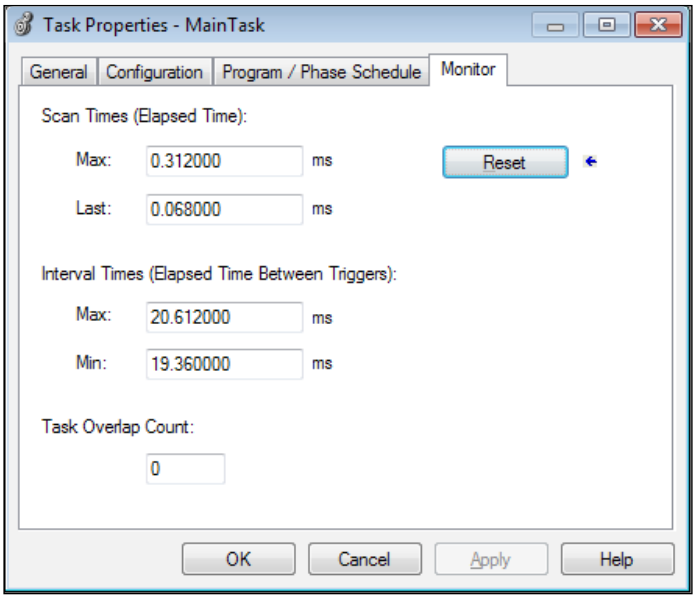
When the **Reserve for System Task** radio button is selected, the controller will always allocate 1 ms to unscheduled communications before returning to the continuous task. This setting is useful for simulating a communication load on the controller (for testing purposes only).

Monitoring task execution time and overlap

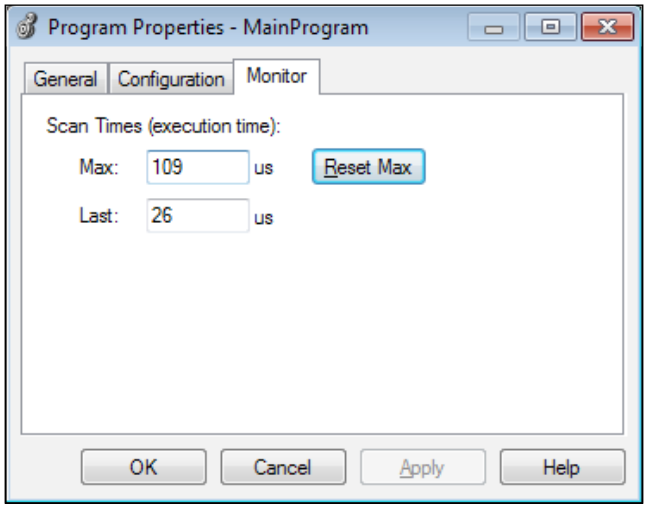
An overlap is when a task is unable to complete the execution of its programs and routines before it is scheduled to run again. Overlaps cause the execution of code to stop and start again from the beginning of the task. They should not occur on a running system and should always be investigated and resolved when reported. In the following exercise, we will learn how to monitor the task execution time and check for the overlap errors by performing these steps:

1. Go online with your program by navigating to **Communications | Go Online**. If prompted, download your project to the controller.
2. Put the controller in the run mode by navigating to **Communications | Run Mode**.
3. Right-click on the **MainTask** scope in the **Controller Organizer** pane, open **Properties**, and select the **Monitor** tab.

4. When Logix is online with the controller, you will see live **Scan Times** (time required to execute the task), **Interval Times**, and **Task Overlap Count** information. If the **Task Overlap Count** value is ever greater than zero, you should investigate the cause and work to optimize the execution of your project.

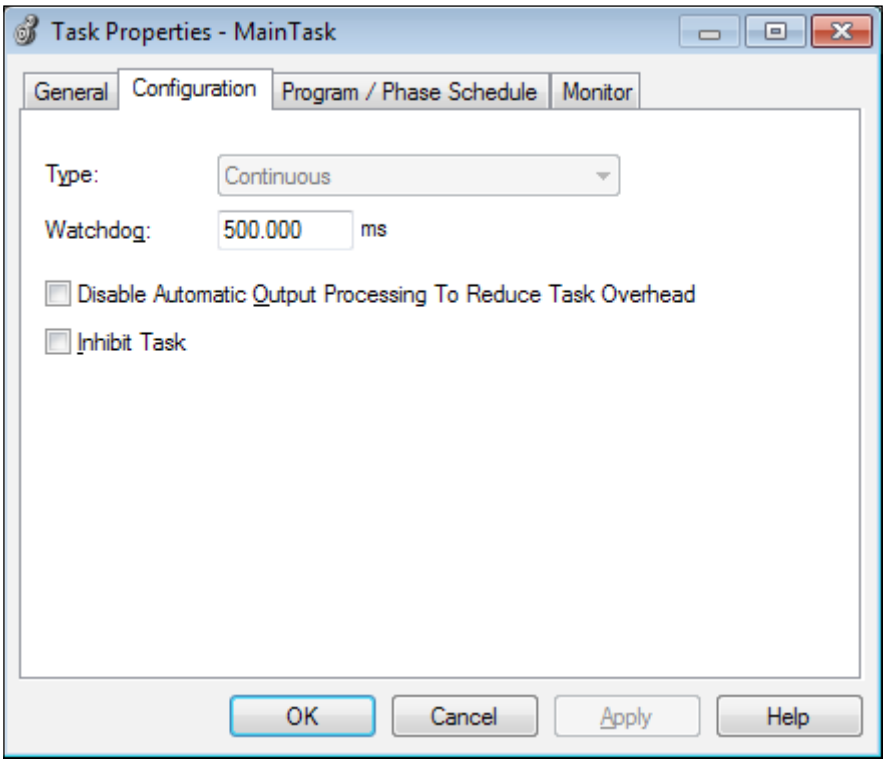


5. Right-click on the **MainProgram** scope in the **Controller Organizer** pane, open **Properties**, and select the **Monitor** tab. Here, you will see the **Max** and **Last** scan time fields (time required to execute **MainProgram**).



