GE Fanuc Automation

State Logic Products

Series 90™ -30
Diagnostic System

For Reducing Downtime in Systems Controlled by Series 90-30 PLCs

User's Guide

GFK-0828 *July 1993*

Warnings, Cautions, and Notes as Used in this Publication

Warning

Warning notices are used in this publication to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist in this equipment or may be associated with its use.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.

Caution

Caution notices are used where equipment might be damaged if care is not taken.

Note

Notes merely call attention to information that is especially significant to understanding and operating the equipment.

This document is based on information available at the time of its publication. While efforts have been made to be accurate, the information contained herein does not purport to cover all details or variations in hardware or software, nor to provide for every possible contingency in connection with installation, operation, or maintenance. Features may be described herein which are not present in all hardware and software systems. GE Fanuc Automation assumes no obligation of notice to holders of this document with respect to changes subsequently made.

GE Fanuc Automation makes no representation or warranty, expressed, implied, or statutory with respect to, and assumes no responsibility for the accuracy, completeness, sufficiency, or usefulness of the information contained herein. No warranties of merchantability or fitness for purpose shall apply.

This User's Guide contains information regarding State Logic® Products available through GE Fanuc.

®State Logic is a registered trademark of ADATEK, Inc.

The following are trademarks of GE Fanuc Automation North America, Inc.

Alarm Master	CIMSTAR	Helpmate	PROMACRO	Series Six
CIMPLICITY	GEnet	Logicmaster	Series One	Series 90
CIMPLICITY 90–ADS	Genius	Modelmaster	Series Three	VuMaster
CIMPLICITY PowerTRAC	Genius PowerTRAC	ProLoop	Series Five	Workmaster

Content of this Manual

Chapter 1. Getting Started – Getting Started is the section you are now reading. Getting Started tells you how to install UPTime on your DOS based computer and other particulars related to accessing information.

Chapter 2. Diagnostic Strategy Overview – UPTime is an interface that allows you to tap into the substantial power and flexibility of State Logic programming plus providing simple to use Fill–in–the–Blank Diagnostics. This chapter provides some basics about the underlying concepts and philosophy of State Logic programming and Diagnostics. **Regardless of what you may already know about State Logic, it is extremely important that you read this chapter carefully.**

Chapter 3. Creating a Diagnostic Strategy – This chapter explains how to create a Diagnostic Strategy using UPTime. A procedure for structuring State Logic programming and information about using Diagnostics are provided.

Chapter 4. Tutorial – This chapter walks you through the creation and execution of a simple Diagnostic Strategy programmed in State Logic with UPTime. Some simple program sections are created then executed. Building on previous examples, a complete Diagnostic Strategy is created and executed.

Chapter 5. Diagnostic Processor – This chapter explains the details of using the Diagnostic Processor. Topics discussed are installing and troubleshooting the module, serial communications, details of the scanning system, and interfacing with the CPU of the system.

Chapter 6. The UPTime Menu System – This chapter describes some of the options available from the UPTime menu system. The options covered are those that are not used all the time and might be difficult to understand.

Chapter 7. Fill–in–the–Blank–Diagnostics – This chapter contains detailed information about creating and using Diagnostics. The four types of Diagnostics are described and the debug mode operations used to view and record Diagnostic actions are explained.

Chapter 8. State Logic Programming – This chapter has all of the details about State Logic programming. The use of the State Logic language is discussed, including language structure and use of English words in the program, and the Perform Functions.

Appendix A. Specifications – This appendix has specifications for both the UPTime software package and the Diagnostic Processor module. The is also a table of standards that this product meets.

The GE Fanuc State Logic products are the result of a Strategic Alliance between GE Fanuc and Adatek, Inc. The purpose of this allicance is to bring Adatek's eleven years of experience applying State Logic principles to the Industrial Control Industry together with GE Fanuc's control hardware expertise.

Adatek, Inc.
700 Airport Way
Sandpoint, Idaho 83864

Phone: 208–263–1471
FAX: 208–263–6596

GE Fanuc Automation
North America, Inc.
P.O. Box 8106
Charlottesville, VA 22906
800–828–5747

ADATEK, INC.

SOFTWARE LICENSE AGREEMENT AND LIMITED PRODUCT WARRANTY

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. YOUR USE OF THIS PROGRAM PACKAGE INDICATES YOUR ACCEPTANCE OF THEM. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD NOT USE THIS PROGRAM BUT PROMPTLY RETURN THE PACKAGE.

Adatek, Inc. provides this program and licenses it for use. You assume responsibility for the selection of the Program to achieve your intended results and for the installation, use and results obtained from the Program.

License

Under this license you may:
Use the Program on a single computer at any given time. The program may be transferred to and used on another computer but shall under no circumstances be used on more than one computer at any time.

YOU MAY NOT:

4. TRANSFER OR RENT THE PROGRAM OR USE, COPY, MODIFY, OR MERGE THE PROGRAM, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PERMITTED IN THIS LICENSE.
5. DECOMPILE, REVERSE ASSEMBLE OR REVERSE ENGINEER THE PROGRAM.
6. REPRODUCE, DISTRIBUTE OR REVISE THE PROGRAM DOCUMENTATION.

IF YOU DO ANY OF THE FOREGOING, YOUR LICENSE AND THIS AGREEMENT ARE AUTOMATICALLY TERMINATED. SUCH TERMINATION SHALL NOT BE IN LIEU OF ANY CRIMINAL, CIVIL, OR OTHER REMEDIES AVAILABLE TO ADATEK, INC.

Limited Warranty

ADATEK, Inc. warrants that the Program will perform substantially as described in the accompanying User's Manual for a period of 90 days from the date of receipt. ADATEK, Inc. does not warrant that the functions contained in the Program will meet your requirements or that the operation of the Program will be uninterrupted or error free. ADATEK, Inc. warrants the disks on which the Program is furnished to be free of defects in materials and workmanship under normal use for a period of 60 days from the date of purchase.

Limitation of Liabilities

IN NO EVENT WILL ADATEK,INC.. BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM . ADATEK, Inc. ACCEPTS NO LIABILITY FOR ANY ACTIONS TAKEN BY THE USER BASED UPON INFORMATION GENERATED BY OR OBTAINED FROM THE PROGRAM.

We Welcome Your Comments and Suggestions

At GE Fanuc automation, we strive to produce quality technical documentation. After you have used this manual, please take a few moments to complete and return the Reader's Comment Card located on the next page.

Contents

Contents

Chapter 1 Getting Started

This chapter provides general information about the Diagnostic System. There is a general overview of the product, instructions on using this manual, installation procedures, serial cable information, hardware requirements, and sources of information about using UPTime.

Overview

The two parts of the Diagnostic System are the UPTime software and the Diagnostic Processor module.

UPTime stands for "Uptime Promotion Tool". UPTime is a complete environment (programming tool and on–line debugger) for creating and monitoring the Diagnostic Strategy. The purpose of the strategy is to reduce downtime in applications controlled by GE Fanuc Series 90–30 PLCs. This strategy is composed of fill–in–the–blank point level diagnostics and systems level State Logic programming using natural English terms, phrases and sentences. These programs mirror the controlled process step for step, checking the operations, collecting data, providing information about the process, and aiding preventive maintenance.

The Diagnostic Strategies created by UPTime are executed by the Series 90–30 Diagnostic Processor. This is a parallel processor module that plugs into the PLC backplane.

The Diagnostic Processor accesses the CPU register and I/O tables through the Series 90–30 backplane as the CPU controls the I/O through its normal program execution cycle. This is a parallel processing system, since the CPU executes a control program while the Diagnostic Processor is executing it Diagnostic Strategy. The Diagnostic Processor is also capable of any control operation turning ON outputs or setting analog and register values. If both the CPU and the Diagnostic Processor attempt to control the same data element, the CPU always takes precedence. For information about the Diagnostic Processor see the Diagnostic Processor User's Manual.

How to Use this Manual

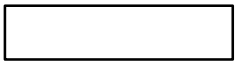
This product differs significantly from any other product on the market, therefore it is very important to read sections 2 and 3**, Diagnostic Strategy Overview** and **Creating a Diagnostic Strategy**.

After reading these sections, install UPTime in the computer and follow along with the Diagnostic Strategy Tutorial chapter. The tutorial demonstrates how to create and execute a Diagnostic Strategy and how the system features can be used to reduce process downtime.

The manual also has chapters that describe creating Diagnostics, State Logic Programming, the Diagnostic Processor Module, the UPTime menu system, and system specifications.

Notational Conventions:

- All text that should be entered at the keyboard are printed in **bold italics**.
- All references to individual keys are enclosed in angle brackets <>.
- Sample program lines to show examples but not necessarily entered into your computer are displayed in a box.



- Displays showing computer screens are all captures of actual UPTime displays with rounded corners on the surrounding box.
- References to menu options appear between double quote marks.
  "Make a New Project"

Brief Description of the Manual Sections

1. **Chapter 1. Getting Started**

   Getting Started is the section you are now reading. Getting Started tells you how to install UPTime on your DOS based computer and other particulars related to accessing information.

2. **Chapter 2. Diagnostic Strategy Overview**

   UPTime is an interface that allows you to tap into the substantial power and flexibility of State Logic programming plus providing simple to use Fill–in–the–Blank Diagnostics. This chapter provides some basics about the underlying concepts and philosophy of State Logic programming and Diagnostics. **Regardless of what you may already know about State Logic, it is extremely important that you read this chapter carefully.**

3. **Chapter 3. Creating a Diagnostic Strategy**

   This chapter explains how to create a Diagnostic Strategy using UPTime. A procedure for structuring State Logic programming and information about using Diagnostics are provided.

4. **Chapter 4. Tutorial**

   This chapter walks you through the creation and execution of a simple Diagnostic Strategy programmed in State Logic with UPTime. Some simple program sections are created then executed. Building on previous examples, a complete Diagnostic Strategy is created and executed.

5. **Chapter 5. Diagnostic Processor**

   This chapter explains the details of using the Diagnostic Processor. Topics discussed are installing and troubleshooting the module, serial communications, details of the scanning system, and interfacing with the CPU of the system.

6. **Chapter 6. The UPTime Menu System**

   This chapter describes some of the options available from the UPTime menu system. The options covered are those that are not used all the time and might be difficult to understand.

7. **Chapter 7. Fill–in–the–Blank Diagnostics**

   This chapter contains detailed information about creating and using Diagnostics. The four types of Diagnostics are described and the debug mode operations used to view and record Diagnostic actions are explained.

8. **Chapter 8. State Logic Programming**

   This chapter has all of the details about State Logic programming. The use of the State Logic language is discussed, including language structure and use of English words in the program, and the Perform Functions.

9. **Chapter 9. Specifications**

   This chapter has specifications for both the UPTime software package and the Diagnostic Processor module. The is also a table of standards that this product meets.

UPTime Hardware Requirements

1. IBM PC compatible or PS2
2. 640K RAM
3. DOS version 3.1 or higher; if DOS 5.0 or higher is used, then DOS must be loaded in the high memory area.
4. Hard Disk
5. 5.25 inch or 3.5 inch floppy disk drive
6. Serial Port
7. Any printer (Optional)
8. Color or monochrome monitor

Serial Cable

Communication between the UPTime host computer and the Diagnostic Processor is through the serial ports. The UPTime software is designed to use two different GE Fanuc serial cables. One cable is in the mini–converter kit that is included with UPTime. The other cable is the normal IBM PC cable, IC690CBL702, or Workmaster cable for a PS2 type computer, IC690CBL705.

Mini–Conver ter Kit

UPTime comes with a serial cable kit that is used to both connect the CPU to a computer running the Logicmaster software and to connect the Diagnostic processor with a computer running UPTime. This kit comes with 3 adapters described in the following table:

9–pin Male to 15–pin Male	CPU Port	HE693SNP232
9–pin Male to 25–pin Male	SLP Ports 1 or 2	AD232/1–2
9–pin Male to 25–pin Female	IBM PC COM 2	HE693XTADP

Mini Converter Kit Adapters

The cable that comes with this kit is terminated with two 9–pin female connectors. The 9–pin male to 15–pin male adapter connects this cable to the CPU to set the configuration with Logicmaster. The 9–pin male to 25–pin male adapter connects the cable to one of the Diagnostic Processor ports. The 9–pin male to 25–pin female adapter connects the cable to a 25–pin serial port on the IBM computer.

UPTime Installation

To install UPTime, insert disk 1 into drive A or B and make this drive the current logged drive. Type **INSTALL** and hit <Enter>. Choose the "INSTALL" option. Be ready to specify the hard drive where UPTime is to be installed. Follow the instructions for inserting other disks. The installation program displays a message when the installation is complete.

UPTime is copy protected so that only one installation is allowed per set of distribution disks. If an attempt is made to run UPTime without proper installation, a message is displayed saying that this is an unauthorized version of UPTime. The installation program does not allow a second installation from the distribution disks.

If there is a need to move UPTime to another computer use the UNINSTALL option of the installation program. This option removes UPTime from the computer on which it is installed. The distribution disks are modified to allow UPTime to be installed again on another computer.

To run UPTime, make sure that \UPTIME\S90–30 is the current directory by typing **CD\UPTIME\S90–30**and then press <Enter>. Now type **UPTIME** to start the program.

Getting Help

There are three ways to get help:

1. UPTime Help System. UPTime has a built in help system that can always be accessed by pressing the key on your keyboard marked <F1>. This help system is context sensitive, meaning that UPTime provides the helpful information you need based on the location of the cursor on the screen or the highlighted menu option at the moment you ask for help.
2. This manual provides detailed information for most of your needs. If you still need more information, feel free to call either of the numbers provided in the next paragraph.
3. GE Fanuc has personnel specially trained throughout the country to provide customer support for UPTime and other Adatek products that work together with GE Fanuc control products. Call GE Fanuc technical support line at (800) 828–5747. The Adatek technical support number is (800) 323–3343.

Chapter 2 Diagnostic Strategy Overview

The end product of the UPTime Create mode is a Diagnostic Strategy to reduce downtime of a process controlled by a Series 90–30 PLC. The programmer creates the Diagnostic Strategy using one or both of the two types of diagnostic aids: point level Fill–in–the–Blank Diagnostics and advanced systems level State Logic programming.

Fill-in-the-Blank Diagnostics

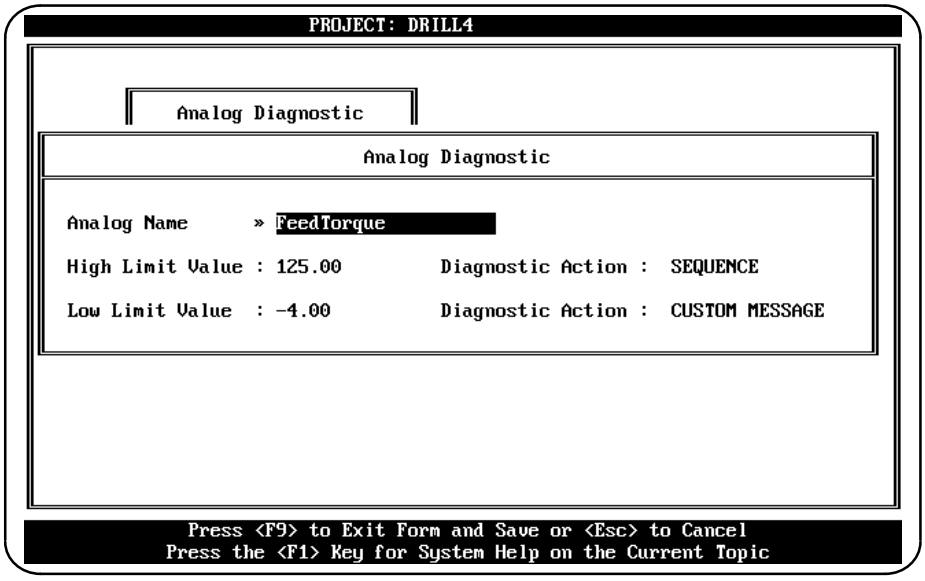
Diagnostics are simple point level alarms that monitor discrete and analog I/O and variable status. The Diagnostic initiates some action when that monitored data element causes the Diagnostic to be in an alarm state.

There are four Diagnostic types:

Discrete
Analog
Variable
Device

Most of the CPU memory locations can be monitored, %I, %Q, %M, %T, %G, %S, %SA, %SB, %SC, %AI, %AQ, and %R. In addition State Logic variables stored in the Diagnostic Processor can also be monitored.

Diagnostics are created by filling in a form such as the one displayed below.



Diagnostic Names

The first blank of each Diagnostic form is the blank for the name of the data element being monitored. Names are used throughout the Diagnostic Strategy in both Diagnostics and State Logic Sequences to refer to I/O and variable data stored in both the CPU and the Diagnostic Processor.

There are three ways to identify an I/O point or data location with a name:

Define the Name in UPTime (PartsCount)
Use the Memory Address Directly (%R34)
Use Nicknames Used in Logicmaster Ladder Logic Program (FST_SCN)

UPTime Names

UPTime displays a list of names defined for the type of Diagnostic being created, when the cursor is on the name blank and the <Ins> key is pressed. For example when creating an analog Diagnostic, UPTime displays a list of all analog I/O names that are currently defined if the <Ins> key is pressed when the cursor is in the name blank of the form.

A name can be up to 20 characters long and must start with a letter but may contain numbers and the underscore character " _ ". The user attaches a name to an I/O or data address through one of the UPTime name definition forms. For example the name, Foward_Limit_Switch, might be attached to %I34, using one of the several methods explained below for defining a name.

Since names must be one word, underscores and uppercase letters are used to simulate word breaks, making the text more readable. Tank_Level and BypassSwitch are examples of techniques for simulating word breaks in names.

Logicmaster Nicknames

Another way to identify I/O and register data is to use the same nicknames used in the Logicmaster 90 programming package to create the Relay Ladder Logic program. When UPTime needs the address for names used in the Strategy, one of the places it checks is a nickname file created by Logicmaster 90 to locate the I/O address. Therefore, if the nickname, REV_LS, has been attached to %I35 in Logicmaster, UPTime knows to use that same address location when that name is found in the Diagnostic Strategy. See the State Logic Programming chapter for detailed information on creating and locating the Logicmaster nickname file.

Direct Memory Reference

In addition to defining the name using UPTime, I/O and register data can be identified by just directly referring to the CPU memory address. Therefore, %I34 can be used directly without defining any name just by entering I34 in the blank for the name.

Triggering Value

Each type of Diagnostic has a value that causes the Diagnostic to be tripped. Digital Diagnostics can be set to trip when the point is ON, OFF, or just changes state. Analog and variable Diagnostics have both high and low limits that activate the Diagnostic. The Device Diagnostic specifies the maximum amount of time for the device to open and a maximum time to close. If the time is exceeded the Diagnostic is tripped.

Actions

Actions are the activity that the diagnostic initiates when tripped. There are three actions that a diagnostic can initiate:

Send a Default Message
Send a Custom Message
Start a State Logic Sequence

Default Message

The Default Message action causes the diagnostic processor to assemble a generic message that states the type of fault that has occurred, the name of the I/O point or variable involved, and its current value. This message is sent out the serial port designated the Diagnostic Port, when the Diagnostic is tripped.

Example Default Message:

Discrete Value Switched to ON : Name is DriveMotorOverload

There is a different Default Message for each type of Diagnostic. The text for each Default Message can be changed by options on the Define menu to any message up to 80 characters in length.

Custom Message

The Custom Message action causes a message to be sent out the Diagnostic Port when the Diagnostic is tripped. The text of this message is entered in the window provided when the Diagnostic form is saved. The message can be up to 128 characters in length.

Sequence

The Sequence action causes a Sequence to begin execution when the Diagnostic is tripped. Sequences are State Logic procedures that are used to do higher level analysis and operations.

The Device Structure

One of the four types of Diagnostics is the Device Diagnostic. A Device is a high level structure that treats several I/O points as a single entity.

There are many mechanisms in industrial applications that use an output\input I/O pair. The output starts an operation and the input indicates when the operation is complete. These mechanisms include motors, valves, presses, clamps, hydraulic rams, etc.

The Device structure groups two outputs and their associated input verifiers into one entity. With two output/input pairs, the Device structure can be used for the many mechanisms that involve a travel cycle such as a hydraulic ram opening and closing.

Creating a Device

A Device is defined by filling in the blanks in a form. The Device is given a name by filling in the name blank. The Device terminology is designed to describe an implement that opens and closes. The blanks for the two outputs are labeled Open Actuate and Close Actuate, and the blanks for the two inputs are labeled Open Complete and Close Complete.

Names must be defined before being used in the form. The same naming capabilities are used in the Device form as in the Diagnostic forms. A name is attached to an I/O point by using UPTime menus and forms and a list appears when the <Ins> key is pressed. The name blank may also be a direct memory address specification such as I76 or Q45, or a name may be a nickname used in Logicmaster when the Ladder Logic program controlling the process was created.

In the Device form there is a normally open or normally closed designation under the blank for each I/O name. For outputs this designation says "Actuate When On" or "Actuate When Off", and for the inputs the designation says "Complete When On" or "Complete When Off". The "Actuate When Off" designation is used for the many mechanisms that start the operation when the output is Off.

Although it is most common that outputs start the operation and inputs indicate when the operation is complete, it is not necessary to set up the Device in this manner. Any Digital point whether input or output can be used in any blank in the Device form.

The Device Diagnostic

The Device Diagnostic has a space for the Device name. The Device must be defined before the Diagnostic is created. A list of Devices is displayed when the <Ins> key is pressed and the cursor is in the name blank.

The Device Diagnostic has four different situations which trigger actions. Two of the conditions triggering actions monitor the time it takes for the Device to open or close. A timer starts when the close or open process is started by the specified I/O point that starts the operation. If the Device takes too long for the open or close operation, then the specified action is triggered.

The last two actions trigger when the Device either opens or closes as indicated by the I/O point that monitors for the completion of that operation.

State Logic

The State Logic part of the Diagnostic Strategy is the programming written for advanced systems level diagnostics, analysis, an instructions. UPTime exploits State Logic in three different ways.

First, State Logic is used to follow the process through its different stages, testing for likely malfunctions at each stage of the process.

The second use of State Logic is as an action specified for a Diagnostic. When the Diagnostic trips, the designated Sequence performs some higher level analysis of why the Diagnostic tripped or sends messages describing how to correct the problem.

The third use of State Logic is for Instructional Sequences that provide a method to display desired information to the operator. A menu is automatically generated that allows the operator to choose which Instructional Sequence to review.

State Logic Theory

State Logic has its roots in Finite State Machine Theory, developed by nineteenth century mathematicians. Because its philosophy is a natural fit to real–time systems, Finite State Machines have become the strategy of choice in disciplines, such as electronics and communications.

State Logic has been used for industrial control applications for many years. It has always been recognized that diagnostics are a particular strength of the State Logic, because the program inherently knows the current stage of the process and simply checks for malfunctions that are possible during that stage.

The Concept of Finite States

The basic concept of State Logic is that a process can be defined as a series of states or Steps. Each Step is defined by two components, **operations** that occur while that Step is active and the **transitions** to other Steps.

Operations describe actions such as turning ON digital outputs, setting variable and analog values, sending messages to an operator, or using a function.

"Make PartCount equal to 45."

is an example of an UPTime program line describing the **operation** part of a Step. A Step can perform an unlimited number of **operations.**

Transitions are a little more complicated since they are themselves defined by two components, the condition causing the **transition** and the target Step.

"If Part_In_Place switch in ON, go to Start_Conveyor Step."

is an UPTime program line representing a **transition** of a Step. The "IF" expression is the condition controlling this **transition** and Start_Conveyor is the target Step. Other conditions that can be tested are the status of the digital inputs, the values of variables and analog inputs, elapsed time, etc. The target Step is the one which becomes active when the condition is true.

A series of Steps in Finite State Machine science is called a State Machine. UPTime calls each such group of Steps a TASK.

It is traditional to diagram Finite State Machines with circles and arrows. The **actions** of a Step are written inside the circles. The arrows show the **transitions** with the condition component of the transition written next to the arrow. The following unlabeled State diagrams show two simple Finite State Machines or Tasks.

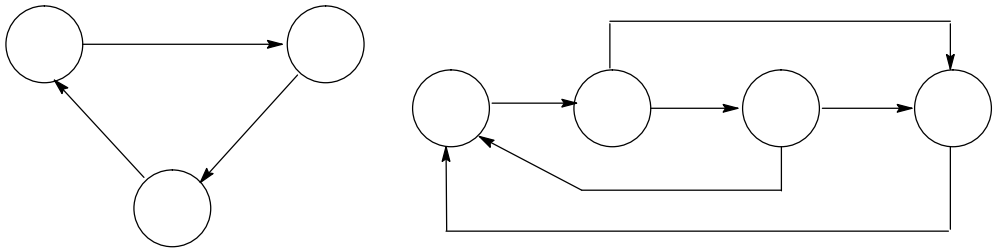


Figure 2-1. State Diagrams

The Task may transition from one Step to any other Step in the Task depending on how the instructions are specified by the system designer. There may be several transitions described in one Step. Each Task is always in one and only one Step at any time. The transfer from one Step to another does not consume any time, therefore outputs that are ON in two successive Steps do not blink OFF during the transition.

Project: CHEMICAL PROCESS

Task: Make_Compound_5
Step: PowerUp
If the Manual_Switch is on and Start_Pushbutton
is pressed go to the Adding_Water Step.
Go to Adding_Water if Auto_Switch is on.

Step: Adding_Water
When Tank_Guage equals 35 gallons,
then go to the Adding_Chemicals Step.

Step: Adding_Chemicals
When the Chemical_Management Task is in the Emptying Step,
go to Check_Pump_2 Step.
When Tank_Guage equals 39 gallons,
send “Tank Filled” to operator_panel and go to the Mixing Step.

Step: Mixing
If hour is past 8 AM, Start_Sequence the exhaust_system.
If 20 seconds pass and the Mixer_Monitor
is less_than 100 rpms, go to the Wait_3 Step.
Go to the Cooking Step when Oven_Element is ON.

Step: Wait_3
Write “PROCESS SHUT DOWN BECAUSE MATERIAL IS TOO THICK”.
Go to PowerUp Step when Reset_Button is pushed.

Task Description in UPTime

An important point is that Finite State Theory does not create or invent TASKS. TASKS are already an inherent part of every process. Programming with a state language is merely the act of describing the sub–systems of the process.

Processes have more than one sub–system so that State Logic programs usually have more than one Task. Each Task executes concurrently or in parallel.

Developing State Logic Programs with UPTime

Developing State Logic programs can be characterized as entering a description of your control system into a **template.** The template is the Finite Step Machine model built into the Diagnostic Processor. UPTime is a tool and framework for entering that description into the template.

State Logic Hierarchy

The State Logic organization has the following hierarchy of structures:

PROGRAM
SEQUENCE
TASK
STEP
STATEMENT
EXPRESSION

State Logic Hierarchy

Each State Logic program is composed of one or more Sequences, which are themselves constituted of one or more Tasks. This same relationship carries throughout the hierarchy except that a Step can be defined with no Statements.

TASK: Drill **TASK NAME**
Step: Drill_Advancing **STEP NAME**
After 3 seconds go to Starting_Drill_Motor Step. **STATEMENT**
When Fwd_Limit_Switch is tripped go to Retracting Step.
Go to Send_TimeOut_Message if 17 seconds pass.

Step: Retracting
When Home_Switch is tripped go to the Increment_Counter Step.
If 15 seconds have passed, write "TOO Long Retracting" to Operator.

Step: Increment_Counter
Add 1 to Parts_Count.
Write "Parts Count equals %Parts_Count" to operator_display.
If Parts_Count is less than 24 go to the PowerUp Step.
If Parts_Count is 24 go to the Send_Message_2 Step.

Step: Send_Message_1
Write "REPLACE DULL DRILL BIT" to message_board, go to Retracting Step.

Step: Send_Message_2
Send "RUN COMPLETED" to operator_display, go to New_Cycle Step.

TASK: Setup_DISPLAY

Step: Operator_Panel

Sample Task with Some Elements Labeled

Sequences

There are three types of Sequences. Continuous Sequences start execution at powerup and are used to follow the process through its different states of operation. Triggered Sequences are a specified action of a Diagnostic and start operation when that Diagnostic is tripped.

Instructional Sequences are used to provide information to the operator through a dumb terminal or using the UPTime Debug mode. The Diagnostic Processor displays a menu that lists all of the Instructional Sequences when "MENU?" is typed at the keyboard. When one of these Sequences is selected it begins operation. Instructional Sequences can also be started from a program Statement in another Sequence.

When a Sequence is started from another Sequence all of the Tasks start executing from the first Step defined following the Task name. If a Task is in the middle of an operation when started, it stops what it is doing and restarts at the beginning Step.

Tasks – A Collection of Steps

By design, a machine or process is a collection of Tasks that operate concurrently. A car engine has an electrical system, a fuel system, a mechanical motion system, cooling system, exhaust system and a starting system that, while independent in action, must be coordinated in time for the engine to work. Similarly all industrial processes, machines and systems contain several Tasks that are mutually exclusive in activity yet joined in time.

Tasks execute concurrently, and every Task always has one and only one Step active at any point in time. Only the instructions of the active Steps are executed, all other logic is ignored.

Steps – The Building Blocks of a Task

Task: Mix_Station

Step: PowerUp
If Can_At_Mix is on, write "Mixing Can" and go to Lowering_Mixer.

Step: Lowering_Mixer
When the Mixer_Down_Switch is tripped,
then go to Mixing_Chemicals Step.
If 23 seconds have passed, go to the Lowering_Hang_Up Step.

Step: Mixing_Chemicals
If Mixer_Overload is tripped, go to Mixer_Overload Step.
When Mixer_Up_Motor is ON, go to Raising_Mixer.

Step: Raising_Mixer
If Mixer_Up_Motor is OFF, go to the Raising_Problem Step.
If Mixer_Up_Switch is tripped,
then go to Mix_Complete.

Step: Mix_Complete
When Can_At_Mix is off, add 1 to Batch_Count and go to PowerUp.

Five Step Task Example with a Single Step Highlighted

In the automobile engine example we said an engine is viewed as a collection of Tasks; Fuel System Task, Electrical System Task, Starting System Task and so on. Each of those Tasks is further described as a precise set of Steps through which that Task will pass while the engine operates.

The automobile engine's Starting System Task has several possible Steps. For example we know there is a Step in which the key is on, the engine is not running and the starter motor is not cranking the engine over. We know there must be another Step in which the key is in another position, the engine is not yet running but the starter motor is cranking the engine over. There are also Steps in which the key is on, the engine is running and the starter motor is no longer cranking the engine.

A Step can set outputs, send messages, perform calculations, and assign values to data variables. Steps also describe transitions to other Steps. Only one Step is active and executed in a Task at any time. If two Steps need to be active at the same time then a concurrent Task is required.

Every Task must have at least one Step. When the Task starts execution, the Task goes to the first Step following the Task name, which is the first Step in the execution sequence of the Task. Thereafter, activity can move to any other Step based on the Statements in the active Step.

Statement – The Command Set for Step Descriptions

The actions of a Step are described with a Statement or a collection of Statements. In UPTime a Statement is a collection of Terms describing the desired actions for that Step. Statements end with a period (.) and can be thought of as sentences, although punctuation and proper grammar are not required.

```
Step: Raising_Mixer
Write "Mixer Moving" to Operator_Panel.
When the Mixer_Up_Switch is tripped, then go to Mix_Complete.
```

Example, UPTime Step with One complete Statement Highlighted

In UPTime, Statements are normal English words, phrases or sentences. In the automobile engine Starting System example, we might find that to make a complete description of the Starting System Task, the activities would have to be described in great detail. A complete range of Statements is available that describes the full and possible ranges of activity of a Step. There can be an unlimited number of Statements in a Step.

Expressions

There are two types of Expressions used in a Statement; Functional and Conditional.

Functional Expressions perform a specific action, including enabling or disabling Diagnostics, starting another Sequence, turning on digital outputs, setting analog outputs to values, performing calculations, setting variables to values, transferring to another Step or communicating with other devices.

Conditional Expressions perform some decision making test which enables or prevents execution of the functional Expression in the Statement. The conditions that can be checked include digital point status, analog values, a read from a serial port, or status of any system variable, including the current Step activity from other Tasks.

Functional and Conditional Terms are listed below using typical UPTime terminology.

Functional Terms	Conditional Terms
Actuate, Start, Turn on	If, when
Go	Read, get
Add, Subtract, Divide, etc.	
Make, Set	
Write	
Enable_Diagnostic	
Disable_Diagnostic	
Start_Sequence	
Start_PID, Stop_PID	
Suspend_Task, Resume_Task	
Perform	

Scan Overview

In executing the diagnostic strategy, the Diagnostic Processor continuously cycles through the steps of scanning the I/O, then scanning the control program, and finally setting the outputs. These scans occur many times each second.

During the scan of the program, each instruction of the active Step of each Task is executed once. Each Statement of a Step is executed in order from the first Statement to the last unless a GO Term is encountered. As soon as a Go is executed, no more Terms in this Step are scanned, and the scan moves to the active Step of the next Task.

When a Statement is executed, all conditional Terms are evaluated before implementing the action described by the functional Terms. If any conditional Term is not satisfied or false, the execution of this Statement is stopped, the functional Terms are not implemented, and the scan resumes at the next Statement of the Step.

The Diagnostic Processor keeps a table of all digital outputs and flags which are set ON during the program scan. Only the outputs set ON by one of the functional Terms in one of the active Steps during the scan are set ON, all others are OFF. The real world outputs and flags are set ON at the end of the scan. Therefore, an output does not go OFF during the transition from one active Step to the next when that output is set ON in both Steps.

This scan discussion is a general overview of the program and I/O parts of the scan. The State Logic Programming chapter of this manual has a more detailed discussion of the Diagnostic Processor scan procedure.

Communication Functions

The main function of the Diagnostic System is to provide information about the controlled process and how to make adjustments and correct problems. The channel for these communications is through the two serial ports. UPTime features simple methods for sending and receiving information through the serial ports.

The first section of this chapter on Fill–in–the–Blank Diagnostics explained how to use the Default and Custom Message actions. State Logic also provides simple ways to send and receive serial information, the READ and WRITE terms.

The Write Term allows characters to be written to any of the serial ports in the Diagnostic Processor. These can be connected to operator interface terminals or smart panels to present full screen displays or simple messages. The serial port might also be connected to the UPTime host computer, which displays the programmed messages in the online mode UPTime display.

The Read Term is used to bring serial information into the program. When a Read Term is encountered in the execution of a Step, it is treated as a Conditional Term that isn't satisfied until a completed message is received from the serial port. A variable follows the Read Term in the State Logic program. The variable can be any type, string or numeric. That variable stores the data received in the serial port.

The Read Term can be used to communicate to any serial input device. This would include operator interface devices such as terminals, smart panels, and personal computers and also the UPTime Debug Mode. It would also include intelligent sensors such as weigh scales, and the various smart pressure and flow transmitters now sold by various manufactures.

Together the Read and Write Terms make communicating with the operator very powerful yet simple. It also makes it easy to communicate with intelligent sensors, controllers and other machines that populate the plant or factory.

```
Step: Wait_For_Command
   Read Start_Command from Operator_Panel, then go to
   the Start_Process Step.
   If 20 seconds pass go to the Operator_Prompt Step.

Step: Operator_Prompt
    Write "PLEASE SELECT BATCH AND START PROCESS" to
    the Operator_Panel, then go to Wait_for_Command.

Step: Problem_Report
    Write "PROCESS SHUT DOWN BECAUSE MATERIAL IS TOO THICK".
    Go to PowerUp Step when Reset_Button is pushed.

Step: Start_Process
```

Highlighted Communication Functions

Any one of the serial ports may also be set up for the CCM2 communications protocol. Using this protocol enables the Diagnostic Processor to be a slave on a CCM2 network. Typically the protocol is used to communicate with host computers running custom software or, more commonly, graphical user interface software such as CIMPLICITY.

Control Capabilities

The Diagnostic System is designed to reduce process downtimes so its functions are designed to analyze process malfunctions and provide helpful information to the operator. There are times that some control functions are needed such as flashing an alarm light or setting a bit in the CPU. In fact, the Diagnostic Processor does have a complete set of control instructions so that the complete process could actually be controlled by an UPTime program, although control is not the primary goal of this product.

Output OFF By Default

One of the features of control in State Logic is that the outputs are always OFF by default. Therefore, an output that is ON in one Step, goes OFF after a Step transition if the next Step does not also turn that output ON.

There are several advantages to this design. The programmer need not include any logic for turning OFF outputs. He also does not need to decide which outputs need to be turned OFF depending on which outputs have been turned ON in the past and which logic path has been followed to the current stage of the process.

When analyzing a running system, it is not necessary to know the history of activities to know which outputs should be ON. Merely inspecting the currently active Steps give a complete list of all outputs that should presently be ON. If an output is not specified to be ON in any of the active Steps, then it will be OFF.

Diagnostic Processor to CPU Communications

The Diagnostic Processor communicates with the CPU across the PLC backplane, reading and writing to CPU memory locations. The State Logic program sets an output ON when a Statement such as

Turn ON AlarmLight

is executed. The output name AlarmLight is attached to a %Q address that is connected to the light.

In addition to %Q memory locations, many other discrete memory types can be changed by the Diagnostic Processor, %M, %T, %G, and %I. Register values can also be changed. The register types are %AI, %AQ, and %R. The %R registers can be defined to be either integer or floating point values. Floating point register variables use 2 register locations.

An important consideration when controlling CPU data with the Diagnostic Processor is that the CPU always takes precedence should the CPU program and the Diagnostic Processor attempt to control the same memory location. For example, if both processors have %Q34 in their programs, the CPU program always controls %Q34 and any Diagnostic Processor attempt to change %Q34's value is ignored.

When a name is assigned to a memory location, that name is designated as either an INPUT or an OUTPUT. If the Diagnostic Processor is to control that memory location, turning on a bit or setting a register value, the name must be designated to be an OUTPUT. Most names are designated to be INPUTs, since most of the time the Diagnostic Processor is only checking the status of CPU data to follow the process being controlled by the CPU.

Chapter 3 Creating A Diagnostic Strategy

This chapter presents the fundamental concepts of how to build a Diagnostic Strategy with UPTime. Every designer develops his own style in using UPTime, in fact, UPTime is designed to support and even to encourage personal or corporate program development styles. Initially however, it is suggested that the following procedure be followed in creating your first Diagnostic Strategy with UPTime.

Overview

The Diagnostic Strategy is composed of one or both of the two main structures, Fill–in–the–Blank Diagnostics and State Logic programming. This chapter explains features and benefits of each structure and how and when to use them to reduce system downtime.

The Diagnostic Strategy should be thought of as a dynamic entity that evolves as more is learned about the process being monitored. The Strategy should start out fairly simple and then be expanded as it is learned which malfunctions are likely to occur and the need for more information becomes apparent. Since a useful Diagnostic Strategy can be created in a matter of hours using only Fill–in–the–Blank Diagnostics, many systems start out with no programming.

Creating Diagnostics

The first step in creating the Diagnostic Strategy is to decide which I/O points, variables, and Devices need to be monitored by Fill–in–the–Blank Diagnostics. Names for all of these items are defined before the Diagnostics are created. Diagnostics should be created for items that by themselves indicate a fault in the system or indicate a system event of which the operator should be informed.

All that is required to create diagnostics is to fill in the provided forms. See the Overview chapter for a detailed description of the forms.

The purpose of Fill–in–the–Blank Diagnostics is for simple point level response to system events. These Diagnostics monitor I/O points and variable data and initiate a specified activity when the monitored item reaches a specified value. A digital Diagnostic can be configured to trigger an action if the monitored point is either ON, OFF, or changes state. Analog and variable Diagnostics have both high and low limits that trigger the Diagnostic if the data value is outside of those limits. A Device Diagnostic checks the time that the Device takes to open or close.

Diagnostics Only Strategy

For some applications all that is needed in the Diagnostic Strategy is a set of Diagnostics and no programming. Users often use only Diagnostics for the first Diagnostic Strategies put into service. Programming is easily added later to handle more complicated situations.

Diagnostics are enabled or active by default. Only State Logic programming can disable diagnostics, therefore the Diagnostics in a strategy with no programming are always active. A Strategy that uses only Fill–in–the–Blank Diagnostics can be viewed as several alarms that are always armed, ready to send a message when tripped.

Diagnostics vs. State Logic

State Logic can perform many of the same functions accomplished by Fill-in-the-Blank Diagnostics. The State Logic Statement,

If OverspeedSwitch is ON, write "Overspeed switch tripped! Push Reset".

does the same functions that a Diagnostic would that monitors the Overspeed Switch.

UPTime Debug Mode provides several helpful online tools using the Diagnostics. The definition and current alarm status of each Diagnostic is available through the menu options. Also available is a list of Diagnostics that are currently in an alarm state along with the values that caused the alarm and the time that the alarm was tripped. One other feature of the Debug Mode is that a history of Diagnostics that have been in the alarm state is available showing when each alarm was set and when it was cleared.

Because of these online features, it is better to use Diagnostics than to try to accomplish the same functions in State Logic Programming. In addition, Diagnostics execute a little faster than State Logic program instructions, about 10% on average.

State Logic is used for much more complicated analysis than is possible with Diagnostics alone. Try to set up Strategies to take advantage of both the online advantages of Diagnostics and the powerful analysis tools of State Logic. Set up Diagnostics to monitor digital points or variables controlled by the State Logic program. Therefore, the program can perform some complicated analysis, then trip a Diagnostic so that the online functions provide important information about the event.

Creating State Logic Sequences

As explained in the Overview chapter, there are three ways that State Logic is used in the Diagnostic Strategy. Therefore, there are three different types of Sequences; Continuous, Triggered, and Instructional.

One use of State Logic is to follow the process through its different stages of operations, checking for the malfunctions that are possible during each stage. The type of Sequence used for this purpose is the Continuous Sequence. As the name indicates these Sequences execute their logic continuously from power up to power off.

The second use of State Logic is to perform some operation when a Diagnostic is triggered. These Sequences normally do some analysis of the problem by checking I/O and variable values and logically deducing the cause of the problem. Sequences that perform this function are called Triggered Sequences. These Sequences do not start execution until the associated Diagnostic is tripped and do not end execution until they execute the End_Sequence keyword.

The third use of State Logic is to provide instruction routines displaying information needed to correct a problem, perform some operation, or make some equipment adjustments. The type of Sequence that is used for this purpose is the Instructional Sequence. If the operator types <MENU?> at the keyboard connected to the serial port designated the diagnostic port, then a menu listing all of the Instructional Sequences is displayed. An Instructional Sequence only starts execution when it is selected from this menu or when a command from another Sequence starts it.

All of the Tasks in a Sequence start and end execution at the same time. All Tasks start in their beginning Step at the same time. Therefore, whether a Sequence starts as the result of the keyword instruction, Start_Sequence, executed by another Sequence, or because it is initiated by a Diagnostic, all Tasks in that Sequence start simultaneously.