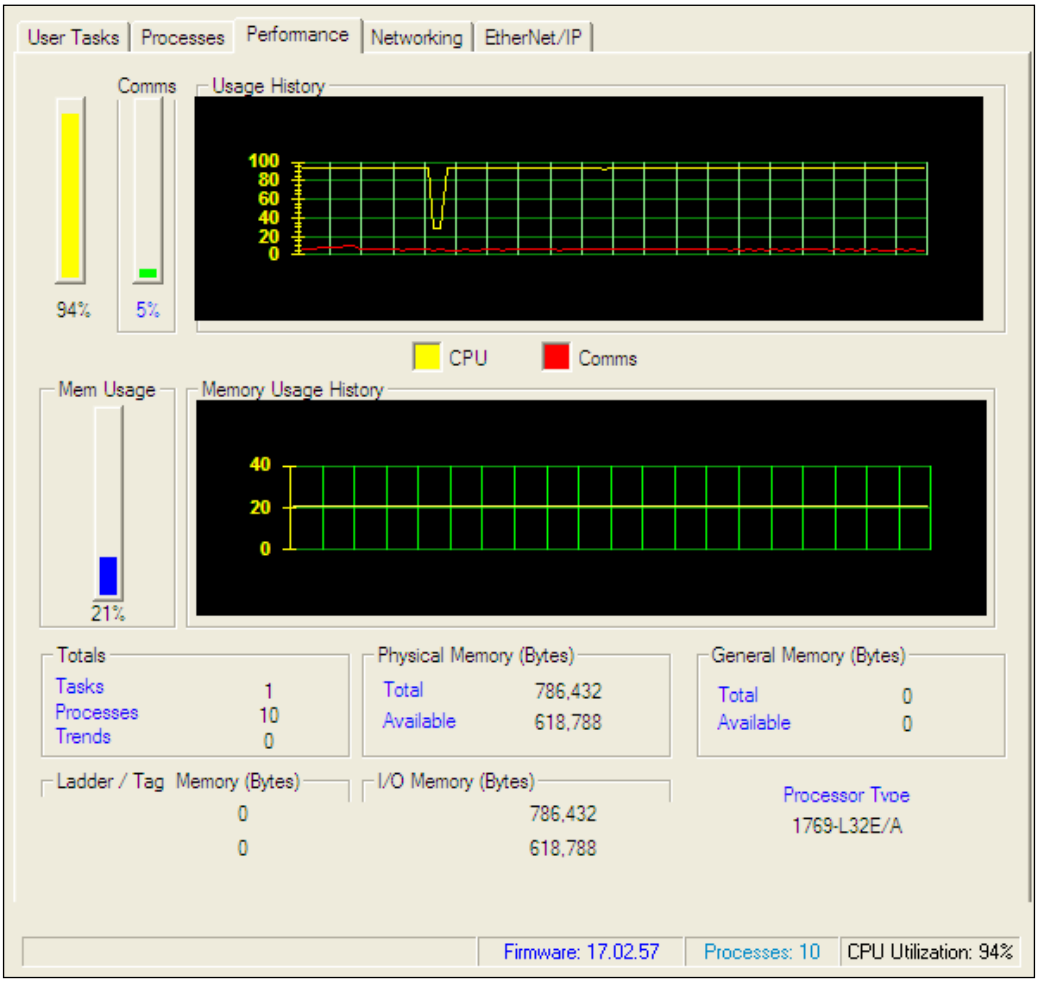
Task watchdog time

Tasks also allow you to specify a watchdog time, which will trigger a major fault if the task runs for too long. The default watchdog time for a task is 500 ms and the watchdog time includes interruptions by other higher priority tasks. Depending on how your project's fault handler is configured, the watchdog time may cause your controller to stop executing, so use this feature with caution.



Logix5000 Task Monitor tool

The **Logix5000 Task Monitor** tool shows the current load of the Logix CPU. It provides insights in the current tasks that are running on the controller, active connections, and memory usage. This tool can be downloaded from the Rockwell Automation website or found on the Rockwell Automation distribution CD or hard disk drive. This tool can provide insights into the way your controller's CPU is being used. You can tune your Logix application by optimizing code, adjusting the task configurations, and modifying the overhead time slice percentage.



Summary

In this chapter, we further investigated the project organizational units we have been using throughout this book. We detailed the way a Logix controller executes tasks and how the CPU divides its time based on priority. We introduced the overhead time slice and emphasized its importance when optimizing a Logix application. Finally, we investigated methods within the Logix platform to monitor and troubleshoot performance issues.

In the next chapter, we will dig deeper into troubleshooting controller issues and faults.

10
Faults and Troubleshooting in Logix

In this chapter, we learn how to identify and troubleshoot faults in a Logix controller. We will also detail a list of fault codes that provide insight into the problems encountered by the platform. We will introduce the process of fault recovery, which allows a program to resume the execution after encountering a specific fault type. We will then look at the convenient troubleshooting applications available for your iPhone and iPad. The following topics will be covered in this chapter:

- Logix troubleshooting and support
- Clearing faults
- Fault handling and recovery
- Trapping a fault
- Rockwell troubleshooting applications

General troubleshooting and support for Logix

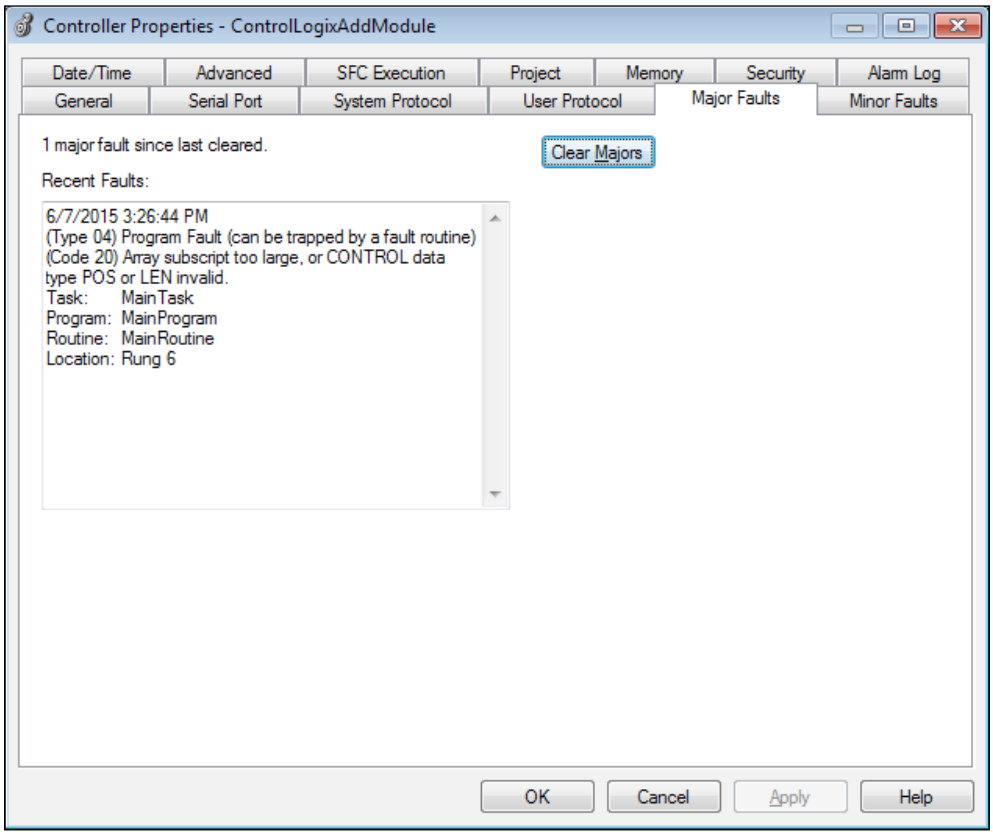
Automation problems have an odd way of occurring at the most inconvenient time, such as on Christmas Eve or your partner's birthday. Regardless of the timing, there is always a requirement to quickly resolve problems as they often translate to lost production time and lost revenue for the company that owns the process. Here are a few best practices for troubleshooting in the Logix platform:

- Get familiar with Rockwell Automation **knowledgebase**. The issue you have encountered has most likely been encountered by someone else too. The Knowledgebase is a great first line of defense for solving a problem. If you don't have an account set up, you should create one before you need it at `http://www.rockwellautomation.com/services/online-phone/techconnect.page`.
- Verify that you have an active Rockwell Automation support contract and keep your Rockwell Automation technical support information and authorization number handy. Rockwell Automation provides world class support, so make sure you understand how to access it before you need it. It also provides a handy wallet size cut out of your technical support information and it is a good idea to carry it around with you as you never know when you are going to get a call from your work asking for help.
- Get to know your local Rockwell Automation sales representative; they can also help you to quickly escalate support cases if required. Buy them coffee or lunch (or let them buy you coffee/lunch) and foster a relationship. They want to help and see you succeed.
- Get involved with the Rockwell Automation community. Sign up for their terrific (and free) TechConnect Education Webinars, Genius Webinars, and attend local lunch and learns or conferences. There is no substitute for in-depth knowledge of the Logix platform when troubleshooting problems.

An introduction to troubleshooting faults

A fault is an error state in your controller that will prevent it from executing normally. It must be resolved in order for the controller to resume its normal execution. There are many reasons that would cause an automation process to stop. There are a few ways of confirming that a controller fault is the cause of a process upset:

- An operator may notice a controller fault is being indicated from an HMI display
- The fault light is lit on your controller or a fault code is being listed on the scrolling status display on the controller
- We can also see that fault indicated when going online with our controller from RSLogix5000 / Logix Designer



A major fault as seen from the Controller Properties window

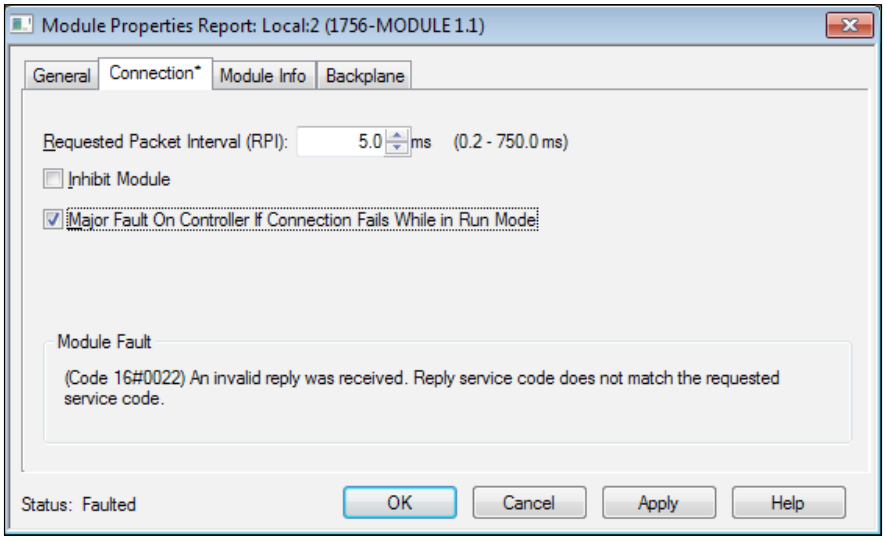
Troubleshooting faults in Logix is the process of locating and resolving problems encountered during the execution of a project on a controller. Access to **piping and instrumentation diagram** (**P&ID**) drawings, cause and effect diagrams, wiring drawings, and junction box drawings certainly will help the cause. However, there is no substitute for the working knowledge of a control system. Troubleshooting is an analytical process of elimination to narrow down and permanently fix the problem. Faults are a mechanism used in the Logix controllers (and by other PLC/PAC vendors) to stop the execution of process when a problem is detected. Capturing and handling faults is the first step when determining the root cause of a serious control system problem. In the next few sections, we will investigate faults and how to handle them in a project.

Faults

Within Logix, there are four categories of faults:

- **Major**: This fault is a fault in which the Logix controller will stop executing routines. An I/O module will allow you to configure the program to enter either a fault mode or program mode when a major fault occurs. Also, the I/O module outputs are set to their configured value for the faulted mode. There are 77 different types of major faults listed in the Rockwell Automation Literature Library document, "Logix5000 Controllers, Major, Minor, and I/O Faults", for example, when the controller detects a problem with the chassis, a major fault will be triggered.
- **Minor**: This fault will trigger a bit to go high in the **MinorFaultBits** system status class object, but the controller will continue to run without interruption. There are 31 different types of minor faults listed in the same document, for example, low battery or energy storage status. When a problem is detected with the L6 battery or the L7 **Energy Storage Module** (**ESM**) MinorFaultBits, bit 10 will be high. Also, the battery light on the front panel of the L6 controller will be lit when this bit is high.

- **I/O**: This fault is due to a problem with an I/O module, which, by default, will not cause the Logix controller to stop executing routines. However, on the individual I/O module properties, in the **Connection** tab, the controller can be configured to trigger a major fault on connection failure. The following screenshot displays the I/O module's properties with the major fault on the I/O connection failure option checked and an active I/O fault code:



  There are 95 different types of I/O faults listed in the above-mentioned document.

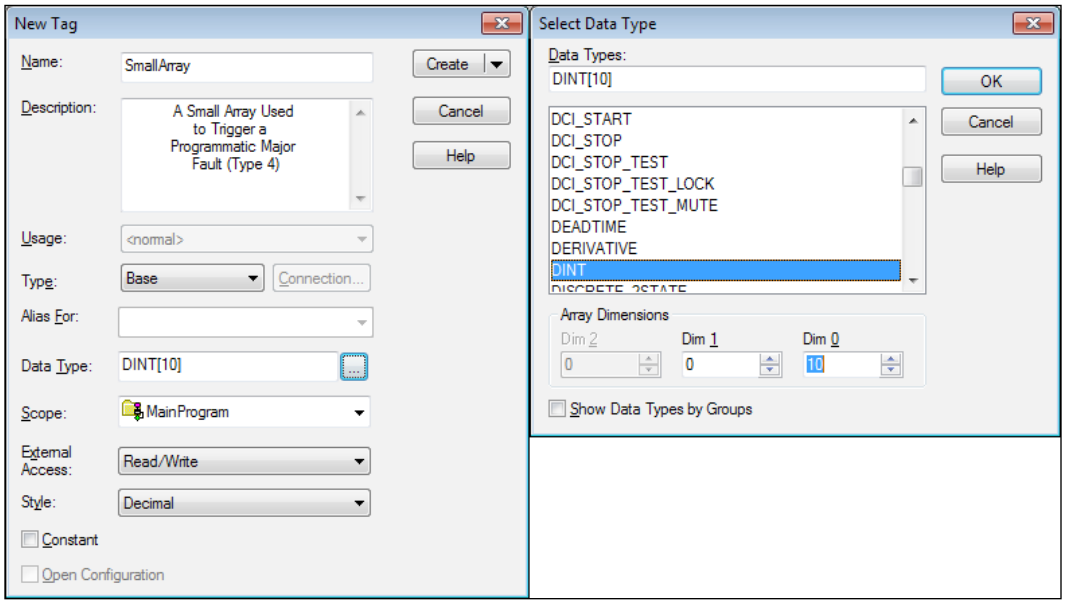
- **User-defined major**: This fault is a custom fault that can be triggered based on any automation condition using a JSR instruction to the fault handler routine (more on that later) and by passing a numeric parameter of 990 to 999 (the reserved range for the user-defined fault codes). A user-defined major fault is handled by the controller just like any other major fault: stopping the execution of routines and putting the output I/O modules to their fault values.