A Sequence stops execution when the keyword instruction, Stop_Sequence, is executed by another Task or when the keyword End_Sequence is executed within the Sequence. In both cases all Tasks in the Sequence stop execution at the same time. When a Sequence is stopped all Tasks are put into the Inactive Step. This is a Step that has no Statements and that all Tasks have by default when the are created. All Triggered and Instructional Sequences must have an End_Sequence instruction in at least one of the Steps of one of the Tasks in the Sequence.

Creating Sequences to Follow System Operations (Continuous Sequences)

One way that State Logic is used to reduce downtime is for more involved systems level analysis of malfunctions and detection of imminent breakdowns. The program is created to follow the process through its different stages of operation checking for likely malfunctions by enabling and disabling diagnostics or directly checking values with programming at each stage. When a problem is detected, State Logic is then used to deduce the cause, by investigating the values of the I/O and variables plus the current Step of the Tasks of the process.

To create a Continuous Sequence, select Sequences from the Create menu and then Diagnostic Sequences from the subsequent menu. Continuous Sequences are one of two types of Diagnostic Sequences. All Diagnostic Sequences start out as Continuous Sequences and then become Triggered Sequences when the Sequence is named as the action of one of the Diagnostics.

Outline the Application

To follow the process in State Logic use a top down design strategy where the components of the main problem are identified at the top level and then each of these components is broken down into its separate parts. This decomposition of the problem continues until the application is completely described. The State Logic model invites top down design because of the hierarchy of its elements, Sequences, Tasks, States, and Statements as described in the Overview chapter. There are several different formats to aid in the top down design process including structure charts and structured flow charts, but we use a simple outline approach.

Identify the Sequences

The goal of this step is to identify the Sequences and Tasks that follow the application. We start at the highest level, decomposing the problem into its general components. See the discussion on Sequences and Tasks in the previous chapter.

Sequences and Tasks are used to describe the systems and sub–systems of the application. Tasks all execute in parallel or concurrently. Sequences are a collection of Tasks and generally represent higher level systems than Tasks. For example, each station on a transfer line might be represented by a Sequence while independent systems or parallel operations at a station would be represented by Tasks within a Sequence.

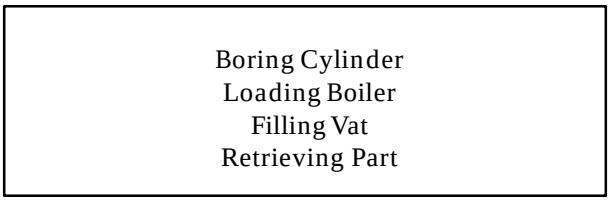
The separation of the process into Sequences and Tasks is quite arbitrary. Often Strategies have only one Continuous Sequence with several Tasks describing all of the systems of the process. There are commands though that start and stop Sequence execution meaning that all Tasks within that Sequence start an stop at the same time. Because of these commands, any group of Tasks that must be started and stopped together should be grouped together in a Sequence.

Identify the Tasks

Think of the independent operations which must be accomplished to achieve goals of the application. The natural separations of activity often become Tasks.

The goal is to decompose the problem into parts that can be defined as a series of I/O operations. Any cycles which repeat even with some variations are prime candidates to be Tasks. An important concept for identifying Tasks is that Tasks are a set of sequential operations. Events which occur in parallel or concurrently should be in separate Tasks.

These main sections of the outline should be general descriptive phrases such as:



The goal is to just follow the application. Most of the independent Tasks are quite obvious, others which require interaction with other Tasks are more difficult to identify at first. This is usually a repetitive process where original efforts must be adjusted as the outline progresses. As with most activities, proficiency increases with the number of efforts.

Identify The Steps

Once the Tasks are determined, then the Steps of each Task should be identified. The Steps use the condition of the inputs and outputs to know when the process is going to another Step. The Steps form the sequence and are really a picture of how this piece of the process (Task) should behave. See the discussion of Steps in the Overview chapter of this manual.

At this point in the design stage the goal is to determine that the process can be followed correctly with the chosen Task architecture. Simply give each Step a descriptive name fitting the major attribute of the activity that takes place when that Step becomes active. Typical Step names are:

Send Message
Add Water
Raise Drill
Start Motor

Step names identify the general action of the State. The specific actions and the transitions are specified in the Statements.

Identify the Step Transition Statements

The goal at this stage of development is to program the changes from Step to Step. Generally it is the outputs of the system which indicate that the process has change to a new Step. Once the transitions are in place the program is a complete description of the process detailing the sequence of output activity.

This part of the programming process is critical, because this description is the foundation for entering diagnostic functions. If the proper foundation is achieved, the adding of diagnostic functions becomes just a matter of identifying the possible faults during each Step of the process. In addition one Task may check the active Step of another to deduce the cause of a fault. Often this stage of programming requires a change to the Steps and even the Tasks already identified, as the description of the controlled process is refined.

Typical Statements for Step transitions are:

If ForwardSolenoid is ON, go to the MovingForward Step.
If the WinchMotor is ON, go to the Pulling Step.

Identify the Diagnostic Statements

After the foundation of Tasks, Steps, and transitions are in place, the Statements providing diagnostic functions are added in Steps where there is a potential problem that must be monitored.

Diagnostic Statements can check digital conditions, variable and analog values, elapsed time, and the current Step of another Task. These checks are used to determine if a fault has occurred. Examples of diagnostic checks are:

If ForwardSwitch is ON, then . . .
If 3.5 seconds have passed, then . . .
If the Drill Task is in the Advancing Step, then . . .
If PartsCount is greater than PartsGoal, then . . .
If VatTemperature = 34.5 degrees Centigrade, then . . .
If InPosition is ON or Position < 34.5 and ManualSwitch is ON, then . . .

There are several actions that diagnostic Statements can accomplish. A message can be sent to one of the serial ports notifying the operator of a problem and detailing a course of action. Data can be tabulated such as keeping track of the number of times an event occurs. Another Task can be started to perform some analysis of the cause of a problem. Outputs can be controlled to signal the control program or to turn ON an alarm signal.

```
Start_Sequence OverloadAnalysis.
Write "Operation complete" to Operator.
Add 1 to PartsCount.
Turn ON AlarmLight.
```

Statements are often complete English sentences, since very specific operations are specified at this level of the outline. In fact, feel free to specify Statements in any comfortable format, but remember Statements always end with a period. Some additional examples combine the Step actions with the transitions:

```
If TankHeater is ON,
write "Tank Heater stuck ON! Open breaker 3 then free switch".
When 5 seconds have elapsed, then go to RaisingMixer Step.
Write "Drill Bit is Dull" to operator, then go to RetractingDrill Step.
Read Command from Operator, then go to Report Step.
```

Program Scan

In the creation of Sequences it is important to understand that the Diagnostic Processor operating system is a scanning system. The Statements of the active Steps are executed once every scan cycle. The scan cycle starts at the start of the program, scanning the active Step of every Task. During program execution there is always one and only one Step active in each Task. The operating system completes a scan of the program many times every second.

During the scan of the active Step of a Task, each Statement of the Step is scanned in the order that it appears. Keep in mind that a Statement is a series of Terms terminated by a period (.).

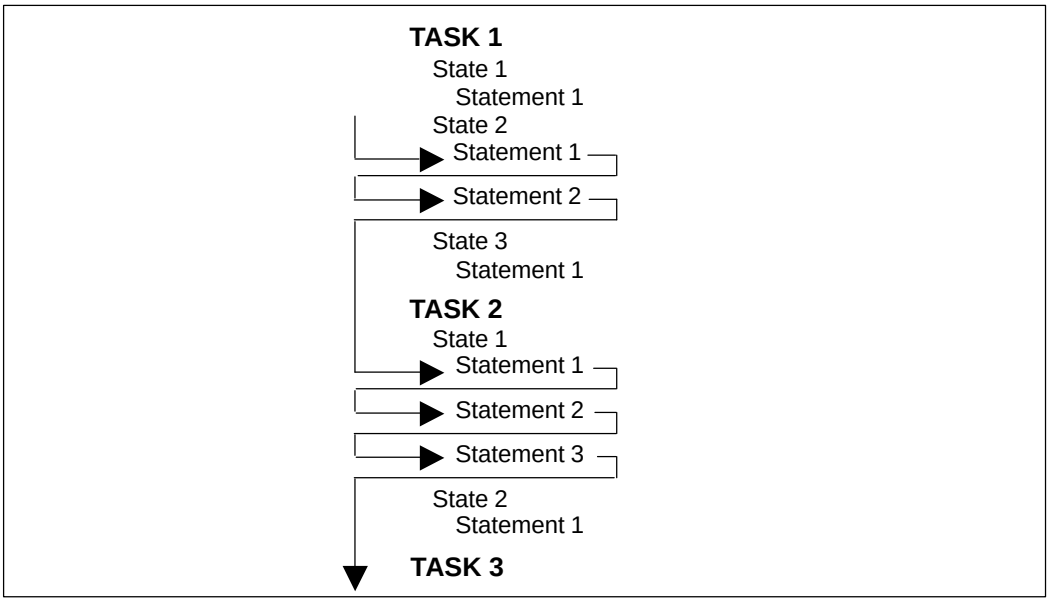


Figure 3-1. Program Scan

The actions specified by Functional Terms are executed when the Functional Term is scanned. Each Statement must have at least one Functional Term, Conditional Terms are optional. If there are no Conditional Terms in a Statement, the Functional Terms are always executed during each scan. When Conditional Terms accompany Functional Terms in a Statement, the Functional Term is executed when all of the Conditional Terms are satisfied. There are two types of conditional Terms (see the State Logic Programming chapter).

Conditional Terms are satisfied as follows:

1. Read – When valid data is received at the appropriate channel.
2. If – When the conditional expression is TRUE.

To understand how Statements are scanned, assume that the Statement Conditional Terms precede the Functional Terms and that the scan proceeds from left to right.

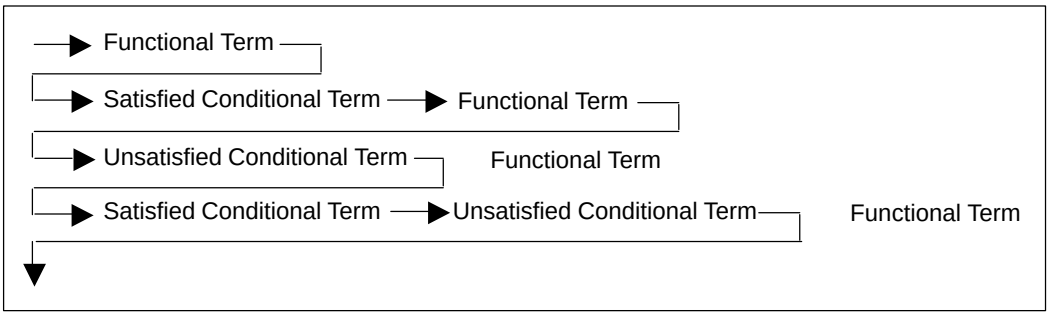


Figure 3-2. Statement Scan

The Statements of a Step are executed in the order that they are written into the program. Functional Terms of Statements with no Conditional Terms are always executed. Conditional Terms in Statements control whether or not the Functional Terms in those Statements are executed. If all of the Conditional Terms are satisfied, the Functional Terms are executed. If any of the Conditional Terms are not satisfied and Conditional Terms, the Functional Terms are not executed. For simplicity this rule assumes that the Conditional Terms are ANDed together. See the State Logic Programming chapter about combining Conditional Terms using the AND and OR logical keywords.

The Statements are executed one at a time. In this manner every Statement of the active Step is evaluated.

There are two types of Functional Terms that can prevent the execution of the rest of the Statements in a Step. One is the Halt command which stops program execution. The other is the "go to ..." command, which immediately causes another Step to become the active Step. No Terms in a Step are scanned after a GO is scanned in that Step.

If Start_Pushbutton is pushed, go to Start_Up Step.
Write "Press Start Push Button Now!!".
If 30 seconds have passed, go to the Restart_Buzzer Step.

This series of Statements causes the Start_Up Step to become the active Step when the input represented by Start_Pushbutton name is true. When GO Term is scanned, all Terms or Statements following this Term are not executed and at the next controller cycle, the scan of this Task starts at the first Statement of the Start_Up Step.

During the program scan any changes to variables are made immediately. Therefore, a variable change in one Task is visible by the rest of the program during the same scan. On the other hand, digital I/O and Flag and analog values are made at the end of the scan. Therefore, if one Task makes a change to the condition of a digital output or Flag, the condition cannot be tested by another Task until the next scan through the program.

Writing The Program

When the outline is completed, the UPTime program is almost completely written. The finished program is very close to the outline.

There may be some changes to the outline because of some naming conventions for how Task, Step, and some other names are entered into the program. UPTime can not provide for the full expressiveness of the English language so some of the sentence constructions may have to be changed, although many alternative structures and the ability to make custom changes to UPTime are provided. Also, the outline is in a general format with no specific reference to the actual I/O of the system so that the wording of the outline usually becomes more specific in the program.

To write the program the Tasks, Steps, and Statements of the outline are entered into the project using the UPTime editor which is available whenever UPTime is in Create Mode. Another part of creating the program is specifying I/O names and circuit configurations. Defining the I/O may be done before, after, or during the writing of the program.

Using English Names in the UPTime Program

When you start a new program, UPTime asks for the name of the first Task. After the name is entered, UPTime starts the program for you by writing the Task keyword followed by a colon and the Task name. UPTime also writes the first Step name, "PowerUp" into the program. Tasks, Steps, I/O points and variables can all be assigned English names. Names can be as brief and code like or as descriptive as you wish.

Clever, descriptive names that fit well to the primary attribute of that Step activity are strongly encouraged. This will pay dividends in future program modifying, clear documentation and easier troubleshooting.

Further, good descriptive names will enhance the quality of the automatic diagnostics that can be created by linking Task, Step and I/O names together for automatic diagnostic output information.

Each name can have up to a twenty characters. These characters may be letters, numbers, or the underscore character (_). Names must begin with a letter. The name must be a continuous string of characters, ie., no spaces are allowed.

Because UPTime uses the space character as a way to tell where one word ends and the next begins, as we normally do in written English, a name can not contain a space. To construct a multiple word name for descriptive purposes the designer should use the underscore character (_) to separate words or use uppercase to start every new word.

```
Table_Movement
TableMovement
```

Naming Sequences

Sequence names are arbitrary. It is suggested that Sequence names be descriptive of the activity they represent. This descriptive use of names means clearer documentation and the ability to create automatic diagnostic output messages by combining Task, Step and I/O names to make complete messages.

Naming Tasks

Many Sequences have only one Task, therefore UPTime automatically names the first Task of a Sequence the same name as the Sequence except that an underscore (_) is added to the end of the Sequence name.

A Task may be added to the program by using the "Add a New Task" option from the Add Menu or just typing in the Task keyword followed by a colon and the Task name. Each Task is assigned a name as it is built and each Task must have a unique name. This name appears at the beginning of every Task. Every time the designer wants to refer to the Task using UPTime such as in writing other Tasks, during debugging or during diagnostics development, the English Task name should be used.

Naming Steps

Each Task contains one or more Steps. Similar to Tasks, a name is assigned to every Step of the program either through the Add menu or directly into the program using the UPTime editor. Once assigned, these names are used when performing any functions associated with Steps while using UPTime.

While every Task in a controller must have a unique name to differentiate it from the others, Steps in different Tasks may have the same name. All Steps within one common Task must have a unique name, but a Step in one Task can be named the same as one in a different Task.

As with Tasks, names chosen for the Steps should be descriptive. By using combinations of words that describe something unique to the Step, such as the action performed or function of the Step, the program becomes self documenting. Using descriptive names makes it possible for people other than the original designer to use and modify a sequence at a later date with minimum learning time spent trying to understand the program.

Naming I/O Circuits, Variables, and Internal Flags

Each input and output from and to the field enters and leaves the controller through some particular hardware card. A name is given to each of these I/O points. All references to I/O circuits use the assigned name.

Names are also used for variables and internal flags. All names must be unique. For example, a variable must not have the same name as a Step or a flag must not use the name of a Task. The only exception to this rule is that Steps in different Tasks may use duplicate names.

The English name should be descriptive and can be made up of several words attached by the underscore character. UPTime allows the user to define I/O points or other name other elements of the program at any time during the programming process.

In addition to defining the name using UPTime, I/O and register data can be identified by just directly referring to the CPU memory address. Therefore, %I34 can be used directly without defining any name just by entering I34 in the blank for the name.

Another way to identify I/O and register data is to use the same nicknames used in the Logicmaster 90 programming package to create the Relay Ladder Logic program. When UPTime needs the address for names used in the Strategy, one of the places it checks is a nickname file created by Logicmaster 90 to locate the I/O address. Therefore, if the nickname, REV_LS, has been attached to %I35 in Logicmaster, UPTime knows to use that same address location when that name is found in the Diagnostic Strategy. See the State Logic Programming chapter for detailed information on creating and locating the Logicmaster nickname file.

Statement Structures

The greatest difference between the outline and the UPTime program is in the expression of the Statements. This section describes how to express the Statements in an UPTime program.

Step: Drill_Advancing

If the Fwd_Solenoid is on and 3 seconds pass, go to StartingMotor Step.
When Fwd_Limit_Switch is tripped go to the Retracting Step.
If 17 seconds pass, go to the Send_Message Step.

Example UPTime Step with Three Statements

Most Steps consist of several Statements, some that monitor I/O and variable values, checking for possible malfunctions that are likely when this Step is active. Other Statements describe what action is to happen while the Task is in that Step, what conditions will cause a transfer, and to what Step the Task transfers to. With UPTime these Statements are written in descriptive English generally but not necessarily consistent with the rules of English grammar. Statements are short sentences or phrases that describe the desired actions in a way that anyone can read and understand. A Statement always ends with a period just as a sentence does in English.

Constructing Statements

There are two types of Terms in a Statement, functional indicating some action taken and conditional indicating some test for decision making.

After 3 seconds pass **go to the Running Step.**
Actuate StartFlag, if Start_Switch is on.
Write "Parts Run Complete" to User_Panel.

Statement examples with Functional Terms highlighted.

After 3 seconds pass go to Running Step.
Actuate StartFlag, **if Start_Switch is on.**
Open Vent **when temperature is greater than 100 degrees.**

Statement example with Conditional Terms highlighted.

Functional Terms describe an action to perform when they are reached in the execution of a Task. **Conditional Terms** describe a condition that needs to be evaluated to decide whether the Functional Terms in the Statement should be executed at this time.

A Functional Term, such as "turn_on Motor_A" or "close the Red_Clamp", generally has a verb that describes the action such as, "turn_on", "close", plus a variable name or I/O name, such as "Motor_A","Red_Clamp".

Terms are combined to form Statements. Most Statements will be a sentence or a phrase. A Statement may be entirely made up of a Functional Term such as "Turn_on the Automatic_Mode_Lite.". A Statement can also be a combination of Functional Terms such as "Turn_on the Automatic_Mode_Lite and the Main_Conveyor.". Often a Statement is a combination of a Conditional Term and a Functional Term such as "If Motor_A is on turn_on the Automatic_Mode_Lite and start Main_Conveyor.".

Every Statement must always have at least one Functional Term. A Statement can contain more than one Conditional Term, or a Conditional Term that is a combination of conditions, such as "If Motor_A is on and the Red_Clamp is closed" or "If Motor_A and Main_Conveyor is on". There may be many Functional and Conditional Terms in a Statement.

Using Keywords, Synonyms and Filler Words

Keywords are the words in a Statement that UPTime recognizes as instructions to perform some function. Keyword can be words that cause an action, such as the word "actuate" when applied to a contact output. Or they can cause a conditional comparison such as the word "if", or be part of the comparison such as the symbol ">".

UPTime comes with default keywords assigned. Some of these keywords also have synonyms defined. Using a synonym in the program is the same as using a keyword. A detailed list of all the keywords appears in the State Logic Programming chapter along wit a description of each one.

UPTime also comes with several filler words such as “the” or “a” defined. Filler words have no meaning to the program. The sole purpose of filler words is to make program Statements more readable and understandable. The user can place filler words anywhere in the Statement. Commas and other punctuation may also be used for clarity without effecting program execution. The only punctuation which has meaning is the period (.) and the exclamation mark (!). The exclamation mark is used to document or add comments to the program.

Keywords are the vocabulary of UPTime and together with filler words make it possible to easily write understandable descriptions that are the program. This vocabulary may be changed to suit any desired convention. All of the keywords, synonyms, and filler words may be changed. UPTime can therefore be configured so that the program is written in a foreign language.

A menu based window function allows the user to make these assignments at any time during an UPTime programming session. In addition, another menu based window function allows the user to see a list of all synonyms previously assigned and to select one to enter into the program.

Use the flexibility to create a language that fits the terminology of the industry, or plant, etc. where UPTime is to be used. The written UPTime programs become even clearer to all involved with operating and maintaining the plant as they use the English names for the process points and the local terminology for the actions and descriptions.

Using Variables

Statements can change and test variable values. At times the application requires responses to values other than those represented by real field sensors. Examples of this might be the number of parts built during a shift, the flow through a pipe calculated based upon the pressure drop across the pipe, the style of part being built this production run, etc.

Items stored in variables are the results of calculations, totals that are being accumulated over time, something that must be remembered from one time period to the next, and constants that may be changed or tuned. Each variable is assigned a name during program development.

All variables are can be accessed globally, that is, once created each variable is available to be shared between all Tasks and all Steps within Tasks. One Step may assign a value to the variable and another Step (or the same Step at a later time period) uses the value of the variable in making a decision. Once a variable is assigned a value, that variable maintains that value until a program Statement assigns a new value to that variable.

When using UPTime to write programs, monitor running controllers, or to generate diagnostics, the user needs to only refer to the variable by its English name. Remember that choosing descriptive names for variables helps to make the program self documenting.

The different variable types are listed below:

Integer Variables

This type of variable represents a whole number from +32767 to –32768. Integer variables have many uses including counts, menu choices, and item quantities.

> Integer variables can also be used as logical variables, or variables that have only two possible values, either 1 or 0. Variables used in this manner can be thought of as true or false, on or off, etc. Using integer variables in this manner differs from using flags, since flags are ON only when a Step turning them On is active. On the other hand a variable maintains its value independent of which Step is active.

Internal Flag

Internal flags are variables that act like digital outputs, but do not produce any physical output from the controller. An internal flag can be set true by a Step in one Task and then checked by a Step in another Task. These flags can be used to coordinate the actions of different independent Tasks.

An internal flag is like a digital output in that if an active Step is not setting it true the controller will automatically turn it OFF.

Floating Point Variable

This variable type is used to store numbers that are not whole numbers or numbers outside the range of integer variables.

String Variable

This variable type stores a collection or “string” of characters. These characters can be any alpha numeric or control character represented by an ASCII code. This type of variable is a little more complicated and is used mainly in accepting inputs or creating outputs to serial communication devices.

Character Variable

Character variables store one single ASCII character. This type of variable is especially useful for operator interfaces when the operator must enter a single character.

Time Variables

The time variables are used to view the values of the Real Time Clock. The time variables are second, minute, hour, day, day of the week, and month. Use these variables to check the current time.

Creating Diagnostic Action Sequences (Triggered Sequences)

Triggered Sequences are almost identical to continuous Sequences; they are created the same way and use the same instructions. The only difference is that a Triggered Sequence is specified as an action for a Diagnostic and starts execution only when the Diagnostic is tripped.

To create a Triggered Sequence, select the "Sequences" option from the Create Menu and then the "Diagnostic Sequences" option from the following menu, just as you would when creating a Continuous Sequence. A Triggered Sequence actually starts out as a Continuous Sequence and then becomes a Triggered Sequence when it is specified as an action in a Diagnostic form.

Creating Interactive Informational Sequences (Instructional Sequences)

To create an Instructional Sequence select the "Sequences" option from the Create Menu and then the "Instructional Sequence" option from the following menu. The purpose of these Sequences is to provide an interactive way of providing information to a display connected the serial port designated to be the diagnostics port. Some of the uses of these Sequences are to provide information about the cause and correction of a malfunction, a series of steps to be followed to perform some procedure such as calibrating sensors or doing preventive maintenance, and many other possible types of information to help the equipment operator or technician such as important phone numbers or normal operating parameters such as normal pressures and temperatures.

Instructional Sequences are almost identical to the other Sequence types except that all Instructional Sequence names appear on the instruction menu displayed when <MENU?> is typed on the keyboard. That keyboard is connected to the serial port that is designated to be the diagnostic port. The Sequence begins execution when its name is selected from the menu. The Instructional Sequence can also be started by a Start_Sequence keyword instruction executed in another Sequence.

The purpose of Instructional Sequences is to provide interactive information. The keyword instructions WRITE and READ can be used to send and receive serial information. Another keyword, When_Done_Next_Step, is specially designed for these Sequences, providing more interactive capabilities. This keyword can also be used in any of the other Sequence types. The When_Done_Next_Step instruction causes the text:

Enter R to Repeat Last Step, S to Stop, and C to Continue:

to be displayed on the screen.

Normally a Step in an Instructional Sequence has a Write term that sends information to a serial display. The Step also may have a Read term to receive information typed at the keyboard so that the operator can direct the flow of information. The When_Done_Next_Step keyword also gives the operator control of the display of information. He may repeat the execution of the Step which redisplays the information, stop the Instructional Sequence which redisplays the menu of all Instructional Sequences, or he may proceed to the next Step in the Sequence.

Chapter 4 Tutorial

This chapter presents a step by step procedure for creating a control strategy for a specific real world application. Follow along with the descriptions duplicating each keyboard entry on your own computer. Upon completion of this chapter, you should have a basic knowledge of using the Diagnostic System to reduce downtime in your own system.

To best understand this tutorial, you should first read the chapters titled **Overview** and **Creating a Diagnostic Strategy**.

Tutorial Overview

This chapter provides a step by step examples of creating a Diagnostic Strategy for a real world application. The tutorial is designed for you to follow along with your own copy of UPTime connected to a Diagnostic Processor. It is assumed that you have a working knowledge of Logicmaster, but there are step by step instructions for completing the few Logicmaster tasks in this tutorial.

Lesson Preview

This tutorial is split into six separate lessons. The lessons start out very simple and progressivly become more involved.

LESSON 1:	Single Diagnostic – describes the fundamental procedures for creating and executing Diagnostic Strategies.
LESSON 2:	Several Diagnostics – explains the different types of Diagnostics and introduces the Digital Device structure.
LESSON 3:	Simple State Logic Programming – Introduce fundamentals of creating and executing State Logic programming in the Diagnostic Strategy.
LESSON 4:	Complete State Logic Example – Demonstrates the different capabilities of State Logic programming used together with Diagnostics.
LESSON 5:	Triggered Sequence Example – Demonstrates State Logic programming triggered by a Diagnostic and how deductive analysis is performed with State Logic Programming.
LESSON 6:	Instructional Sequence Example – Demonstrates the creation and uses of Instructional Sequences.

Tutorial Procedure

Disk #2 of the UPTime distribution disks provides files containing the different stages of the development of the Diagnostic Strategy for the Drill Station application. In addition, disk #2 also has the relay ladder logic program that is designed to control this application. The ladder logic is stored in the file DRILL.SDE.

Some of the UPTime project files include a Sequence that exercises the ladder logic control program running in the CPU. With this simulation Sequence, the ladder logic control program and the Diagnostic Strategy can execute without being connected to any real world inputs. The simulation Sequence controls inputs so that the ladder logic program executes as if connected to a real machine.

The simulation is also set up to allow the user to create malfunctions to demonstrate the Diagnostic Strategy responses. The user causes these malfunctions by forcing flags ON by using the UPTime Debug Mode. The simulation Sequence creates faults when a flag is turned ON.

The Sample Application

Figure 4-1. Tutorial Application

The lessons of this tutorial are designed for a drill station. As described in the Overview section, State Logic encourages decomposition of a project into smaller and smaller problems to solve. Envision that this station is only one of several stations on a transfer line. The other stations would be monitored by other Sequences in the Diagnostic Strategy. The Sequences described in these lessons all refer to only the drill station.

Indexing Conveyor

At this drill station there is an indexing conveyor that properly positions a part before it is drilled. This conveyor starts operation by waiting for a part to be loaded at the starting end of the conveyor. A switch indicates when a part has been loaded onto the conveyor.

Once a part is loaded onto the conveyor the conveyor motor starts running, moving the part into position at the drill. A switch indicates when the part is in place, causing the conveyor motor to stop. The part is now drilled and when the drill bit clears the part as it is being retracted, the conveyor motor starts again, moving the part to the unloading position. When the part trips the switch at the unloading position, the conveyor stops. When the part is removed, the process repeats.

Cutting Lubrication System

There is also a lubrication system to lubricate the cutting action of the drill bit. A solenoid controls the delivery of the lubricant and a pump delivers the lubricant. The level of the lubricant reservoir is indicated by an analog signal.

The lubrication system has a pump and a spray solenoid. When the the drill is advancing toward the part the pump comes on and stays on until the bit clears the part when being retracted. The solenoid comes ON when the bit starts to feed into the part so that there is lubricant available when the part is being drilled.

Drill

The movement of the drill is controlled by the feed / retract motor. There are motor contacts that control the speed and direction of the motor. For lowering the drill toward the part there is both an advancing output to move drill to the part and a feed output to move the drill into the part at a feeding speed.

Switches detect the location of the drill. The positions detected are fully retracted, full depth and the feed position for when the drill bit first makes contact with the part.

There is also an auto / manual / off switch that controls whether the feed operation is done automatically or manually. If the operation is done automatically, the drill starts feeding into the part and continues until the full depth switch is tripped.

If the operation is to be done manually, the feed motor stops when the bit makes contact with the part. The drill feeds into the part whenever the feed pushbutton is pressed. When the full depth switch is tripped, the drill retracts automatically. The manual mode only has effect during the feeding operation. The feed motor has a torque feed back that indicates the torque force that the feed motor is exerting to feed the bit into the part.

There is a spindle motor that turns the drill bit. There is an overload contact indicating when the motor experiences an overload condition.

Clamp

When the conveyor places the part in the proper location, it is clamped in place for the drilling operation. The clamp is controlled by a solenoid valve and there are two other switches that indicate when the clamps are open or closed.

The clamp is a critical maintenance item for this station. The number of open / close cycles is a good measure of when the clamp needs to be maintained. For this application there must be notification of when the critical number of clamp cycles have been reached.

Set up Series 90–30 System for the Lessons

For the following lessons, a control system is simulated by executing a ladder logic program in the CPU while the Diagnostic Processor simulates real world input changes from State Logic programming. These programs are included on your UPTime distribution disk #2.

Series 90–30 System Setup

Use the Logicmaster 90 software package to configure your system and load the provided ladder logic program. Follow these steps:

1. After starting Logicmaster, select the configuration package, key <F2>.
2. Create a new program folder for this tutorial by entering a folder name.
3. Set up the configuration of each chassis slot to match the hardware used in your PLC system, <F1> key. Configure the Diagnostic Processor module as a PCM and the Configuration Mode as PCM CFG. Press <Esc> when completed.
4. When the configuration is complete exit the configuration package.
5. Enter the Programmer Package by pressing <F1>. Then select the folder that you just created in the Configuration Package.
6. Select Program Display / Edit <F1>
7. Select More <F9>
8. Select Include <F4>
9. Now put UPTime disk 2 into drive A and enter the ladder logic program file name selected from the table above as follows: A:\DRILL. This operation loads the ladder logic program.
10. Press <Enter> and the DRILL.SDE program file is now part of the Logicmaster folder you are working with.
11. Press <Esc> then <F9> for Utilities. Now press <F2> to Store from Programmer to PLC.
12. Press <Enter> to start the store operation. Press <Y> to questions about changing the program in the CPU.
13. Press <Alt+R> to start the PLC program running.

The CPU is now set up for this tutorial.

UPTime Orientation and Setup

After installing UPTime, following the instructions in the getting started section of this manual, make the logged directory \UPTIME\S90–30 and type **UPTIME** at the DOS prompt. The first screen shows the logo including the current version number.

The next screen shows the standard appearance of most UPTime displays. The top bar displays the project name and the bottom bar displays the major active keys and their functions. The main window displays menus and forms and serves as the editor screen when creating State Logic programming.

The main menu is now displayed on your screen. This menu provides access to UPTime's two modes, Create Mode where the Diagnostic Strategy is created offline and Debug Mode for online operations where real time values and Diagnostic conditions are displayed.

UPTime is a menu driven program which means that most of the available operations are chosen from a series of menus. The bar at the bottom now describes how to use the menu. Use the up and down arrow keys to move the highlight bar to the desired choice then press <Enter>. Selections can also be made by pressing the highlighted letter of the desired choice or pressing a hot key without selecting any menus. Hot keys are shown next to some of the options on the menus and are listed in the Menus and Keys Chapter of this manual.

The instruction bar also indicates that pressing <Esc> cancels the operation and that pressing <F1> enters the HELP system. The function of these keys is consistent throughout the program. Selections from all menus are made the same way.

Pressing <F1> always provides help information on the current operation, therefore it is referred to as a context sensitive help system. While the help screen is displayed, pressing <F1> again displays functions which may be executed by pressing other function keys. Some help screens display instructions on how to view additional help screens. To return from the help screens press <Esc>.

Select the "Create" option from the menu to start creating the Diagnostic Strategy for the drill station application described above.

LESSON 1 – Creating a Simple Diagnostic Strategy

This lesson demonstrates the procedures for both the creation and execution of the simplest Diagnostic Strategies, a single Digital Diagnostic. First a Digital Point is named and then a Diagnostic for that point is created by filling in the blanks of the form.

The main menu for the Create Mode is now displayed. This menu is the gateway to all Diagnostic Strategy creation functions. The first action is to create a new Diagnostic Strategy, referred to as a project in UPTime. Select the "PROJECT" option from the menu and then "Make a New Project" from the next menu. Enter a project name, DRILL, and UPTime creates a new Diagnostic Strategy with that name.

The first Strategy is very simple, using only one Fill–in–the–Blank Diagnostic. This Diagnostic monitors the overload contact from the spindle motor.

UPTime refers to all I/O points by a name so first a name must be attached to the I/O point connected to the spindle motor overloaded contact.

Define a Digital Point

Select the "VARIABLES" option from the menu and then the "Digital Points" option from the subsequent menu. UPTime states that there are no digital points defined so select "YES" to define one.

A form is provided to define a digital name. In the first blank enter the name as follows, SpindleMotorOverload. Notice that name separations are simulated using uppercase letters where a new word would normally start. Press the <Enter> key when the name is typed in. UPTime is actually not case sensitive.

The next blank in the form is now highlighted, so press <Enter> again to list the types of digital points that can be defined. Select %I from the list.

The next blank specifies the address number for this definition. Enter 20 for this number. The next blank is set to INPUT and should not be changed for this example. Now as the bottom bar indicates, to save this information and exit the form press <F9>. The name definition is displayed by itself in a list. Press <Esc> to return to the starting menu. The name SpindleMotorOverload can now be used anywhere in the Diagnostic Strategy to refer to the %I10 memory location.

Create a Fill–in–the–Blank Diagnostic

Now select "DIAGNOSTICS" from the menu. The next menu lists the four Diagnostic types. Select the "Digital Diagnostic" option.

Tell UPTime that you want to create one when it reports that there are no Diagnostics defined. A form is displayed to create the Diagnostic.

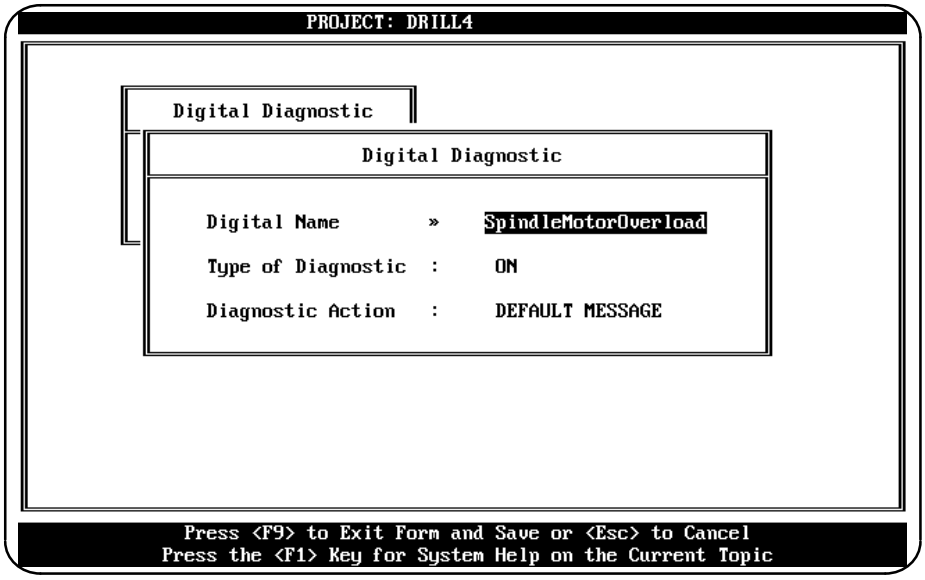


Figure 4-2. Digital Diagnostic Form

The first blank to fill in is for the Name. The Diagnostic uses the name to locate which point to monitor. The '>>' character next to the name blank indicates that UPTime displays a list of names that have been defined for this type of Diagnostic, when the <Ins> key is pressed. Press <Ins> and select the name just defined, SpindleMotorOverload.

The next blank indicates what condition of the point causes the Diagnostic to go to the alarm state. Press any key to display a list of possibilities. Select ON from the list so that this Diagnostic trips when the spindle motor overload is ON.

The last blank selects the action that the Diagnostic initiates when it is tripped. Press any key to display the possibilities. Select the "Default Message" option which causes the Diagnostic Processor to automatically construct and display a message when the Diagnostic is tripped.

Save the data entered by pressing <F9> or just <Enter> when the form is completed. Press <Esc> to return to the menu.

Send Diagnostic Strategy to Diagnostic Processor

The first Diagnostic Strategy is now complete, so the next step is to direct UPTime to translate the project to a format understood by the Diagnostic Processor and then send the translated project to the Diagnostic Processor. Connect the serial ports of the UPTime host computer with the Diagnostic Processor serial port. Use the serial cable and grey adapter included with UPTime.

From the PROJECT menu select the "Translate and Download Project to the Processor" option. Notice the hot key for this option is <Alt+F2>. If these keys are pressed from the previous menu, you need not use the project menu to make this selection.

UPTime asks whether to search for undefined words, select "NO". After some work, UPTime displays a screen of statistics about this project. There should be one digital point listed. Press any key to go online, or in other words enter Debug Mode.

Introduction to Debug Mode

Debug Mode is used for online interaction with the Diagnostic Processor. The two functions of Debug Mode are to monitor execution of the Diagnostic Strategy and to control the Diagnostic System.

Items that the Debug Mode monitors are the state of Diagnostics, values of I/O and variables, the execution of the State Logic program, the list of alarms (tripped Diagnostics), a history of past alarms, current I/O definitions, and PLC Fault tables.

The Debug mode controlling functions are controlling (running, halting, and configuring) the Diagnostic Processor, forcing I/O and changing variable values, changing the current Step of a Task in the program and clearing PLC faults.

Debug Mode Screen

At the top of the screen is the familiar bar displaying the name of the project. As in the Program Mode the bottom bar shows the current use of some of the more important function keys. <F10> returns UPTime to Create Mode for editing the project. Notice that the help system continues to be accessed by pressing <F1>, and that the <F3> key brings up the main menu for this mode.

Terminal Log

There is a window below the top bar titled "Terminal Log". Any messages sent by the Diagnostic Processor are displayed in this window. There are several different types of messages that are sent to the terminal log.

When UPTime is connected to the serial Port designated to be the Diagnostic Message Port, messages from Diagnostics appear in the terminal log. All programmed Write Terms which send messages to the serial port that connects to UPTime are displayed in the Terminal Log. These messages may be operator instructions or menus for the operator to enter information into the program.

All runtime error messages are written into the terminal log with the time that the event occurs. This is an important record of events for troubleshooting or debugging a program. The Terminal Log also displays responses to the Debug Mode commands (Display, Change, and Force). This record also includes a time stamp for each function call.

The first message is displayed at the top of this window with the next messages appearing below the previous ones. The oldest messages scroll out the top of the window when the window fills. Messages scrolled off the screen are not lost but can still be viewed by pressing the up arrow key. After scrolling, you must press <End> to exit the scroll mode and again have access to the Debug Mode menu.

Error and Status Windows

There are two windows on the line below the Terminal Log window. The one on the left displays any run time error messages such as a divide by zero message or a buffer overflow message. Notice that the error message is cleared by pressing <Ctrl + R>. The smaller window on the right displays the current status of the controller, ie. running or halted. The status window also indicates if there are any items forced.

Executing the Simple Diagnostic Strategy

Back to our first Diagnostic Strategy which has just been downloaded to the Diagnostic Processor. Lets run the project (Diagnostic Strategy). Press <F3> to access the menu then select "Project". Now select "Run the Program in the Processor".

The Diagnostic Strategy is now executing, constantly monitoring the Spindle Motor Overload contact. Notice the Status window indicates the processor is RUNNING. If this contact ever goes ON, the Diagnostic trips, sending a message to the screen.

Make the Diagnostic Trigger

To make the Diagnostic go to the alarm state, force the SpindleMotorOverload input ON by following these steps. Press <F3>, then select "Force" from the menu and then the "Set or Modify Force List" from the next set of options. The screen now displays the Force List. As the keys listed in the bottom bar indicates, press <Ins> to add an item to the list. Now select the "Digital Points" option, and then select SpindleMotorOverload which is the only item in the list of Digital Points. Choose the "ON" option which adds the item to the list, and then press <Esc>. Notice the Status window indicates that an item is being forced.

The Diagnostic Processor automatically constructs and displays the message when the Diagnostic trips. The parts of the message for a digital Diagnostic from left to right are whether the message indicates an alarm or clearing an alarm already set, the date and time of the event, the type of Diagnostic, and the name of the digital point. This same message appears if some other display device such as a dumb terminal is attached to the Diagnostic Message Port.

Debug Mode Diagnostic Functions

There are several Debug Mode functions dedicated to providing real time and historical information about the state of the Diagnostics. These functions are all accessed from the menu so press <F3> to start. Select the "Diagnostics" option and then "Digital Diagnostic Display" from the next menu and then the Diagnostic name, SpindleMotorOverload.

A window displays information about the Diagnostic configuration and the current value of the Diagnostic I/O point being monitored. Your screen should show that the digital value is "FORCED ON". Hit <Esc> twice to return to the terminal log screen.