  There are 10 different types of user-defined faults available in the same document, such as type 4 code 990 and type 4 code 999.

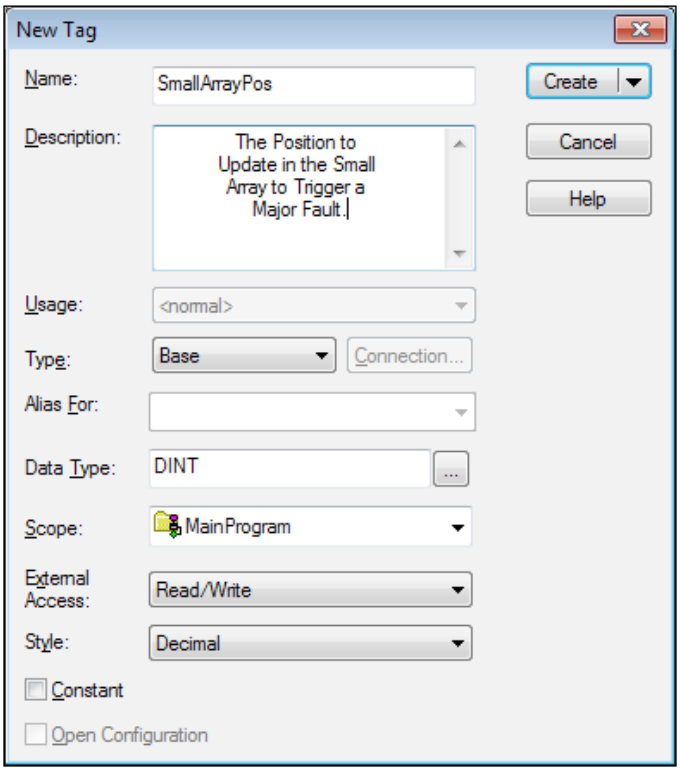
Clearing a fault

Once a controller encounters a major fault, not much can be done with it until the fault has been cleared. You will be unable to download an updated program to your controller until you clear all the major faults. In the following exercise, we will trigger a major fault, and then learn how to view and manually clear it. We will trigger a programmatic major fault (type 4) by referencing a value outside of an array's configured range. We will need to add two more tags to our project in order to trigger the major fault—an array and array position tag. Perform these steps:

1. Create an array tag by right-clicking on the program tags in the **Controller Organizer** pane and select **New Tag** (or press *Ctrl* + *W*), and then create an array called **SmallArray** with the following properties:
    - **Name**: `SmallArray`
    - **Description**: `A Small Array Used to Trigger a Programmatic Major Fault (Type 4)`
    - **Data Type**: **DINT[10]**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**

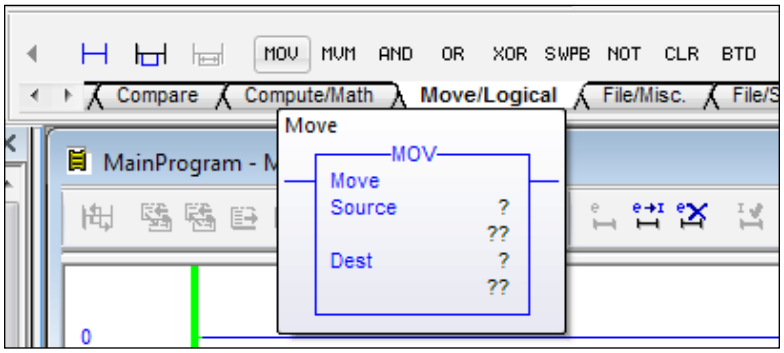


2. Next, create a **DINT** tag with the following parameters:
    - **Name**: `SmallArrayPos`
    - **Description**: `The Position to Update in the Small Array to Trigger a Major Fault.`
    - **Data Type**: **DINT**
    - **Scope**: **MainProgram**
    - **External Access**: **Read/Write**
    - **Style**: **Decimal**

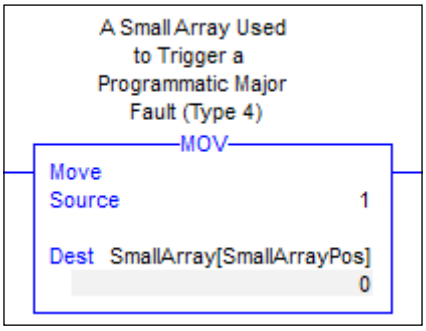


3. Next, we will add a ladder rung to the **MainRoutine** program, which we can use to trigger the major fault. Open the **MainRoutine** program from the **Controller Organizer** pane and add a new rung to the end of the routine by right-clicking on the last rung (**End**) and selecting **Add Ladder Element…** (or by pressing *Alt* + *Insert*).

4. On the newly created routine, add an **MOV** instruction by navigating to the **Move/Logical** element group and clicking on the **MOV** element.



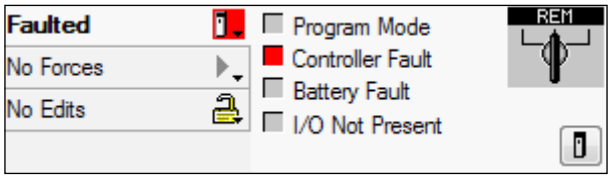
5. Next, set the **Source** measure of the **MOV** instruction to a literal value of **1**, and then set the **Dest** value of the **MOV** instruction to the newly created **SmallArray** and **SmallArrayPos** as the array position value, **SmallArray[SmallArrayPos]**.