Select "ALARM" from the menu and then "Display Current Alarm List" from the next set of options. This option lists all of the Diagnostics that are currently in an alarm state. The information displayed shows the date and time the Diagnostic was tripped, the name of the item being monitored, and the value causing the diagnostic to trip. Hit <Esc> to return to the terminal log screen.

Clear the force condition on the Digital Point, SpindleMotorOverload, by selecting the "Clear All Forces" option after selecting "FORCE" from the menu. The Diagnostic Processor constructs another message that is displayed in the terminal log screen showing that the alarm has been cleared. Look at the alarm list again to see that there are currently no alarms to view.

Select "ALARM" from the menu again and then choose the "Display Current Histogram" option. The Diagnostic Processor maintains a record of Diagnostic changes. Every time

a Diagnostic trips or clears an entry this histogram logs that event. The histogram also logs each time the project is started or halted. The most current events are displayed at the top of the list.

Congratulations, you have just created and run your first Diagnostic Strategy. When creating Diagnostic Strategies, it is a good practice to start out simple and move slowly toward more complicated efforts as you learn more about the Diagnostic System and the information you need to know about your own application.

LESSON 2 – Adding More Diagnostics

You can create a Diagnostic for every I / O and variable defined for an application. In this lesson we add a Diagnostic for each of the other Diagnostic types.

Press the <F10> key to go from Debug Mode back to Create Mode. For this lesson add the data elements in the following table to the DRILL project created in the previous lesson.

Name	Type	Address
FeedTorque	Analog Channel	%AI1
ClampCycles	Integer Variable	N / A
ClampSolenoid	Digital Point	%Q2
ClampOpen	Digital Point	%I6
ClampClosed	Digital Point	%I5

Analog Diagnostic

First define the analog channel, FeedTorque, by selecting the "Analog Channels" option from the VARIABLES menu. Fill in the blanks for the name, type, and number as shown in the table above. Select Input for the Input or Output blank.

When defining the analog channel, UPTime asks whether to scale the channel. UPTime automatically scales the raw data to engineering units. For this example enter YES to the scaling option and enter A / D or D / A values to be a LOW of 0 and HIGH of 32767 and engineering units of 0 and 500. These scaling values mean that the full range of torque in the sensor, which might be a 4 – 20ma signal and is converted to a number in the range of 0 to 32767 by the analog module, actually indicates real world values of from 0 to 500 foot–lbs. References to scaled analog values are floating point values. References to unscaled channels are integer values.

Now create an analog Diagnostic using FeedTorque for the analog name. The analog Diagnostic form allows both high and low limits to be specified and allows floating point values for the limits. Enter 125 for the high limit and 4 for the low limit. For the High Limit action select "DEFAULT MESSAGE" and "CUSTOM MESSAGE" for the low limit action.

Press <F9> to save the data. UPTime provides a window to enter the custom message for the low limit action. Enter the following text:

"There is no torque on the Feed Motor. The drive chain may be broken or may have come off the sprocket."

Variable Diagnostic

Now define the variable ClampCycles that stores the number of open and close cycles of the clamp for predictive / preventive maintenance purposes. The value of this variable is controlled by State Logic programming and is explained in a subsequent lesson.

Create the variable thru the VARIABLES menu by selecting "Numeric Variable" and then selecting "Integer" for the data type. The form for defining the variable has a blank for the name and another for the "Save Over Halt" option. This option specifies whether the value in the variable should be maintained whenever the processor goes through a run cycle whether stopped from Debug Mode or because of a power failure. Select "Y" for this option.

Next create the Variable Diagnostic selecting Integer for the type and using ClampCycles as the integer name. The Variable Diagnostic also has a high and low limit. Enter 22000 for the high limit and 0 as the low limit.

For the actions select "CUSTOM MESSAGE" for the high limit and "NONE" for the low limit action. The NONE selection means the diagnostic does not trip when the value is less than the low limit. For the custom message for the high limit action enter the following text:

Clamp maintenance past due. Number of clamp cycles since last maintenance is %ClampCycles – maximum allowed: 20000.

This text demonstrates how variable values can be used in a custom message. When the message is displayed, the value of the variable following the per cent sign (%) is inserted at the variable location. Both numeric and character variable can be used. See the section describing the WRITE term in the State Logic Programming chapter of this manual.

Device Diagnostic

A Device is a unique higher level UPTime structure that treats a group of Digital Points as one entity. A Device has many uses but is designed primarily to represent mechanisms that cycle between two states and usually includes some travel from one state to the other.

Devices usually have two digital outputs, each causing the mechanism to go to a specific state. Normally Devices include two inputs, each signalling that the mechanism's travel from one state to another is complete. A Device Diagnostic monitors the time that it takes for the mechanism to change state once the forcing output starts the change. The Device Diagnostic can also be set up to trigger every time the mechanism reaches either state.

Mechanisms that can be represented by the Device structure are presses, hydraulic rams, motors, and clamps. This structure is very flexible and can therefore be used for several purposes. It is not necessary to use an output for the point causing a change or use an input for points that detect the change, in fact any of the different types of digital points can be used for any of the points that define the Device. The same digital point can be used more than once in the Device definition. Use Devices and the Device Diagnostic when the time from the point that one digital point changes state until another digital point changes state is critical.

Define the Digital Points

In this application the Device is used to represent the clamp. The first step in defining this Device is to define the digital points connected to the clamp.

Define ClampSolenoid, ClampOpen, and ClampClosed as shown in the table above. In the digital definition form the Input or Output blank is used to indicate whether the ladder logic program in the CPU controls that point or the Diagnostic Processor controls the point. Normally the CPU controls all digital points and the Diagnostic Processor only monitors the points, therefore digital points are usually defined as inputs. Even the %Q points are considered to be inputs to the Diagnostic Processor since the points are only viewed not changed.

The Diagnostic Processor does have complete control capabilities, and it does make sense to use the Diagnostic Processor to flash alarm lights or set a communication bit for the CPU. For these cases the digital points are defined as outputs.

Define the Device

To define this Device select “Digital Devices” from the VARIABLES menu . Fill in the blanks as shown in the following screen display.

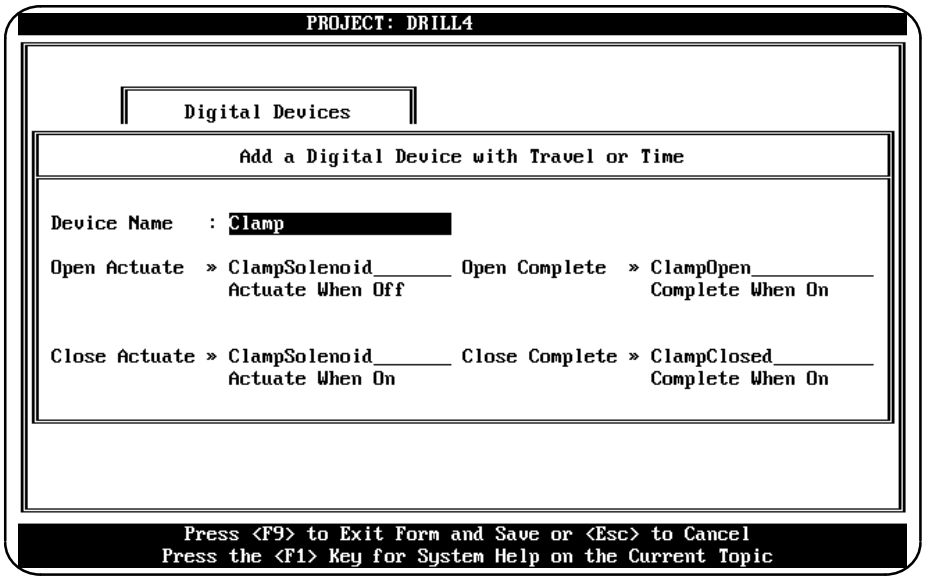


Figure 4-3. Defining the Clamp Device

The text under each digital point, "Actuate When ON / OFF" or "Complete When ON / OFF", is used to indicate which condition of the digital point starts the operation or signals the completion. When that text is highlighted pressing any key except <Enter> changes the ON / OFF indication. Notice that ClampSolenoid is used to start both the open and close cycles. When the solenoid is ON, the clamp closes. The clamp opens when the solenoid goes OFF.

Define the Diagnostic

Fill in the Device Diagnostic form for the Clamp diagnostic as shown in the following screen capture.

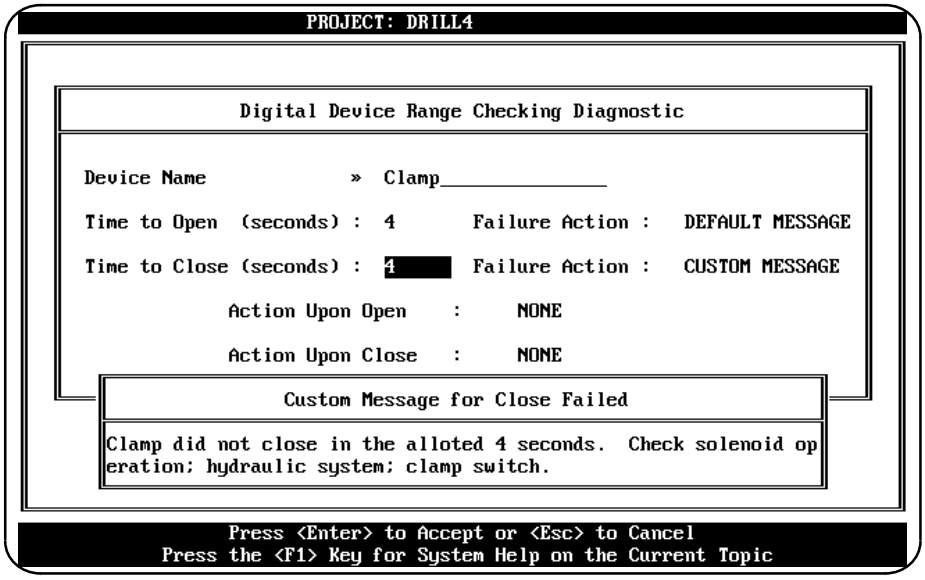


Figure 4-4. Defining Clamp Device Diagnostic

Notice that there are four possible actions for the Device Diagnostic. The CUSTOM MESSAGE text for the Close Failed action is displayed.

Execute the Strategy

Now execute this strategy in the Diagnostic Processor. First translate and download the project, using the <Alt + F2> hot key. When you reach Debug Mode run the project.

Now use the force option as in the previous lesson to force the analog channel FeedTorque to 130 to demonstrate the default message constructed by the Diagnostic Processor for the analog high limit action.

To demonstrate the action for the ClampCycle variable Diagnostic, change the variable value to 22000. From the menu select "CHANGE", then "Numeric Variables", and select ClampCycles, and then enter 22000. The first message records the change command and the time the command was invoked. The next message is the custom message for the high limit action of this Diagnostic. This message demonstrates inserting variable values in the message, by embedding the ClampCyles value in this message.

Creating State Logic Continuous Sequences

This section describes the preparation for the State Logic programming sections. State Logic Sequences are used for more complex analysis and functions than available with the Fill–in–the–blank Diagnostics. See the Overview and Creating a Diagnostic Strategy chapters of the manual for descriptions and directions for creating the three types of Sequences (Continuous, Triggered, and Instructional). The section after this one continues the lessons with lesson 3.

Outline the Application

The first step in creating the Continuous Sequences is to decompose the application into its separate systems which operate in parallel (at the same time). This level of decomposition gives us the highest level of the outline of this application.

Identify the Sequences and Tasks

Most Sequences have only one Task. Most Tasks are identified easily as the natural systems or sub–systems of the application. For situations where it is not clear where to divide the application into different Sequences, look to the parallel and sequential aspects of the application and the diagnostic functions that must be accomplished. Events that occur one after the other belong in the same Task, but an event that occurs at the same time as another event belongs in a different Task.

The Sequences for this application each have only one Task. The top level outline for this application is naturally broken down into the four systems:

IV. Indexing Conveyor Operation

V. Drill Bit Lubrication System

VI. Drill Operation

VII.Clamp Cycles

Identify the Steps of the Tasks

The goal of this phase of development is to follow the system through its states of operation so that there is one Step for every state of the system. When the real world system changes state the State Logic Task should also change its current Step. Right now the goal is to follow the states of the application so that there is a framework from which to add Diagnostic functions as they are needed.

I. Indexing Conveyor Operation

- **A. Loading Station Waiting**
- **B. Moving Part to Drill**
- **C. Part At Drill**
- **D. Moving Part to Unload Station**
- **E. Unloading Station Waiting**

II. Drill Bit Lubrication System

- **A. Lubrication System is OFF**
- **B. Lubrication System is Ready**
- **C. Lubricating**

III. Drill Operation

- **A. Waiting for Part to Arrive**
- **B. Drill Advancing**
- **C. Feeding**
- **D. Stopped Feeding**
- **E. Retracting Drill**
- **F. Drill Retracted**

IV. Clamp

- **A. Clamp ON**
- **B. Clamp OFF**

Identify the Transition Statements

As a general rule, the outputs of a system indicate a change from one Step of a process to another. There are times when inputs must also be used, but generally monitor the outputs of the control program to detect when the process changes to another step. The errors or faults of the system are indicated by comparing the inputs and times to normal or expected conditions. For example, after the Conveyor motor comes on it is expected that the PartAtLoad switch goes off in 1.5 seconds. If the switch does not respond as expected, there is a fault in the system.

First program only the Step transitions so that Tasks just follow the process, and do not attempt to do any diagnostic testing. After the transitions are programmed, the result is a framework or outline of the application that is a direct representation of the process being controlled. Once the framework is in place diagnostic checks are added, checking for likely faults at different Steps of the process. The first part of this section shows the Tasks programmed just to follow the application.

Task: Clamp_

Step: ClampOFF
If ClampSolenoid is ON, go to ClampON.

Step: ClampON
If ClampSolenoid is OFF, go to ClampOFF.

Clamp Task

This is a very simple Task that describes a very simple apparatus. The output controlling the clamp is always in one and only one of these two Steps.

Task: INDEXING_CONVEYOR_

Step: WaitingToStart
If ConveyorMotor is ON, go to MovingToDrill Step.

Step: MovingToDrill
If ConveyorMotor is OFF, go to PartAtDrill Step.

Step: PartAtDrill
If ConveyorMotor is ON, go to MovingToUnload Step.

Step: MovingToUnload
If CoveyorMotor is OFF, go to WaitingToStart.

Indexing Conveyor Task

This Task is a good example of the power of State Logic. Because of its sequential nature, the State Logic program keeps track of the Step of the process, providing a lot more information than just knowing the status of the I/O of the system. The process can be in more than one state with the same state of the I/O.

Task: Drill_

Step: WaitingForPart
If RapidAdvanceOutput is ON, go to Advancing Step.

Step: Advancing
If FeedAdvanceOutput is ON, go to Feeding Step.
If FeedPosition is ON, go to StoppedFeeding.

Step: Feeding
If FeedAdvanceOutput is OFF, go to StoppedFeeding Step

Step: StoppedFeeding
If FeedAdvanceOutput is ON, go to Feeding Step.
If RetractOutput is ON, go to Retraccting Step.

Step: Retracting
If RetractingOutput is OFF, go to WaitingForPart.

Drill Task

This Task demonstrates how the process may branch, with different Steps being an active part of the process at different times. During the feeding cycle there is provision for the manual mode where the feed speed contact to the feed / retract motor is cycled OFF and ON in response to the feed pushbutton, but again only outputs are used to indicate when another Step of the process is active.

The Advancing Step shows a situation where inputs are used to sense a change to a different Step of the process. Inputs must be used in this case to detect whether the system is in the feed position, but feeding has not started, because either the manual switch is ON or there is some fault in the system. In most cases though ouputs are all that must be monitored.

Task: Drill_Bit_Lubricate_

Step: LubricationOff
If LubricationPump is ON, go to LubSystemReady Step.

Step: LubSystemReady
If LubricationSolenoid is ON, go to Lubricating Step.
If LubricationPump is OFF, go to LubricationOff.

Step: Lubricating
If LubricationSolenoid is OFF, go to LubSystemReady Step.
If LubricationPump is OFF, go to LubricationOff.

Drill Bit Lubrication Task

This is a simple Task that cycles with the lubrication system. If the pump is ON the system is ready, and if the solenoid is ON the system is lubricating.

Identify the Diagnostic Statements

Once the Tasks, Steps, and transitions are in place, Statements to perform diagnostic functions are added for the Steps where you want to watch for problems that are anticipated.

The outputs have been used to establish how the process transitions from Step to Step, how the inputs, time, variable values, and the knowing the active Steps of other Tasks are used to detect abnormal operation or other important conditions.

Once an anticipated condition is detected one of several functional terms are used to take the desired action. A message can be sent to one of the serial ports, a variable value can be changed, a digital or analog output can be changed, the active Step of another Task can be changed, or a specialty function can be executed. Another option is to go to another Step to accomplish some operation then return to the Steps that normally follow the process operation.

Examples of diagnostic Statements testing for different conditions and demonstrating different operations follow:

If ForwardLimitSwitch is OFF, write "Chassis stuck in forward position".
If StationOperation Task is in the Completed Step, add 1 to PartCount.
If 37.5 seconds have passed, turn on the TimeDelayFlag.
If TankTemperature + 45 > OptimalTemperature * 2, go to OverTemp Step.

LESSON 3 – Create State Logic Programming (Continuous Sequences)

In this lesson one very simple Task is created and executed. The State Logic simulation of the drill station is used to demonstrate the execution of the simple Task. This section introduces the UPTime editor, specifically designed to create State Logic programming. Also downloading the project and online interaction with the program is introduced.

Load Simulation Program

The lessons using State Logic programming are designed to execute with the ladder logic program that is already loaded into the CPU. The lessons rely on that program to be cycling through its logic. Since the PLC is not connected to any real world I/O, the distribution disks have an UPTime project that simulates the real world inputs. The project, called DRIL–SIM, has a Sequence that controls the inputs of the ladder logic program so that it behaves as if connected to a real world drill station. This project also includes all of the Diagnostics created in the previous lessons.

The simulation project is first loaded into UPTime and then the State Logic lessons are added to this project. Follow these steps to load the simulation project into UPTime:

1. From the Create Mode main menu select "PROJECT".
2. Place distribution disk #2 into drive A.
3. Select "New Path/Drive for Projects" from the PROJECT menu.
4. Enter A:\ in the box provided.
5. Select the PROJECT menu again and then "Retrieve a Project from Disk" option.
6. Select the project named DRIL–SIM from the list.

The simulation project is now loaded to make the ladder logic program execute its logic. The lessons now add to this project. All of the simulation is in the SIMULATION Sequence.

Add and Execute a Sequence

First add the CLAMP Sequence to the simulation project. To add the Clamp Sequence first select "SEQUENCES" from the Create Mode Menu, then select "Diagnostic Sequences" from the next menu.

There is the Sequence for process simulation displayed in the list of Sequences. As the bottom bar indicates to add a Sequence press the <Ins> key, then for the Sequence name type in **CLAMP**. Select NO for the Restart in Last Step option. Enter **ClampOff** for the name of the first Step, and then just press <Enter> without entering any data for the Max Time option.

Using the Editor

We have now entered the UPTime editor. The editor provides many word processor functions, as the bottom bar shows, the editor menu for accessing these functions is available by pressing the <F3> key. One Sequence at a time is edited. Its name is displayed in the top bar. Text typed in is entered at the cursor position.

The Task and first Step name are entered automatically. There is nothing magical about this text, as it could have been entered directly from the keyboard and can be changed at any time.

Enter the rest of the Sequence as follows:

Task: Clamp_

Step: ClampOFF
If ClampSolenoid is ON, Add 1 to ClampCycles and go to ClampON.

Step: ClampON
If ClampSolenoid is OFF, go to ClampOFF.

To the framework Steps shown earlier has been added a counter function to maintain the number of times the clamp has closed. This data is used by the ClampCycles variable diagnostic created earlier to indicate when it is time to perform maintenance on the clamp.

Download Project

The next step is to download the project to the Diagnostic Processor. First exit the editor by pressing <F3> and then selecting the "RETURN" option from the menu. Now select the "PROJECT" option from the Create Mode menu. From the Project menu select the "Translate and Download Project to the Processor" option.

UPTime asks whether to search for undefined words, select "YES". UPTime now looks through the program trying to find any words that are not filler words, keywords or defined names. There are no undefined words, so UPTime completes the download.

Debug Mode

Now go to Debug Mode after the statistics screen is displayed. To observe the Clamp Sequence execute first create a monitor table.

1. Press<F3> for the menu and then select "MONITOR". Choose the "Add a New Monitor Table" option, and then enter **CLAMP** as the name for the table.
2. The next display shows a list of the different types of data that can be entered into the table. Select "Task's Current Step" and then CLAMP from the list of Tasks.
3. Now press <Esc> to bring up the data type list again. This time select "Digital Points" and then ClampSolenoid from the list of digital points.
4. Press <Esc>, then Numeric Variables, then select ClampCycles. Now press <Esc> 4 times to return to the Terminal Log screen. The monitor table now displays the value of those items.

Run the project and watch the values change as the State Logic simulation Sequence causes the ladder logic program to cycle through its program. Create a monitor table to watch the critical elements of the program, especially the current Steps of the Tasks.

LESSON 4 – Add Diagnostic Logic to Framework Logic

This lesson demonstrates several methods of implementing diagnostic testing in continuous Sequences. All of the Tasks displayed earlier that provided the framework of following the drill station application are added to in this lesson.

From the distribution disk number 2 load in the project named DRILL2. This project includes the continuous Sequences with diagnostic Statements as listed below, plus several added Diagnostics, and all of the Diagnostics created in the first 2 lessons.

Load this project into UPTime and then check the different elements of this Diagnostic Strategy as it is described below each Task listing.

Indexing Conveyor

```
Task: INDEXING_CONVEYOR_

    Step: WaitingToStart
          If ConveyorMotor is ON, go to MovingToDrill Step.
          If 8 seconds have passed, go to the Delayed Step.

    Step : Delayed
          Turn on PartDelayed.
          If ConveyorMotor is ON, go to MovingToDrill Step.

    Step: MovingToDrill
          Enable_Diagnostic PartAtLoad.
          If ConveyorMotor is OFF, go to PartAtDrill Step.

    Step: PartAtDrill
          Enable_Diagnostic PartAtDrill.
          If ConveyorMotor is ON, go to MovingToUnload Step.

    Step: MovingToUnload  Max_Time 8
          If ConveyorMotor is OFF, go to WaitingToStart.
```

To the framework Task listed in a previous section have been added several Statements to perform diagnostic checks for the different stages of the process. The two new Diagnostics that have been added for this Task are listed in the following table:

Table 4-1. New Diagnostics for Indexing Conveyor

Name	Class	Address	Type
PartDelayed	Digital	%M50	ON
PartAtLoad	Digital	%I11	ON

Part Delayed

This Diagnostic and the Delayed Step demonstrate using a Digital point as a flag to trip a Diagnostic. The Diagnostic is continuously monitoring the point and is tripped when the State Logic program sets the flag.

If the conveyor waits 8 seconds for a part, the Diagnostic sends a message out the serial port and an entry is made in the alarm list and the alarm histogram. It may be that there is a mechanical reason for the delay or that an operator is not doing their job, in either case a record is made that the drill station has been delayed due to a lack of parts.

The %M digital point is used as a flag to trip the Diagnostic. Notice that this digital point is defined as an output, since its value is controlled by the Diagnostic Processor. Also note that there is no logic to turn OFF this point. Since outputs are OFF by default, this point goes OFF when the Task changes to a Step that is not turning the point ON.

PartAtLoad

Using this Diagnostic demonstrates how Diagnostics are enabled when needed and disabled when not needed. The State Logic program is following the application and controls the Diagnostic at the correct Step of the process.

After the conveyor starts moving the part to the drill, the PartAtLoad Diagnostic is enabled. This Diagnostic informs the operator in the event that a part is placed in the load position before the conveyor system is ready for the next part. This Diagnostic is Disabled just after the Task transitions to the WaitingToLoad Step, when a new part is expected to be loaded.

Max_Time Diagnostic

The last Step of this Task demonstrates the Max_Time built in time out function. This function is created by following a Step name with the Max_Time keyword followed by the maximum number of seconds that that Step should be active.

In this case the maximum amount of time that it should take for the part to move from the drill to the unload position is 8 seconds. If it takes longer, a message is sent to the terminal log saying that the Indexing_Conveyor_ Task was too long in the MovingToUnload Step.

Drill

```
Task: Drill_

    Step: WaitingForPart
        If RapidAdvanceOutput is ON, go to Advancing Step.

    Step: Advancing
        If FeedAdvanceOutput is ON, go to Feeding Step.
        If FeedPosition is ON, go to StoppedFeeding.
        If 0.5 seconds and RetractPosition is ON,
            write "The Retract Switch has not gone OFF!

    Possible causes are:
        1. Switch is defective.
        2. Drive Belt is Broken.
        3. Drive Belt Jumped off of the Pulley." to OperatorPanel.

    Step: Feeding
        Enable_DiagnosticLubOFF.
        If FeedAdvanceOutput is OFF, go to StoppedFeeding Step.

    Step: StoppedFeeding
        If FeedAdvanceOutput is ON, go to Feeding Step.
        If RetractOutput is ON, go to Retracting Step.
        If 5 seconds,
            write "Drill in manual mode.  Press pushbutton to feed."
                to OperatorPanel.

    Step: Retracting
        Disable_DiagnosticLubOFF.
        If RetractingOutput is OFF, go to WaitingForPart.
```

The Advancing and StoppedFeeding Steps of the Drill_ Task demonstrate State Logic programming used to send a diagnostic message out a serial port. The second Statement in the Advancing Step tests for the case that the switch indicating the drill in the retract position has not gone off when expected. The message lists three possible causes for the problem. The WRITE term in the StoppedFeeding Step indicates that the drill is in manual mode and is waiting for the feed pushbutton to be pressed in order to continue.

Messages can be sent to either serial port by specifying the name of the port after the message. All text within the double quotes is sent formated just as it appears in the program. The message from the WRITE term is displayed of the screen, but there is no indication in the alarm list or the histogram that this condition exists. If UPTime is connected to the Diagnostic Processor, the message is displayed in the Terminal Log where the message is captured for future review.

There is one added Diagnostic for this Task, LubricationOFF. The digital point for this Diagnostic is set by the Bit_Lubrication Task everytime the lubrication system is OFF, but the Diagnostic is enabled and disabled by the Drill_ Task. The use of this Diagnostic shows how to check on conditions in another system of the process.

Drill Bit Lubrication System

Task: Bit_Lubrication_

Step: StartUp
If LubricationPump is ON, go to LubSystemReady Step.

Step: LubricationOff
Disable_DiagnosticLubricantLevel.
If LubricationPump is ON, go to LubSystemReady Step.
If 2 seconds have passed, turn on LubOFF.

Step: LubSystemReady
Enable_DiagnosticLubricantLevel.
If LubricationSolenoid is ON, go to Lubricating Step.
If LubricationPump is OFF, go to LubricationOff.

Step: Lubricating
If LubricationSolenoid is OFF, go to LubSystemReady Step.
If LubricationPump is OFF, go to LubricationOff.

This sequence demonstrates activating a Diagnostic when it is needed and disabling it when its messages are not meaningful and would clutter up the alarm lists. The flag LubOFF is also used to indicate to another Task when the Lubrication System is off. This flag is checked by a Diagnsotic that is enabled by the Drill_ Task.

Execute DRILL2 Diagnostic Strategy

Download this project to the processor and start it running. The simulation Sequence provides a means for creating alarm situations for the Diagnostics and diagnostic programming demonstrated in this project.

The following list displays the names of flags that cause the simulation Sequence to cause a system fault. They cause faults, force these flags ON one at a time then observe how the Diagnostic System responds.

Flag	Description
DelayLoad	Causes part to be delayed arriving to the load position on the conveyor
LoadON	Causes PartAtLoad switch to stay ON when conveyor starts
RetractStuckON	Causes the RetractPosition switch to stay ON when drill is advancing.
LubricantLowFlag	Causes the lubrication level to be –20 which is low enough to trigger the LubricantLevel Diagnostic but not low enough for the ladder logic control program to stop the system.
ManualModeFlag	Causes manual mode switch to be ON. After advancing to the feed position, the drill waits for the pushbutton to be pressed before drilling the part. Use this flag together with the Feed-Flag.
FeedFlag	Simulates the feed pushbutton being pressed. When this flag is ON, the feed pushbutton is also ON. Use this flag together with the ManualModeFlag.

LESSON 5 – Triggered Sequence

This lesson introduces the triggered Sequences. This Sequence type starts execution when a Diagnostic triggers and the action of that Diagnostic is specified to be the SEQUENCE.

This Sequence shows a typical use of State Logic to analyze the cause of a problem. By knowing the process, the programmer can create deductive programming that uses I/O status, serial interaction with the operator, the current Step of Tasks, and other data items to deduce the cause of a fault.

For this lesson use the DRILL3 project from disk 2 of the UPTime distribution disks. The Sequence added in this lesson triggers off of the FeedTorque Diagnostic created in lesson 2. The only difference between a Continuous Sequence and a Triggered Sequence is that the Triggered Sequences start execution when triggered by a Diagnostic, and Triggered Sequences must have an End_Sequence keyword in one of its Steps.

```
Task: FEED_OVERTORQUE_

    Step: Initialization
        MinimumLevel = −15.
        Go ReportOvertorque.

    Step: ReportOverTorque
        Write “Too much torque on FEED MOTOR” to OperatorPanel.
        If Lubrication_ Task is in the LubricationOff Step,
            go to the LubricationPumpOff Step.
        If LubricationLevel is < MinimumLubLevel go to LubricationLow.
        Go to CheckLubAction Step.

    Step: LubricationLow
        Write “Lubrication Reservoir is LOW!”  to OperatorPanel.
        Go to Finished.

    Step: CheckLubAction
        Write “Is lubrication flowing on drill bit? (Y/N)” to OperatorPanel.
        Read Response from OperatorPanel and go to CheckResponse.

    Step: CheckResponse
        If Response = “N” go to NozzleClogged.
        If Response = “Y” go to BitDull.
        Write “Invalid Response − Try Again!%CRLF” to OperatorPanel and
            go to CheckLubAction.

    Step: NozzleClogged
        Write ”
        Drilling action not lubricated! Nozzle is clogged.
        Stop Drill Station Operation and Clean and Adjust Nozzle!”
            to OperatorPanel.
        Go to Finished.

    Step: LubricationPumpOff
        Write ”
    Lubrication Pump is not running!!
    Check these items:
        1. Overload Interlock
            Reset Button is behind motor starter.
        2. Circuit breaker 12 in main panel.
            If breaker trips after reset call maintenance at 3498”
        to OperatorPanel.

        Go to Finished.

    Step: BitDull
        Write “Over Torque condition caused by dull bit.
            Type ’MENU?’ and press <Enter> for instruction list.
            Select the REPLACE Drill BIT option for instructions.”
        to OperatorPanel.

        Go to Finished.

    Step: Finished
        If FeedTorque < 125 then go to end.

    Step: End
        End_Sequence.
```

After this Sequence is added to the project, it is listed as a Continuous Sequence. When the FEED_OVERTORQUE Sequence is named as the action for the FeedTorque Diagnostic high limit action, this Sequence becomes a Triggered Sequence.

Download and run this project and force on the following flags ON to create fault conditions. Each flag sets the FeedTorque analog value to be 130 which causes the Diagnostic to trip when the drill is feeding.

For some flags other conditions are created to indicate other problems in the system. Respond to questions when the Sequence prompts for some information. The Sequence deduces the problem and informs the operator. The prompt to type MENU? is prompting the user to call up the Instructional Sequences which are not available in this program but are added in the next lesson.

Clear the force to allow the program to continue running normally.

Flag	Description
HighTorqueLowLub	Makes the analog channel value for the lubricant level be too low.
HighTorqueLubOff	Makes the lubrication level go to –280 which is low enough for the control program to turn off the lubrication pump. This fault shows the proper division between control and diagnostics. The critical situation is handled by the controller, but the Diagnostic System notifies what caused the system to stop.
HighTorque	This flag just causes the FeedTorque to be high.

LESSON 6 – Instructional Sequences

This section demonstrates the creation and execution of Instructional Sequences. These Sequences are created the same way that the other Sequences are created and the same keywords are available for use in both. To start entering an Instructional Sequence, select SEQUENCES from the menu then Instructional Sequences.

What makes Instructional Sequences different is how they start execution. When in the UPTime Debug mode or from a dumb terminal, a menu of all Instructional Sequences is displayed on the screen when <MENU?> is entered at the keyboard. From this menu any one of the Instructional Sequences can be started. Instructional Sequences may also be started by other Sequences by using the Start_Sequence keyword.

To use the Instructional Sequences load in the project DRILL4. This project includes the Instruction Sequence listed below in addition to all of the Diagnostic Strategy created in the previous lessons. When this project is running and the Debug Mode Terminal Log screen is displayed, type **MENU?** at the keyboard to start the Instruction menu.

The three Instructional Sequences used in this program are displayed in the following sections.

```
Task: FILLING_RESERVOIR_
   Step: Introduction
      Write "%cls
         FILL LUBRICATION RESERVOIR STEPS

1. Turn OFF Lubrication Pump
2. Move Lubrication Drum 3tR to Drill Station Area Red Square
3. Remove Reservoir Cover
4. Insert Delivery Hose
5. Use Hand Pump to Fill to Green Line
6. Remove Hose
7. Replace Cover
8. Start Lubrication Pump

" to OperatorPanel.

   End_Sequence.
```

The Filling_Reservoir instrctions just present a list of steps to be performed. Notice the End_Sequence keyword at the end of this Sequence. Each Triggered and Instructional Sequence must have this keyword somewhere in that Sequence. When this keyword is executed, all of the Tasks in the Sequence stop execution and go to the Inactive Step.

```
Task: REPLACE_DRILL_BIT_
   Step: CheckDrillCycles
      If DrillCycles < 200 go to BitDullEarly.
      Go to StartReplacement.

Step: BitDullEarly
    Write "Drill bit is dull in fewer operations than expected.

Check the following:
    1. Lubrication System
    2. Feed Rate

Report Incident to Shift Foreman" to OperatorPanel.

          Write "Press any key then <Enter> to Acknowledge this message
          and display replacement procedure." to OperatorPanel.

          Read Acknowledge then go to StartReplacement.

Step: StartReplacement
    Write "
          Follow these Steps for Bit Replacement:

    1. Set Stop Switch
    2. Remove Bit By Pulling Release Locking Handle
    3. Find New Bit in Locker 13.
    4. Insert Bit and Secure Locking Handle" to OperatorPanel.

Write "Drill Bit Replacement Procedure Complete!" to OperatorPanel.
          DrillCycles = 0.

          End_Sequence.
```

This Task shows how different instructions are given depending on some test of data stored in the PLC system. If the number of drill operations on the bit is lower than a minimum expected, the operator is given some special instructions. In both cases the normal replacement procedure is provided.

```
Task: CLAMP_MAINTENANCE_
   Step: StartMaintenance
      Write "%CLS
   There are three steps for maintaining the clamp:

1. Disassembly
2. Lubrication
3. Assembly

" to OperatorPanel.

When_Done_Next_Step.

Step: Disassembly
    Write "
       DISASSEMBLY

1. Remove pivot pin snap rings.
2. Punch pivot pins from.
3. Remove return springs.
4. Remove clamp jaws.
" to OperatorPanel.

When_Done_Next_Step.

Step: Lubrication
Write "

    LUBRICATION

1. Liberally coat pivot pins with MOLY–L UB  2–4D.
2.  Use SUPER–GEL grease gun on zerks under clamps." to OperatorPanel.

When_Done_Next_Step.

Step: Assembly
Write "

          ASSEMBLY

1. Use alignment tool to position clamp jaws.
2.  Use  small  port–a–press to push pivot pins in place.
3. Connect return springs." to OperatorPanel.

When_Done_Next_Step.

Step: ResetCycleCount
ClampCycles = 0.
End_Sequence.
```

The Instructions for maintaining the clamp demonstrate using each Step of the Sequence to describe a Step in the procedure. The When_Done_Next_Step keyword is used to give the user control of displaying the Steps of the procedure. When this keyword is executed, a prompt is displayed asking whether to proceed, repeat the last Step, or end the instructions.

Chapter 5 Diagnostic Processor

This chapter focuses on the Diagnostic Processor. The hardware features are described in detail, and there is a section that describes how the Diagnostic Processor interacts with the Series 90–30 control system. Other sections describe setup options and how the State Logic program scan works.

Hardware

The Diagnostic Processor is a module which inserts into a Series 90–30 PLC chassis. This module has the State Engine operating system to execute Diagnostic Strategies composed of Fill–in–the–Blank Diagnostics and State Logic programming produced by the UPTime programming software package.

The Diagnostic Processor is inserted together with a CPU module into the Series 90–30 PLC chassis. The CPU must be a model 331 or 341, Revision 3.52 or higher. The CPU executes a Relay Ladder Logic Program at the same time the Diagnostic Processor is executing a diagnostic strategy. The Diagnostic Processor manipulates the CPU memory space and the CPU controls the I/O during its normal cycle of operations. More than one Diagnostic Processor can be installed in one Series 90–30 PLC system, and the Diagnostic Processor must be installed in the main rack with the CPU.

Physical Description

The Diagnostic Processor occupies a single slot in the Series 90–30 rack, communicating with the PLC CPU over the backplane to perform its many diagnostic functions. The Diagnostic Processor accesses CPU I/O, register, and system data. The Diagnostic Processor has control capabilities, but the system is designed with the expectation that the CPU does almost all of the control.

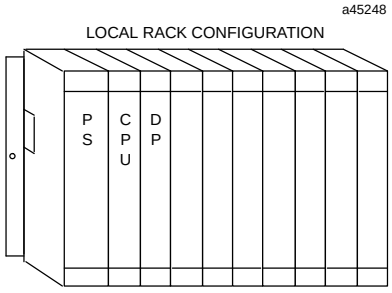


Figure 5-1. Diagnostic Processor in Series 90–30 Chassis

The 90–30 Diagnostic Processor uses an 8 mHz 80C188 microprocessor and has two serial ports, port 1 is an RS–232 port and port 2 is an RS–422/485 port. One port is designated the programming port, to be used to communicate with UPTime. The other port may be configured to be a CCM port, communicating to a host computer using the CCM protocol. See the section on Serial Communications for more information on using the serial ports.

There are three LED indicators located at the top front edge of the Diagnostic Processor. The top LED is the status indicator for the Diagnostic Processor. During power–up this LED flashes while the Diagnostic Processor is running its diagnostic checks. If this LED is off, either the power is off, there is some hardware malfunction of the Diagnostic Processor, or there is no CPU present in the Series 90–30 PLC system. When the LED is on, the Diagnostic Processor is functioning normally.

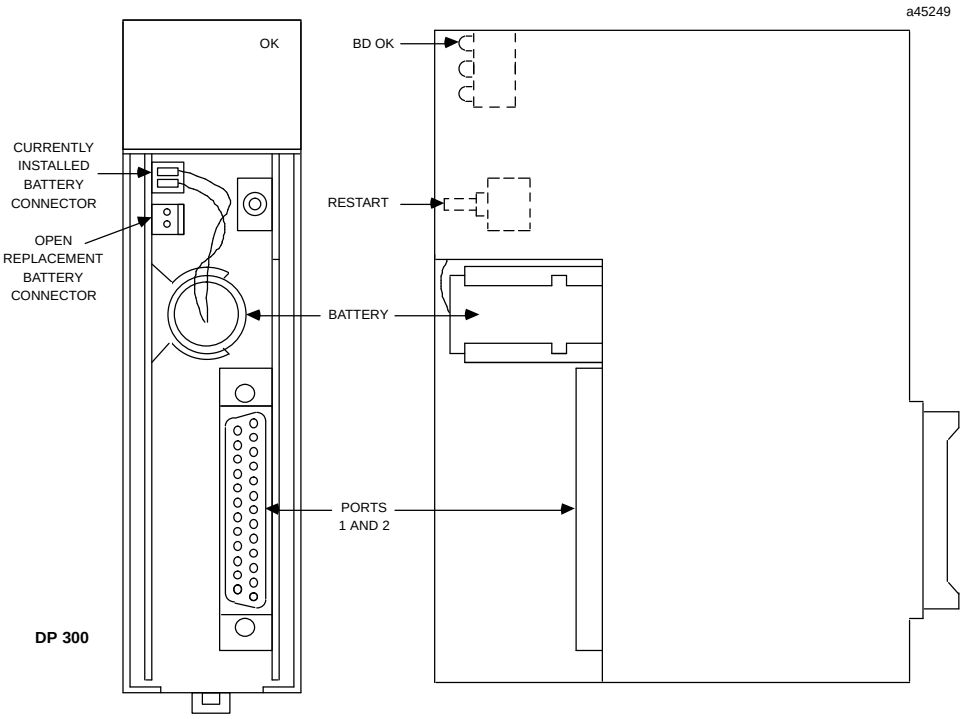


Figure 5-2. Series 90–30 Diagnostic Processor

The Diagnostic Processor comes with a battery to maintain memory when power is removed. This is a lithium battery which is installed as shown in the Diagnostic Processor drawings. When the battery reaches a low charge, this condition is reported in the PLC fault table.

Each Diagnostic Processor also comes with a Reset switch. When the reset button is pressed for less than 5 seconds, the Diagnostic Processor behaves as if power was lost momentarily. The program begins operation if the Diagnostic Processor is set up for auto–r un operations, otherwise the Diagnostic Processor is in HALT mode. If the reset switch is pressed for more than 5 seconds, then the Diagnostic Processor memory is initialized and the program must be reloaded.

Operational Description

The Diagnostic Processor accesses areas in the CPU memory for I/O references and Register values. The Diagnostic Processor and the CPU communicate this information over the PLC backplane.

When a Ladder Logic control program and a State Logic diagnostic strategy are running at the same time, the State Logic and Ladder Logic programs should not be controlling the same outputs. The Diagnostic Processor cannot control an output that is being controlled by the CPU. The 90–30 CPU always takes precedence when both processors are controlling the same outputs.

All of the outputs used in the State Logic program should be selected to be contiguous if system response time is important. Outputs being non–contiguous cause the scan rate to increase significantly.

When changing outputs, the Diagnostic Processor writes to a byte of I/O bits at a time, so the Diagnostic Processor actually controls eight ouputs at a time. Therefore, if some outputs of a byte are not named in the Diagnostic Processor program, UPTime issues a warning that the other outputs in that byte are changed when the program executes.

Installation and Maintenance

This section describes how to install the Diagnostic Processor into a Series 90–30 PLC rack. There is also a section describing maintenance considerations.