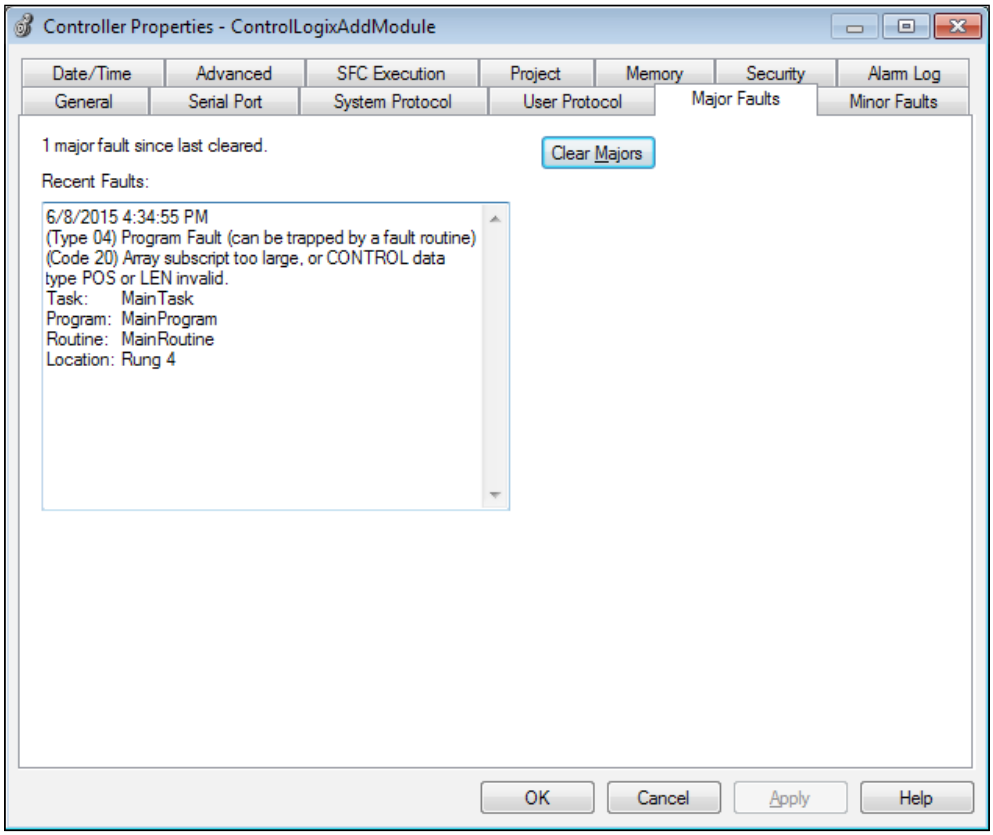
6. The **MOV** instruction will allow us to overflow the array and trigger an array subscript to large major fault. Download the updated project to the controller, and go online. Open the program tags for the main program and update the value of **SmallArrayPos** to **11**.

+ SmallArray	{...}	DINT[10]	A Small Array Used to Trigger a Programmatic Major Faul...
+ SmallArrayPos	11	DINT	The Position to Update in the Small Array to Trigger a Maj...

7. Overflowing the array triggers a major fault in the controller. The controller status now displays as **Faulted**.



8. The major fault can be viewed by navigating to **Next Communications | Go To Faults**. Click on the **Clear Majors** button to clear the fault.



9. In order to clear the fault condition and continue the normal operation of the controller, we will adjust the **SmallArrayPos** value back down to **1**.
10. Finally, start changing the controller back to run mode by navigating to **Communications | Run Mode**.

Fault handling and recovery

Minor faults should be reported to the process operator either by updating a bit for the HMI or by illuminating a light on a panel faceplate. Major faults, however, should be trapped and automatically recovered, when possible. Some major faults cannot be recovered and this further divides major faults into two categories:

- Major recoverable faults
- Major unrecoverable faults

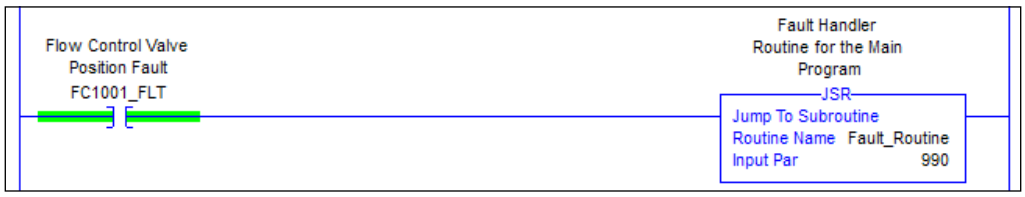
Logix allows the user to create a fault routine, which can trap a fault and attempt to recover and continue running the process when possible. When a Logix controller encounters a major fault, it will execute the program assigned to the controller fault handler.

It is a recommended best practice to at least log all faults that occur in the controller before programmatically clearing them. As discussed earlier, faults play an important role in troubleshooting problems with a process and should never be ignored.

If the fault is still present after the controller fault handler has been executed, the controller will stop running. Fault routines can be declared at a program scope or at the controller scope (for capturing faults during tag assignment, startup prescan, and SFC postscan).

It is important to note that when a major fault occurs during the execution of a logic instruction, like in the preceding exercise, the controller does not actually execute that instruction. The controller will simply move down to the next instruction in the routine.

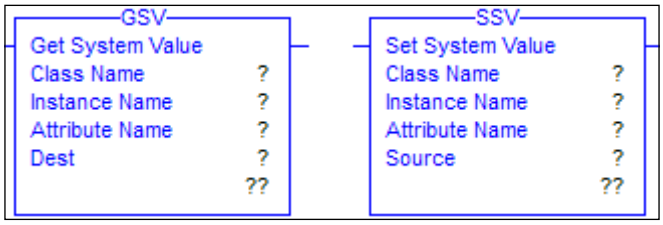
Fault routines can also be executed using a JSR instruction with a parameter of 990 to 999 (the reserved range for the user-defined fault codes).



In the next section, we will introduce the instructions used to check and reset the major and minor fault information from the controller system values.

Get System Value and Set System Value

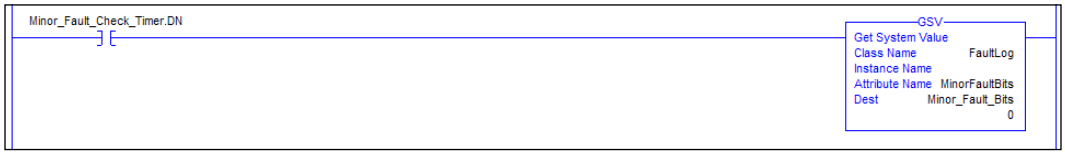
Legacy Rockwell PLCs (PLC5 and SLC500) had a dedicated status file (S2), which was continuously updated by the controller with system values. The ControlLogix platform has removed the dedicated status file in order to reduce the controller processing load. In place of the dedicated status file, ControlLogix has provided two self-serve instructions that allow for direct control over the system value access that are neatly organized into class objects. The **Get System Value** (**GSV**) instruction will retrieve a system value status from the controller that can update a specified destination tag with that value. The **Set System Value** (**SSV**) instruction will update a controller system value status from a specified source tag.