Selecting the Right Slot

The Diagnostic Processor can be installed in any unused slot in the CPU rack of the Series 90–30 PLC if the following rules are followed:

1. Configuration created by Logicmaster must match the physical location of the modules. Configuration faults are logged in the PLC fault table.
2. For the 90–30 Diagnostic Processor all the slots between the Diagnostic Processor and CPU must be occupied. If any slots between are empty the Diagnostic Processor cannot communicate with the CPU. The Diagnostic Processor must be in the main rack with the CPU.

Inserting the Diagnostic Processor

Follow these steps to insert the Diagnostic Processor into the Series 90–30 rack:

1. Power down the Series 90–30 PLC system
2. Locate the desired rack and slot.
3. Slide the 90–30 Diagnostic Processor completely into the slot.
4. Press down firmly to lock the module in place, but do not use excessive force.
5. Power up the PLC rack. The Status LED flashes during power–up diagnostics. The LED comes on steady when the Diagnostic Processor is ready for operations.

Configuration

Use the Logicmaster 90–30 configuration software to add the Diagnostic Processor to the Series 90–30 I/O configuration. This software is used to describe the modules present in the PLC racks. Rack and slot location and other features for each module are entered by completing setup screens that describe the modules in a rack.

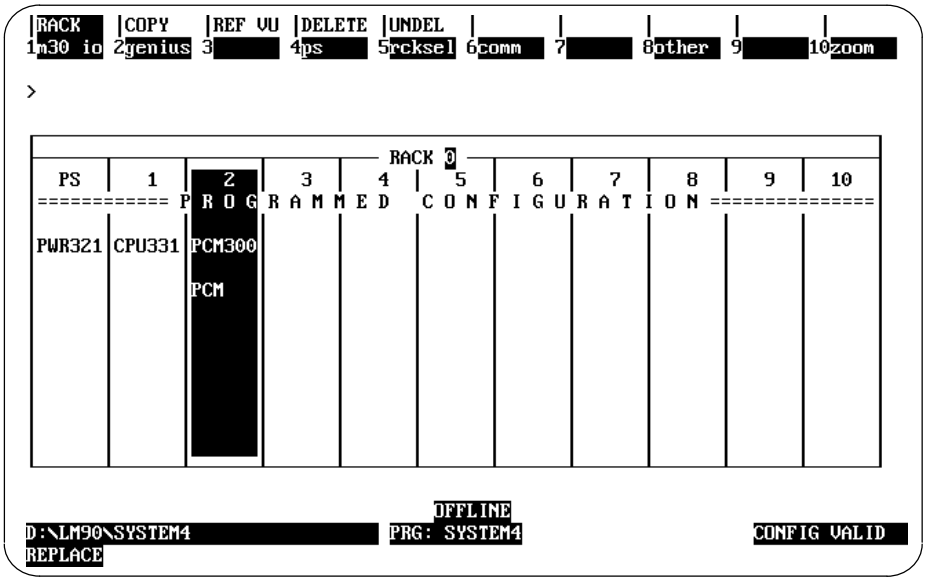


Figure 5-3. Sample Logicmaster Configuration Screen

From the main menu of the Logicmaster 90–30 configuration software, press I/O <F1>. The screen displays a representation of the modules in a rack. To add a Diagnostic Processor to the configuration, highlight the desired slot, then press Other <F8> and then PCM <F1>.

Now press Zoom <F10> to view the current configuration. Press <Enter> to enter the highlighted catalog number and display the PCM detail screen.

Set the Configuration Mode to PCM CFG for the 90–30 Diagnostic Processor. First highlight the Config Mode option and repeatedly press the <Tab> key until PCM CFG is displayed on the screen. The serial ports are under program control and the parameters are initialized by the Diagnostic Processor.

Press the <Esc> key to save the configuration and return to the rack display. The display should now show a PCM in the correct slot. Send the configuration to the PLC CPU and the configuration is complete.

During program execution the CPU must be in run mode with the outputs enabled. Use LM90–30 to set the status of the CPU before executing the State Logic program.

Battery

The Diagnostic Processor comes with a 3 volt lithium battery (IC697ACC701) to maintain memory through a power cycle. If the battery charge becomes low, a fault is set in the fault table. These faults can be monitored by the State Logic program.

To replace the battery, connect the new battery to the extra set of battery connections then disconnect the old battery. A Product Safety Data Sheet for the battery is available. Order from GE Fanuc using number GFK–0633.

Troubleshooting

This chapter provides procedures for diagnosing Diagnostic Processor problems. If these procedures do not solve the problem, contact the GE Fanuc Hotline (1–800–828–5747) for assistance.

Status LED is not ON Steady

1. Check that power is supplied to the I/O rack housing the Diagnostic Processor. Try removing and reinstalling the Diagnostic Processor.
2. Cycle power to the Diagnostic Processor, then press the reset button for more than 5 seconds.
3. Turn the power OFF and disconnect the battery and short the Diagnostic Processor battery terminal connection points to clear the Diagnostic Processor. Reconnect the battery, turn ON power again, and reset the Diagnostic Processor.
4. Check that the CPU is functioning properly by checking its “OK” LED.
5. Check that there are no empty slots between the CPU and the Diagnostic Processor. If there are empty slots, the Status LED blinks continuously.
6. If the Status LED is still not ON, try to download a program from UPTime. If you can connect with the Diagnostic Processor and download a program, then the Status LED is faulty.
7. If you get a message that UPTime cannot connect to the processor, then check the fault table in the CPU using Logicmaster 90. If there is a fault “Bad or missing module”, then the Diagnostic Processor is faulty and must be returned for repairs.
8. If there is no fault then contact the GE Fanuc Hotline for assistance.

Reset Blinks Port 1 or Port 2 LED

If the Port 1 and Port 2 LEDs have an alternating blinking pattern, then the Logicmaster 90–30 or CPU firmware is out of date with the Diagnostic Processor firmware. Get updated versions of the Logicmaster software and/or the CPU firmware

Serial Communication Problems

This section is split into two parts, communication problems to UPTime and problems with other serial devices.

Communications Problems with UPTime

1. Check that the serial cable used conforms to one of the types specified for communications to the Diagnostic Processor ports. Check that the cable is firmly secured at both ends.
2. When there is a communications problem, a message indicating problems connecting to the processor is displayed together with some options. Select the "Change Host Comm Port Settings" option to check the computer port being used and the baud rate. The default Diagnostic Processor baud rate is 19,200.
3. The Diagnostic Processor programming port may have been changed. UPTime must be connected to the designated programming port. The default programming port is Port 1 but can be changed by UPTime. Make sure the serial cable is connected to the current programming port.
4. The Diagnostic Processor serial port configuration may have been changed. These parameters can only be changed by Statements using the Set_Commport keyword in the State Logic program. Reset the Diagnostic Processor to return the Diagnostic Processor serial port parameters to their default state. If the program is set up to start running automatically and the program continues to change the serial port parameters after the reset switch is pressed, turn off the power to the Diagnostic Processor, remove the battery, and short the Diagnostic Processor battery leads to reset the Diagnostic Processor.
5. Port 2 of the 90–30 Diagnostic Processor is an RS–422/485 port and Port 1 is an RS–232 port. Make sure that the host computer serial port setup matches the Diagnostic Processor port for RS–232–RS–422/485 option.
6. If there is still no communication between the Diagnostic Processor and UPTime then call the GE Fanuc Hotline for assistance.

Communications Problems with Another Serial Device

1. Check that the serial cable used conforms to one of the types specified for communications to the Diagnostic Processor ports. Check that the cable is firmly secured at both ends.
2. Make sure that the serial port parameters of the serial device match the settings of the Diagnostic Processor. The default settings of the Diagnostic Processor are 19,200 baud rate, 8 data bits, 1 stop bits, and no parity.
3. Make sure the RS–422/485–RS–232 standards are the same for Diagnostic Processor and the serial device.
4. When connecting a device to the programming port, make sure that you exit UPTime normally before disconnecting from the Diagnostic Processor.
5. If there are still problems contact the GE Fanuc Hotline for assistance.

Serial Communications

Serial communications with the Diagnostic Processor are provided through one of the two serial ports. Each of these ports can send and receive serial data independent from the other. Port 1 is an RS–232 port and port 2 is an RS–422/485 port. There is only one 25–pin connector on the Diagnostic Processor which provides access to the connections for both ports. The Diagnostic Processor comes with a WYE cable which provides one 25–pin connector for each port.

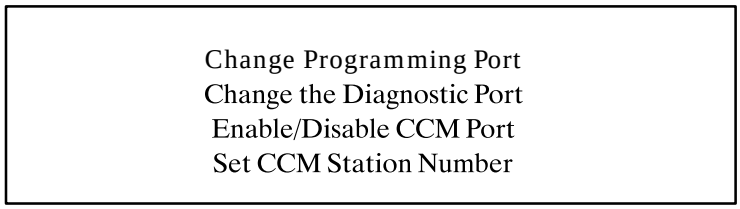
UPTime must communicate with the Diagnostic Processor through the programming port. Either port can be the programming port, but port 1 is the default programming port. When UPTime is not connected to the Diagnostic Processor, the programming port can be used to communicate with some other serial device. To properly set the programming port to communicate with another device, exit UPTime normally before disconnecting from the Diagnostic Processor.

Either of these ports may also be a CCM port. The CCM port uses the GE Fanuc CCM2 protocol for all communications. This protocol is used for connecting the Diagnostic Processor with a host computer which collects data and changes data in the Diagnostic Processor. Typical CCM uses are connecting to a graphical user interface program such as CIMPLICITY or for custom host interaction with the Diagnostic Processor.

The Diagnostic Processor is always a slave to the host, ie. all communications are initiated by the host computer. The Diagnostic Processor may be one of several devices on a CCM network. The CCM port is always the port that is not designated as the programming port. The CCM port is a normal RS–232 or RS–422/485 port when CCM communications are not enabled.

Serial Port Setup Using UPTime Debug Mode

The options available in the UPTime menus to change the configuration of the Diagnostic Processor serial ports are displayed below.



Serial Options Set by UPTime

Each of these options is set in the State Engine Configuration Menu selected from the Debug Mode Project Menu in UPTime.

Designating the Programming Port

The programming port is the State Processor serial port to which UPTime is connected. If UPTime is not connected to the programming port, the Diagnostic Processor cannot communicate to these software packages.

Port one is the default programming port. To change the programming port select the “Diagnostic Processor Configuration” option from the Debug Mode PROJECT menu and make the selection from the provided form.

Designating the Diagnostic Port

The Diagnostic Port is the serial port where all messages are sent from Diagnostics that have been tripped. The Diagnostic Port and the programming port can be the same port, but the Diagnostic port and the CCM port cannot be the same when CCM protocol is enabled.

Port one is the default Diagnostic Port. To change the Diagnostic Port select the “Diagnostic Processor Configuration” option from the Debug Mode PROJECT menu and then make the selection from the provided form.

CCM2 Protocol Serial Port

CCM2 protocol is a standard open communications protocol defined by GE Fanuc. The Diagnostic Processor acts as a slave in a master–to–slave architecture. The remote master computer must poll to retrieve data from the Diagnostic Processor. All communications are initiated by the host.

The CCM2 protocol defines the message structure, framing, error checking and handling, and timing for all message types. At the lower physical level, the serial port of the Diagnostic Processor is electrically RS–232 and can be configured for any Baud rate, parity, and stop bits desired via a command in the program executing in the Diagnostic Processor. Use the Set_Commport keyword in the UPTime program followed by the parameters, to configure the port so that CCM2 communications port electrically matches the serial port of the master computer.

Either of the 2 serial ports can be a CCM2 Communications port. The CCM port is always the port that is not the programming port.

Enabling CCM2 Communication

Both serial ports are normally normal serial ports using no communications protocol. To use the CCM2 communications protocol on the CCM port, the CCM2 Communications must be enabled. To enable CCM2 Communications select the “Diagnostic Processor Configuration” option from the Debug Mode PROJECT menu.

There are other CCM2 options on this menu. The CCM2 options available are:

Enable CCM Protocol Port
Disable CCM Protocol Port
Set CCM Protocol Station Address

The Diagnostic Processor using CCM2 only recognizes and responds to messages sent to its address, which may be any number from 1 to 89. The default address will be 1 and the User should assign a Station Address to match the number the Remote Master expects. The Station Address should be validated or assigned before CCM2 protocol communications are enabled.

If the “Disable” function is chosen, all CCM2 communications are immediately discontinued, even if in the middle of a message, and all inputs on that serial port are taken to be ASCII character inputs that will be handled as appropriate to the current Diagnostic Processor program execution.

Programmable Setup Options

On power up the serial port parameters are set to their default settings. These settings may be changed by using the Set_Commport programming keyword in the diagnostic program. The Set_Commport keyword and parameters must be executed by the diagnostic program to change the serial port settings. Just changing the parameters in the UPTime serial communications setup form does not automatically change these settings.

After the Diagnostic Processor serial port parameters are set, UPTime automatically enters the Set_Commport instruction into the program when the following selections are made. Select the LIST option from the Sequence editor menu, press <F3>; then select the Communication Ports option. Highlight the appropriate port and press <Enter>. UPTime prompts whether to enter the port name only or enter reconfiguration data. After selecting Enter Reconfiguration Data, the Set_Commport keyword plus the instructions to change the serial port parameters are entered into the UPTime program.

These options affect the Diagnostic Processor serial port configuration only, not the UPTime host computer serial port setup. The option to change the UPTime host computer serial port configuration is available only when there is some problem connecting UPTime to the Diagnostic Processor. The only options which can be changed on the UPTime serial port configuration are selecting COM1 or COM2 and changing the baud rate.

Parameter	Settings
Baud Rate	19.2K 1200 9600 300 4800 2400
Data Bits	8, 7 or 6
Parity	Even, Odd, None
Stop Bits	2, 1.5, or 1
Respond to Backspace	Enabled Disabled
Stop Transmit on Receive	Enabled Disabled
Receiver Always On	Enabled Disabled
Line Feed After Carriage Return	Enabled Disabled
Auto Echo	Enabled Disabled
XON/XOFF Protocol	Enabled Disabled
End of Message Carriage	Hex 0D (Character Return)

Table 5-1. Serial Port Parameters

Serial Cables

The Diagnostic Processor is designed to work with two GE Fanuc standard serial cables. One cable is the Mini Converter Kit that comes with the Diagnostic Processor. The other is the PCM to IBM–PC cable. Use the RS–422 / 485 serial cables to connect to Port 2, if RS–422 / 485 communications are used.

Mini Converter Kit

UPTime comes with a serial cable kit that can be used to both connect the CPU with Logicmaster and to connect the Diagnostic Processor with UPTime running on an IBM PC. This cable kit comes with three adapters:

9–pin Male to 15–pin Male	CPU Port	HE693SNP232
9–pin Male to 25–pin Male	Diagnostic Processor Ports 1 or 2	AD232/1–2
9–pin Male to 25–pin Female	IBM PC COM 2	HE693XTADP

Table 5-2. Mini Converterr Kit Adapters

The cable provided in the kit is terminated with two 9–pin connectors. One of the adapters connects the 9–pin serial cable to the 15–pin serial port on the CPU (HE693SNP232), another adapter connects the serial cable to the 25–pin female Port 1 on the Diagnostic Processor (AD232/1–2), and the third adapter can be used to connect to a 25–pin port on the IBM computer running Logicmaster 90 or UPTime (HE693XTADP).

Other Cable Options

Other cables that work with the Diagnostic Processor are the IBM–AT cable, IC697CBL702, and the Workmaster II or PS/2 cable, IC697CBL705. The connector on the Diagnostic Processor end can be made to be a right angle connector, so that the Diagnostic Processor front cover can be closed and the cable brought out the side of the Diagnostic Processor or through the bottom of the Series 90–30 chassis.

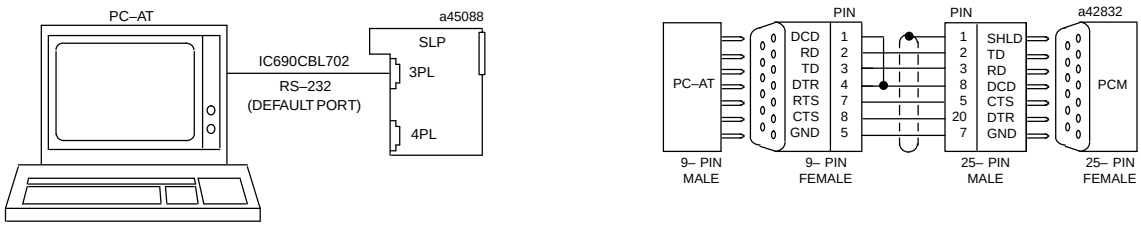


Figure 5-4. IBM PC–AT to Diagnostic Processor Cable

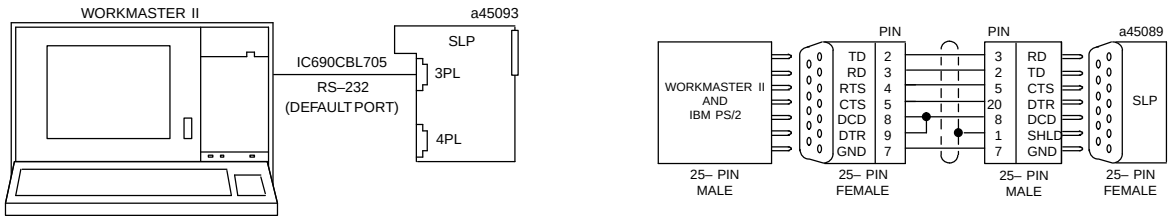


Figure 5-5. Workmaster II or PS/2 to Diagnostic Processor Cable

Custom Made Cables

If you want to make your own cable to connect to some serial device, use the drawing of the Diagnostic Processor pin assignments and the cable connection drawings in the previous section for the required connections. Hardware flow control lines must be used when communicating with the Diagnostic Processor using RS–232. The flow control lines are not used for RS–422 / 485 communications.

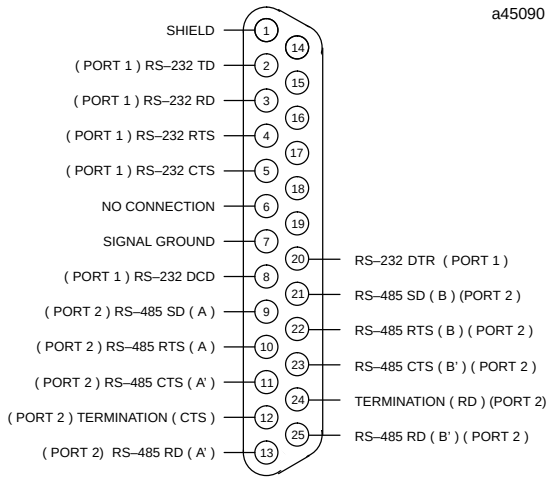


Figure 5-6. Serial Port Assignments for Series 90–30 Diagnostic Processor

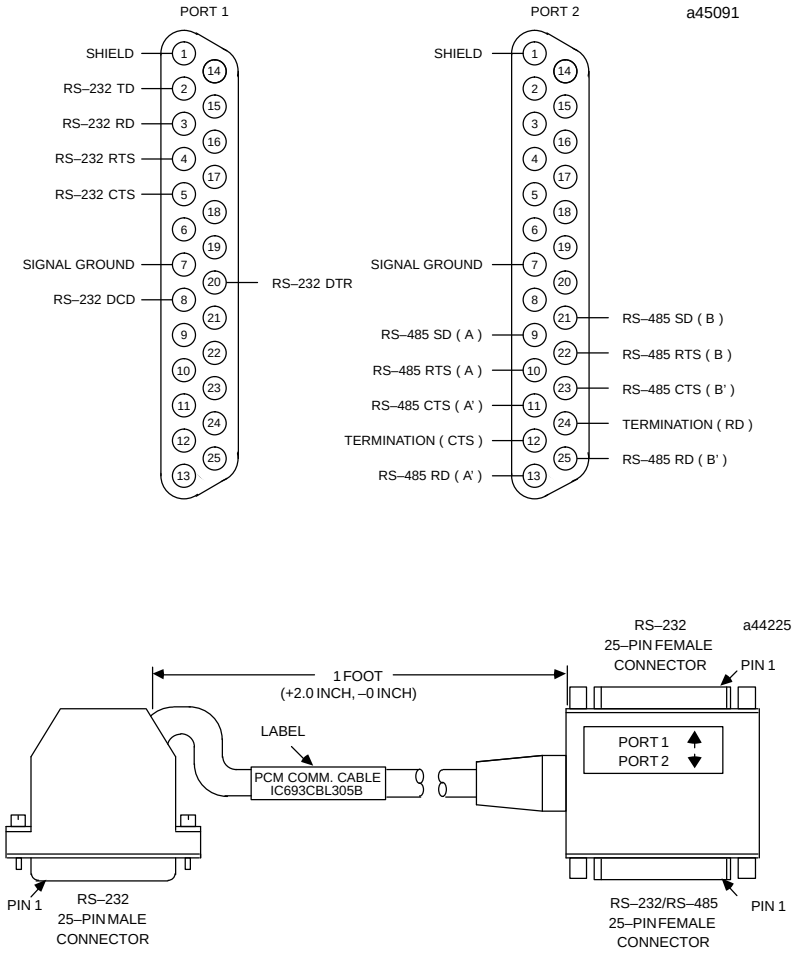


Figure 5-7. WYE Cable Connections for Series 90–30 Diagnostic Processor

Note

In the drawing above, the SD (Send Data) and RD (Receive Data) connections are the same as TXD and RXD used in other terminologies. (A) and (B) are the same as – and +. A' and B' denote inputs, and A and B denote outputs.