GSV and SSV instructions

The GSV and SSV instructions can be accessed from ladder logic and structured text and are not directly available in function block. Typically, you will see these values scheduled periodically (for example, with a **TON** instruction) so as not to burden the controller's processor. The GSV and SSV instructions play an important role in retrieving fault information and resetting the fault before causing the controller to stop:

- The `FaultLog` class name and `MinorFaultBits` attribute name system value can be used to check for any minor faults present in the controller



Timer (TON) instruction executing a GSV to collect MinorFaultBits

- The `MajorFaultRecord` system value attribute can be used to check for the presence of major fault and collect its details

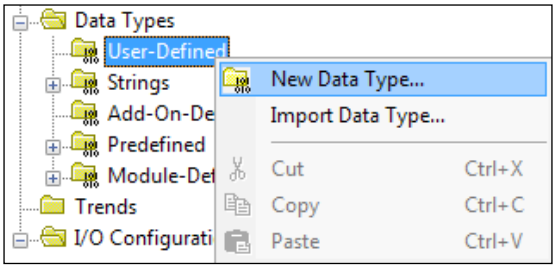
User-defined data types

User-defined data types are an object-based construct that allows you to create custom structures containing a number of different data types. User-defined data types are used to organize data into objects that align with the properties of physical-world equipment. In the case of fault trapping, we will be using a UDT to capture the details of a major fault.

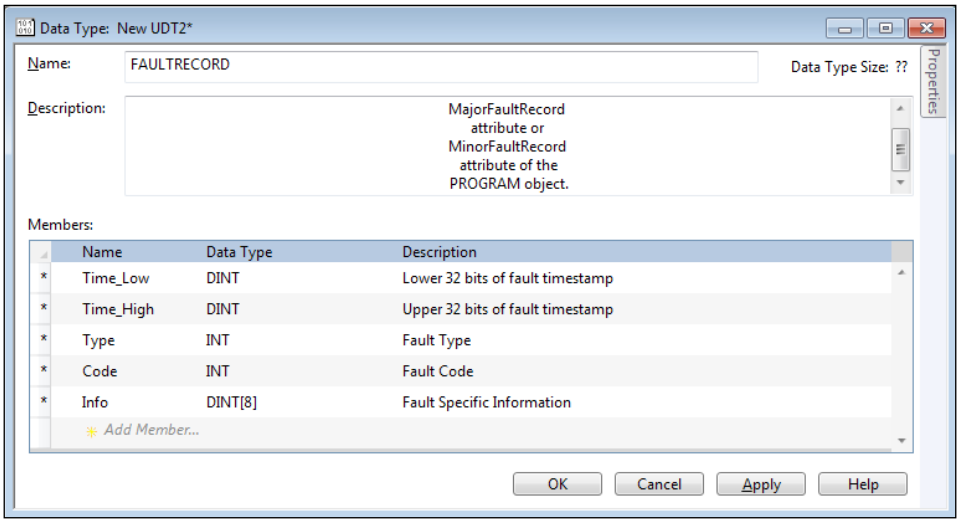
Trapping a fault

In the following exercise, we will trap a major fault and clear it before the controller stops with these steps:

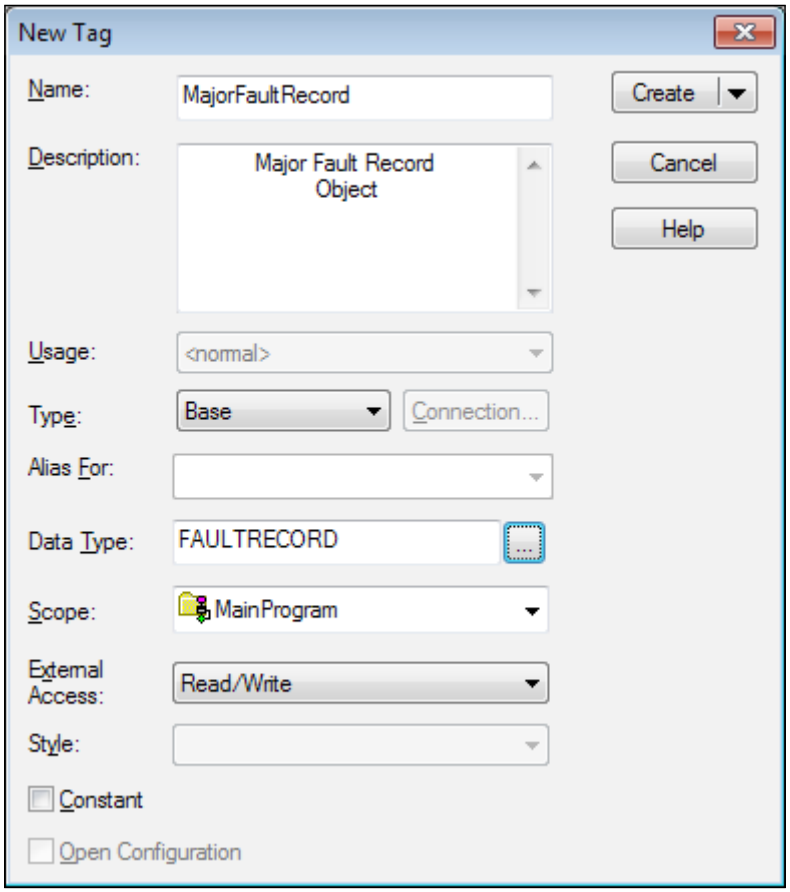
1. In order to trap a major fault, we must create a user-defined data type to store the fault information.



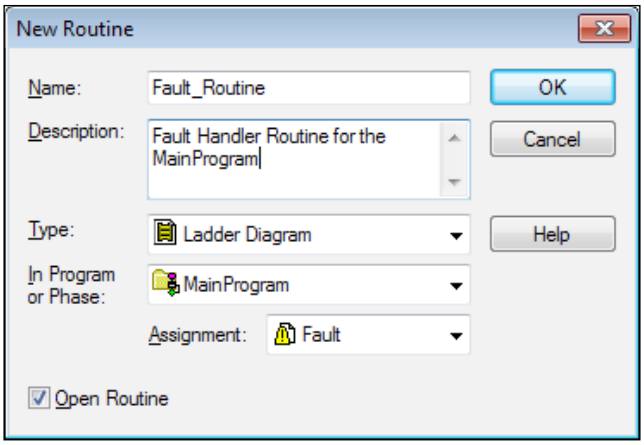
2. In the **Data Type** window that appears, creates the following `FAULTRECORD` user-defined data type:



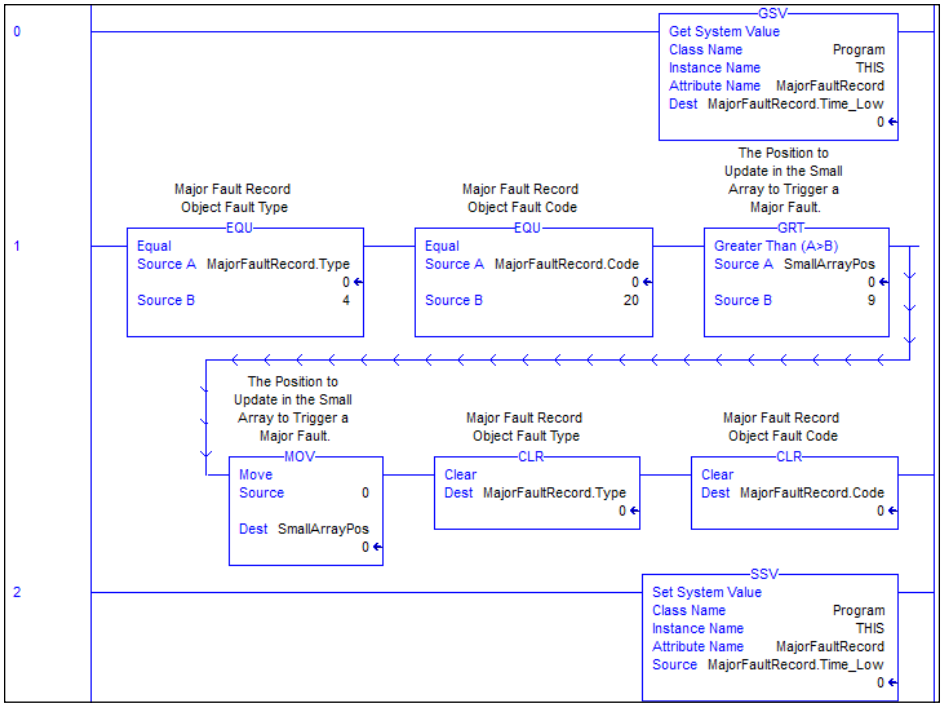
3. We will need to add an instance of our newly created **FAULTRECORD** user-defined data type to our project at the **MainProgram** scope level.



4. Next, we will need to create a fault handler routine to trap the array overflow major fault in the scope of the **MainProgram** scope. Under the **MainProgram** scope, create a new ladder logic routine with the name, `Fault_Routine`, and set the **Assignment** value to **Fault**:



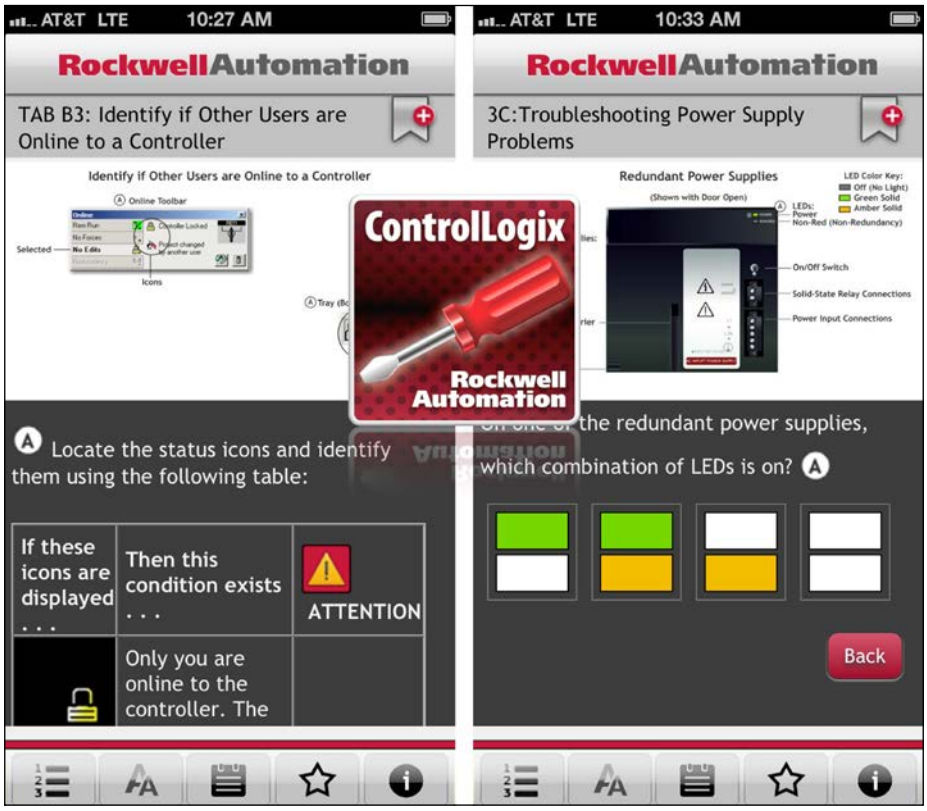
5. Add the following ladder logic instructions to check for our fault type and fault code, and clear the major fault and the condition causing it.



6. Download and run the project on your controller. You will find that when you try to trigger a major fault using the **SmallArrayPos** value, as we did in the previous exercise, the **SmallArrayPos** value will be reset to zero and the controller will no longer trigger a major fault.

Rockwell troubleshooting application for iPad and iPhone

Rockwell Automation provides a handy app for your Apple iPad or iPhone that offers troubleshooting guidance and support. The app will allow you to look up for information on the fault codes and will step you through the process of troubleshooting the issue to resolution. The app includes a ton of embedded resources, which can be extremely valuable when troubleshooting an issue at a remote site, such as `https://itunes.apple.com/us/app/rockwell-automation-controllogix/id662157429?mt=8`.



The Rockwell Automation troubleshooting app for the Apple iPhone and iPad

Summary

In this chapter, we provided recommendations for improving your troubleshooting capabilities in the Logix platform. We also learned how to identify and troubleshoot the various types of faults that can occur on a Logix controller. We used ladder logic to trigger a major fault, and then we learned how to trap the major fault and prevent the controller from stopping when it occurs. Finally, we highlighted a helpful app provided by Rockwell Automation for troubleshooting the Logix issues while in the field from your iPhone or iPad.

Rockwell Automation Literature Library Resource

Some useful links for learning Rockwell Automation are mentioned as follows:

- 1756 ControlLogix Controllers (Technical Data):

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/td/1756-td001_-en-p.pdf`

- Logix 5000 Controllers IEC 61131-3 Compliance:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm018_-en-p.pdf`

- Safety Accelerator Toolkit (Quick Start) to learn basic procedures for developing safety applications:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/qs/iasimp-qs005_-en-p.pdf`

- GuardLogix Controllers (User Manual) to learn how to configure, program, and operate a GuardLogix controller:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/1756-um020_-en-p.pdf`

- GuardLogix Controller Systems to learn more about safety system details for a GuardLogix controller:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/rm/1756-rm093_-en-p.pdf`

- GuardLogix Safety Application Instruction Set to learn more about safety application instructions:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/rm/1756-rm095_-en-p.pdf`

- ControlLogix-XT Extreme Environment System – ControlLogix-XT and Flex I/O-XT products:

  `http://ab.rockwellautomation.com/Programmable-Controllers/ControlLogix-Extreme-Environment-Controllers`

- CompactLogix System (Selection Guide)

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/sg/1769-sg001_-en-p.pdf`

- RSLinx Classic (Getting Results Guide):

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/gr/linx-gr001_-en-e.pdf`

- RSLinx Enterprise (Getting Results Guide):

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/gr/lnxent-gr001_-en-e.pdf`

- Technical Capabilities of the DF1 Half-Duplex Protocol to learn the DF1 networks:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/wp/1761-wp003_-en-e.pdf`

- ControlNet Network Configuration:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/cnet-um001_-en-p.pdf`

- ControlNet Coax Media Planning and Installation Guide to learn the ControlNet networking components:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/in/cnet-in002_-en-p.pdf`

- DeviceNet Network Configuration:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/dnet-um004_-en-p.pdf`

- RSNetworkx (Getting Results Guide):

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/gr/dnet-gr001_-en-e.pdf`

- Ethernet Design Considerations to learn about the EtherNet/IP networks:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/rm/enet-rm002_-en-p.pdf`

- EtherNet/IP (Media Planning and Installation Manual):

  `http://www.odva.org/Portals/0/Library/Publications_Numbered/PUB00148R0_EtherNetIP_Media_Planning_and_Installation_Manual.pdf`

- 1756 ControlLogix I/O Specifications (Technical Data):

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/td/1756-td002_-en-e.pdf`

- 1756 ControlLogix HART Analog I/O Modules:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/1756-um533_-en-p.pdf`

- ControlLogix Digital I/O Modules:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/1756-um058_-en-p.pdf`

- Logix5000 Controllers I/O and Tag Data:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm004_-en-p.pdf`

- Logix5000 Controllers Program Parameters:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm021_-en-p.pdf`

- RSNetWorx for EtherNet/IP (Getting Results Guide):

  `https://www.rockwellautomation.com/resources/misc/html/global-assets/rockwellsoftware/get/ENET-GR001A-EN-P.pdf`

- SoftLogix 5800 System:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/1789-um002_-en-p.pdf`

- Logix5000 Controllers Program Parameters:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm021_-en-p.pdf`

- Logix5000 Programing Software Edition Comparison:

  `http://www.ab.com/en/epub/catalogs/12762/2181376/2416247/360807/1837528/RSLogix-5000-Programming-Software.html`

- Logix5000 Controllers Tasks, Programs, and Routines:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm005_-en-p.pdf`

- Logix5000 Controllers Major, Minor, and I/O Faults:

  `http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm014_-en-p.pdf`

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

Thank you for buying
Learning RSLogix 5000 Programming

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

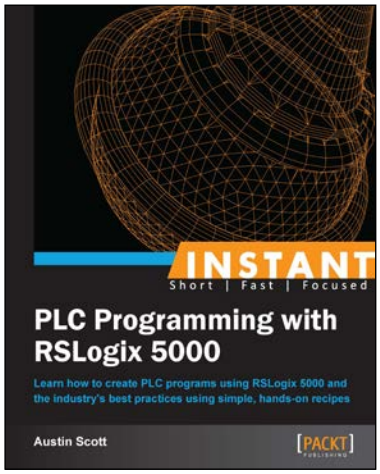
Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING

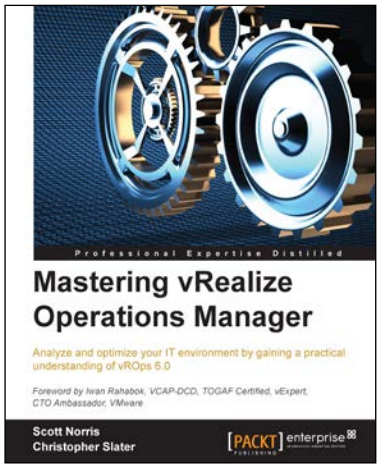


Instant PLC Programming with RSLogix 5000

ISBN: 978-1-84969-844-3 Paperback: 68 pages

Learn how to create PLC programs using RSLogix 5000 and the industry's best practices using simple, hands-on recipes

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create Ladder Logic (LL), Functional Block Diagrams (FBD), Structured Text (ST), and Sequential Function Chart (SFC) routines.
3. Explore object-orientated features such as user-defined types and routine generation techniques.



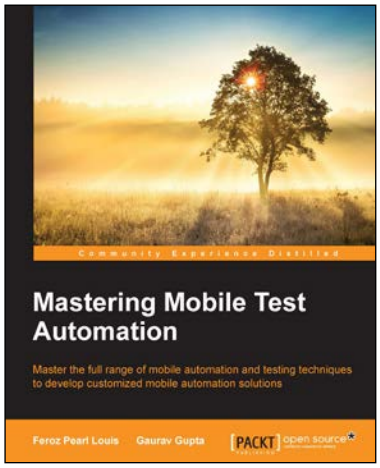
Mastering vRealize Operations Manager

ISBN: 978-1-78439-254-3 Paperback: 272 pages

Analyze and optimize your IT environment by gaining a practical understanding of vROps 6.0

1. Get complete control of capacity management in your virtual environment.
2. Display the most appropriate performance metrics and assemble your own dashboard.
3. Analyze and process data from different sources into a single repository, allowing you to understand every layer of your environment.

[PACKT]
PUBLISHING

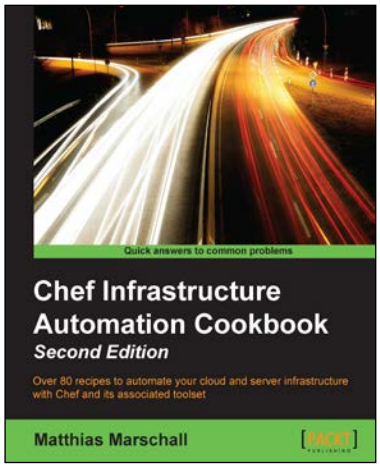


Mastering Mobile Test Automation

ISBN: 978-1-78217-542-1 Paperback: 274 pages

Master the full range of mobile automation and testing techniques to develop customized mobile automation solutions

1. Design and develop top-notch, efficient, and scalable mobile automation frameworks.
2. Develop automation solutions quickly and effectively using real, emulated devices and mobile-specific tools.
3. A comprehensive and resourceful guide to automate mobile applications through user agents, emulators, and simulators.



Chef Infrastructure Automation Cookbook

Second Edition

ISBN: 978-1-78528-794-7 Paperback: 278 pages

Over 80 recipes to automate your cloud and server infrastructure with Chef and its associated toolset

1. Automate error-prone and tedious manual tasks and manage your servers on-site or in the cloud.
2. Equip yourself with the Chef development kit, and learn how to create simple Chef cookbooks and various other artifacts for managing systems with Chef when live.
3. Packed with working code and easy-to-follow, step-by-step instructions to configure, deploy, and scale your applications.