Diagnostic Processor/CPU Interface

UPTime programs execute in the Diagnostic Processor. The Diagnostic Processor is a module with the Diagnostic Processor operating system imbedded that plugs into one of the slots of a Series 90–30 PLC chassis. The CPU is the direct interface to the modules plugged into other slots in the system. The Diagnostic Processor accesses the CPU memory to interact with the system I/O. Communication between the Diagnostic Processor and CPU is across the Series 90–30 backplane.

This section describes the CPU memory accessed by the Diagnostic Processor, the different communication methods, and how to set up your system.

CPU Memory Accessed by the Diagnostic Processor

UPTime programs access discrete and register types of memory stored in the CPU. Discrete types are bit oriented being either ON or OFF (1 or 0). The discrete types accessed by the Diagnostic Processor are %I, %Q, %M, %T, %G, %S, %SA, %SB, %SC. The following table describes each of the discrete memory types.

Memory Reference	Reference Description
%I	Real world discrete inputs. These locations store the status of the inputs following the last scan of the inputs by the CPU of the system.
%Q	Real world discrete outputs. These locations store the status of the outputs. The CPU updates the real world outputs during its normal execution cycle.
%M	Internal discrete references. These references are used to store intermediate conditions used in the logic decisions in the program.
%T	Temporary internal references.
%G	Global Data references. This data can be shared between multiple devices using the Genius Communications Module to communicate over a Genius communications bus.
%S, %SA, %SB, %SC	System Status references. References system fault information.

Table 5-3. Discrete Memory Types

Register memory types store a value using sixteen bits of memory space. The types of register storage are R–Registers (%R), analog inputs (%AI), and analog outputs (%AQ). These register types are described in the following table.

Table 5-4. Register Memory Types

%R	Stores a value in sixteen bits. State Logic Program Register Variables access these memory locations. A Register Variable can be either an integer or floating point value. Floating point values use 2 %R registers. The write and read terms can also access these locations for character values.
%AI	Real world analog inputs. These locations store the value of the inputs following the last scan of analog values.
%AQ	Real world analog outputs. The CPU transfers these values to the analog outputs during the normal execution cycle.

State Logic program references to CPU memory locations are user selected names that are identified by the memory type and a number representing the storage location, ie. Q14.

Input/Output Memory Designation

Every CPU memory location used in the Diagnostic Processor program is specified to be either an input or an output as viewed from the Diagnostic Processor in respect to the CPU. Output memory locations can be changed (WRITTEN) or used (READ) by a State Logic program, but inputs can only be read. Therefore, if the State Logic program is going to change a memory location, it must be defined as an output, but if the Ladder Logic program in the CPU is going to change the memory location, it must be defined as an input for the State Logic program. The CPU and Diagnostic Processor cannot both change the same memory locations. The CPU always takes precedence if both processors attempt to change a memory location.

Any communication between a State Logic program and a Ladder Logic program requires that an output to one program must be an input to the other. Therefore %Q's which are outputs to the CPU may be used as inputs to the State Logic program logic and %I's, normally inputs to the CPU may be changed by the Diagnostic Processor and thus are actually outputs to the State Logic program. The Diagnostic Processor cannot change %I memory locations that are configured to input modules.

An important consideration in assigning inputs and outputs is to try to have all the outputs together in a contiguous block. Any non–contiguous outputs has an adverse affect on system response times. UPTime issues a warning during translation, if there are any non–contiguous outputs.

Diagnostic Processor Memory Capacities

The Diagnostic Processor can access the full range of 90–30 CPU memory and any point can be an input or output to the Diagnostic Processor.

Table 5-5. 90-30 Diagnostic Processor Discrete and Register Capacities

Type	Capacity
%I	512
%Q	512
%AI	128
%AQ	64
%T	256
%M	1024
%G	1280
%S	32
%SA	32
%SB	32
%SC	32
%R	2048

Clearing Outputs at Power Up

Much of the discrete memory in the CPU is retentive, meaning that its value is saved when power is lost, because the memory is battery backed. These discrete locations must be cleared when power is first applied to the system, otherwise outputs that were ON when power was lost may go ON momentarily when power comes back on again.

This situation is not an issue for most systems since critical control outputs are usually controlled by the CPU ladder logic program. If remaining ON at power on is a problem, add a rung using the FST_SCN contact to clear the outputs controlled by the Diagnostic Processor.

Diagnostic Processor Scan Considerations

The Diagnostic Processor operating system is a scanning system. There are several steps that are executed in sequence to complete a scan. These steps are continuously repeated while the Diagnostic Processor is in operation. One of the steps executed when the Diagnostic Processor is running a program is the scan of the program. The first section below discusses all of the steps executed for each scan cycle and the next section discusses the program scan step.

Steps of the Scan Cycle

These are the steps that the Diagnostic Processor executes each scan cycle:

1. **Character Input Service**

   The Diagnostic Processor retrieves messages that have completely received from the serial ports and dispatches them. These messages may be inputs to READ terms or UPTime commands.

2. **Real Time Clock**

   The Diagnostic Processor updates the current time registers from the Real Time Clock.

3. **Transfer Digital I/O**

   Inputs are written from the CPU to the Diagnostic Processor Input Image Table. Outputs written from Diagnostic Processor Output Image Table to PCIM and then the output image table is cleared.

4. **Transfer Analog I/O**:

   Analog output values are sent to the CPU and input values are read from the CPU when the update time for a block of analog channels has elapsed. High speed analog channels are updated every scan.

5. **Program is Scanned**

   If there is any programming, the active Step of every Task is scanned. The Diagnostic Processor looks to the Input Image Table for tests of digital inputs. Digital outputs that are to be ON are set in the output image table, outputs test ON only when verification that the output is ON is received back from the block. Variable values are updated immediately and flags set ON are not tested as true until the next scan.

6. **Diagnostics are Checked**

   If there are any Diagnostics, the triggering setpoints are checked against the current values. If any Diagnostics trigger, the specified action is initiated.

7. **Service CCM Messages**

   If there is a completed message received from the CCM serial port, that command is executed.

8. **PID Update**

   PID Loops are updated when the update time period for the loop is completed.

Program Scan

The program scan starts at the start of the program, scanning the active Step of every Task. During program execution there is always one and only one Step active in each Task. The operating system completes a scan of the program tens and maybe hundreds of times every second.

During the scan of the active Step of a Task, each Statement of the Step is scanned in the order that it appears. Keep in mind that a Statement is a series of Terms terminated by a period (.).

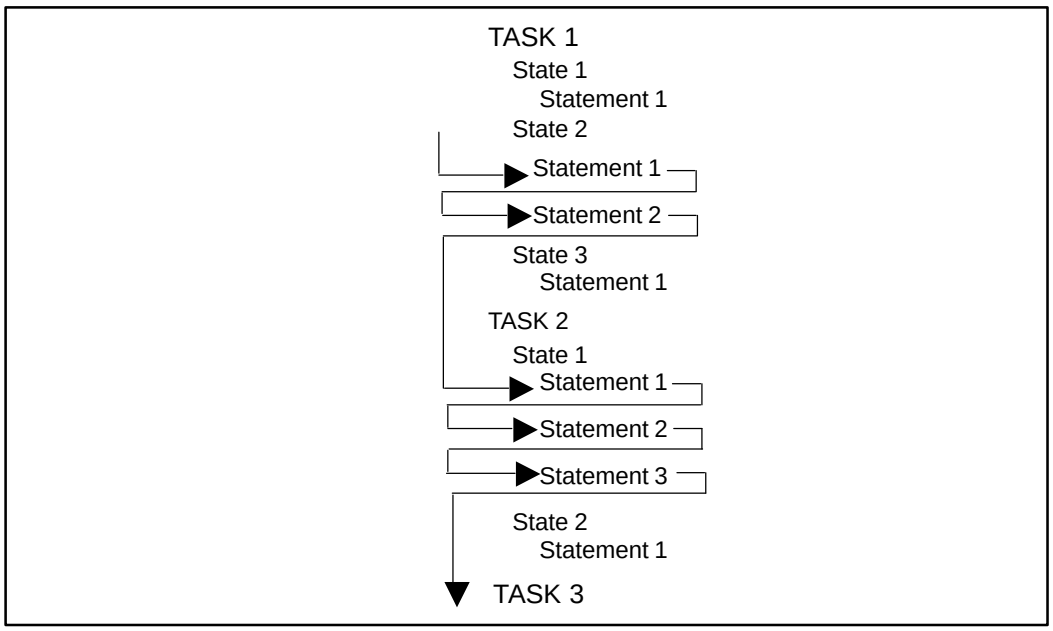


Figure 5-8. Program Scan

The actions specified by Functional Terms are executed when the Term is scanned. Each Statement must have at least one Functional Term, Conditional Terms are optional. If there are no Conditional Terms in a Statement, the Functional Terms are always executed during each scan. When Conditional Terms accompany Functional Terms in a Statement, the Functional Term is executed when all of the Conditional Terms are satisfied. There are four types of conditional Terms (see the reference section).

Conditional Terms are satisfied as follows:

1. Read – When valid data is received at the appropriate channel.
2. Wait – When the elapsed time that the Step has been active exceeds the stated value.
3. If – When the conditional expression is TRUE.
4. For / Until – When the conditional expression is FALSE.

To understand how Statements are scanned, assume that the Statement Conditional Terms precede the Functional Terms and that the scan proceeds from left to right.

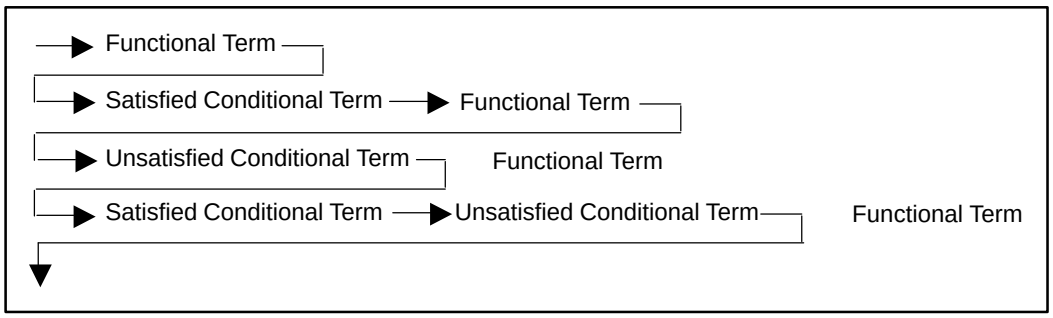


Figure 5-9. Statement Scan

The Statements of a Step are executed in the order that they are written into the program. Functional Terms of Statements with no Conditional Terms are always executed. Conditional Terms in Statements control whether or not the Functional Terms in those Statements are executed. If all of the Conditional Terms are satisfied, the Functional Terms are executed. If any of the Conditional Terms are not satisfied, none of the Functional Terms are executed.

The Statements are executed one at a time. In this manner every Statement of the active Step is evaluated.

There are two types of Functional Terms that can prevent the execution of the rest of the Statements in a Step. One is the Halt command which stops program execution. The other is the "go to ..." command, which immediately causes another Step to become the active Step.

For example:

If Start_Pushbutton is pushed, go to Start_Up Step.

causes the Start_Up Step to become the active Step when the process input represented by Start_Pushbutton is true. All Terms or Statements following this Statement are not executed, and at the next Diagnostic Processor cycle, the scan of this Task starts at the first Statement of the Start_Up Step.

During the program scan any changes to variable and analog values are made immediately. Therefore, a variable change in one Task is visible by the rest of the program during the same scan. On the other hand, digital I/O and Flag conditions are made at the end of the scan. Therefore, if one Task makes a change to the condition of a digital output or Flag, the condition cannot be tested by another Task until the next scan through the program.

Analog Scaling and Update Rates

Analog modules transfer raw numerical integer values to PLCs. Each increment represents a step on the full range for the module. Some normal ranges of raw analog values are 0 to 4095 or –32768 to 32767. These raw values represent some real world values.

For example, a scale might have a range of 0 to 50 pounds actually sending an analog signal of 0 – 5 volts to an analog module. The module converts the analog signal to an integer value in the range of 0 to 4095. If the scale detects 25 pounds, it sends a 2.5 volt signal to the analog module which converts the voltage to a number, 2047.

You can have UPTime do the conversions of raw analog values to real world engineering units for you. After each analog channel is defined, UPTime asks if you want to scale this channel. If the answer is YES, a form is displayed which accepts four values which specify how the scaling is done.

The D/A and A/D values are the raw values and the engineering units are the converted values. Two scaling points where the conversion of A/D or D/A value to engineering value is known are specified. In the example of the scale the low scaling point values are 0 for the A/D or D/A and 0 for the engineering units. The high point is 4095 for the A/D or D/A and 50 for the engineering units. After these points are specified, the Diagnostic Processor can convert any other raw value to the correct real world value so that a raw value of 2047 is converted to 25 for use in the State Logic program.

When analog channels are scaled, the Diagnostic Processor converts the raw analog values to floating point numbers. Since floating point operations use a lot of time, the analog values and their conversion to floating point values are not updated every scan. Conversely, unscaled analog channels are always updated every scan.

The Diagnostic Processor uses a default scheme of scanning the analog channels at 1/10 second intervals. To inspect or change this scheme choose the "Series 90 Scan Rates" from the DEFINE menu.

Use this option to change UPTime's default analog scanning scheme. A form is provided to show how the analog channels are scanned. There are 8 columns and 10 rows. The entry in each one of the 80 locations represents a block of 8 analog channels.

Each 1/10 of a second a collection of analog channels are scanned. The entries in each row represent the blocks of channels that are scanned for that 1/10 second time interval. There are 8 columns each representing a block of 8 channels, so that up to 84 analog channels can be scanned every 1/10 second. If a block is to be scanned every interval or 10 times a second, then that block must appear once in every row of the form.

There are 4 data entry locations at the top of the form. To enter values into the form, fill in these data entry locations for the Time Entry (Row), Block Entry (Column), Type and Number. Item is entered into the form when the Number value is entered.

Other Diagnostic Processor Setup Options

Run–time Error Setup

The are several errors which may occur during execution of the State Logic program such as divide by zero or integer overflow or perform function errors. Run–time errors are divided into critical and non–critical categories.

The default response of the Diagnostic Processor is to halt the program when a critical error occurs and to continue program execution after a non–critical error. These responses may be changed through options on the Debug Mode PROJECT menu.

Automatically Start Program Execution

By default the Diagnostic Processor powers up in the Halted mode. The Diagnostic Processor can be set up to automatically start running the program when power comes on.

To set up the Diagnostic Processor to automatically execute the program on power up, select the "Diagnostic Processor Configuration" option from the Debug Mode PROJECT menu. When set to automatically run the program, the previous settings of other Diagnostic Processor setup options are also invoked on power up, including programming port, CCM Enables, CCM Station Number, Error Response Setup, and Simulation Mode Status.

Simulation Mode

The Diagnostic Processor can be put in Simulation Mode through the option on the Debug Mode PROJECT menu. When this mode is active the Diagnostic Processor is completely disconnected from the real world I/O; the Diagnostic Processor does not change any outputs and does not look at any of the inputs.

This mode is useful to debug a new program before connecting the I/O or for troubleshooting a machine problem without interacting with I/O points. The Diagnostic Processor set to this mode executes the program.

Make the program execute as desired by using the CHANGE option to change current Steps of the Tasks, variable values, and analog values. Use the FORCE option to change I/O or flag status. Changes can also be made by using the VIEW option to display the English program. Put the cursor on a name to display the current value, then a new value may be entered.

Setting the System Clock

The CPU has a clock that maintains the current month, day, day of the week, hour, minute, and second. These values are always available through the System Variables Month, Day, Day_of_Week, Hour, Minute, and Second.

The clock cannot be changed from the Diagnostic Processor. The time must be changed in the CPU using Logicmaster 90. After the time has been changed in the CPU, restart the Diagnostic Processor to transfer the new time to the Diagnostic Processor.

Chapter 6 UPTime Menus and Keys

This chapter describes the UPTime menu options and keystroke functions. Only those menu options that are not immediately apparent are explained. The keystroke section lists general key usage, plus hot keys and miscellaneous key functions.

UPTime Menu System

This section goes through the UPTime menu system, explaining aspects of the menus that are not immediately apparent from the display. First discused are the Create Mode menus, then the Debug Mode, and finally the Setup menu is explained.

Create Mode

The UPTime Create Mode is a highly specialized editor and compiler designed to create a Diagnostic Strategy composed of both Fill–in–the–Blank Diagnostics and State Logic programming. This section explains all of the menu selections available in the UPTime Create Mode. The top level Create Mode menu has options of DIAGNOSTICS, SEQUENCES, VARIABLES, DEFINE, PROJECT, and QUIT.

Diagnostics

This option is the gateway to creating all Diagnostics. The next menu displays all of the Diagnostic types, Analog, Digital, Variable, and Device. From these selections a list of the particular type is displayed. When the list is displayed it may be changed by adding other Diagnostics of that type, editing an existing Diagnostic, or deleting a Diagnostic. See the bottom banner of the display for the keystrokes to accomplish these functions when the list is displayed.

Sequences

This option is the gateway to the creation of all Sequences. The menu displayed when this option is chosen has two selections, Diagnostic Sequences and Instructional Sequences. When one of these selections is chosen a list of all Sequence names of that type are displayed. From this list Sequences can be added, deleted, edited, renamed, and a list of the Diagnostics that start a Triggered Sequence can be displayed.

Sequence Editor

When adding or editing a Sequence, the UPTime editor is invoked. The editor is used to edit one Sequence at a time.

The editor is a specialized word processor that is designed to create State Logic Sequences. The editor menu is invoked by pressing <F3>. This menu provides many word processing functions and specialized operations for creating State Logic programming.

Add

All of the ADD options are used to add program lines to the English text through the use of fill–in–the–blanks forms. The ADD options always enter text into the program. This text is no different from text entered from the keyboard and may be edited as other text is edited.

The Add option is the gateway for creating the Perform functions. As with the other Add operations, a form is provided and when completed the text for the function call is inserted into the program at the current cursor location.

Sequences

This option is used to change the Sequence being edited or just view other Sequence names. Both Instructional and Diagnostic Sequences are accessed.

List

The List option is used to view and change lists of user defined names for I/O points, variables, PID Loops, Communication Ports, and Internal Flags. The predefined lists of Keywords, Filler Words, Math Functions, PID Parameters and System Time Variables are also available from the List option. All of the lists may be modified except the Math Functions, Time Variables, and PID Parameters. The List option is also useful for selecting a name and entering it into the program at the current cursor location.

When the list option is selected, the LIST menu is displayed showing all of the categories of names. After selecting a name category, the bottom banner displays the keys available to perform different functions with the LIST (<Ins> Add to List, <Del> Delete from List, <Enter> Write the Name into the Program, or <–>> Edit the Name and/or Configuration Data).

All UPTime lists except the keyword list are displayed in alphabetical order. To find a entry on the list start entering the letters of the entry. After a letter is entered, UPTime finds the first entry starting with that letter. When the next letter is entered, UPTime highlights the first entry starting with those two letters. Use this procedure to help locate items on the list.

List – Communication Ports

The option on the LIST menu that requires some explanation is the Communications Ports option. Use this option to change communication port names and parameters, to enter a Communications Port Name into the program at the current cursor location, or to enter a Statement into the program that changes the serial port parameter settings when the Statement is executed.

The program Statement that changes the serial port parameters uses the Set_Commport keyword. In order for the communications port parameters to be changed, the Statement using the Set_Commport keyword must be executed by the program. To enter the Statement to change the serial port parameters press the right arrow key to edit the parameters in a form. When finished press <Enter>, then select the "Insert Reconfiguration Data for Port" option. UPTime now enters the instructions into the program. Remember that just changing the parameter form does not change the port setup, but the Set_Commport Statement must be executed in the program.

Define

There are four Define menu options that define a word in the program, search for all undefined words, configure how often analog channels are updated, and display the serial port configurations. The Communications port option enables the serial port names to be changed, but does not allow the configuration setup to be changed. The Series 90 I/O Scan Rates allows the update times for analog channels to be changed.

Find

These functions move the cursor around in the English text quickly and allow single and global replacements of text. The start of another Task and the location of the last translator error can also be found with these functions.

Text

This option provides typical word processing block functions. When selected this option allows a section of text to be highlighted. When the block is selected, a list of four functions is displayed to manipulate the block; Copy, Move, Remove, and Copy Several Times. To do cut and paste operations select a block and remove it, then select the block option and without highlighting any text hit enter and then the Copy option. The Removed text is entered at the cursor location.

Return

This option just returns to the top Create Mode level with the menu displayed.

Variables

This option is the gateway for creation and maintenance of most of the State Logic variables, including Analog Channels, Digital Points, Register Variables, Numeric Variables, and Digital Devices. Other sections describe the use and creation of these variables.

Define

This option is primarily used to change the Default Message text. UPTime comes with Default Messages already defined, but the message for each type of Diagnostic Action can be changed through this menu option. Select the option and then the type of Diagnostic Action from the list displayed. The current message is displayed and can be edited in the provided window. The message can be up to 80 characters long.

Project

This menu contains project management functions needed to manipulate the files that make up a project. There are functions that act upon the entire project such as retrieve and save as well as functions to utilize portions of a project such as import. The function of most of the options on this menu are clear, but some options do need some more explanation.

Save and Copy

The save function saves only changes to the current Sequence, so changing the path and saving the project does not work for making a new copy of the project, since some of the Sequences are not copied. To make a new copy of the project use the COPY function.

Set Path/Drive to Logicmaster Nickname File

When this path is defined, UPTime checks this file for any undefined names when it cannot find a name in its own files. For this option enter the drive, directory and filename.

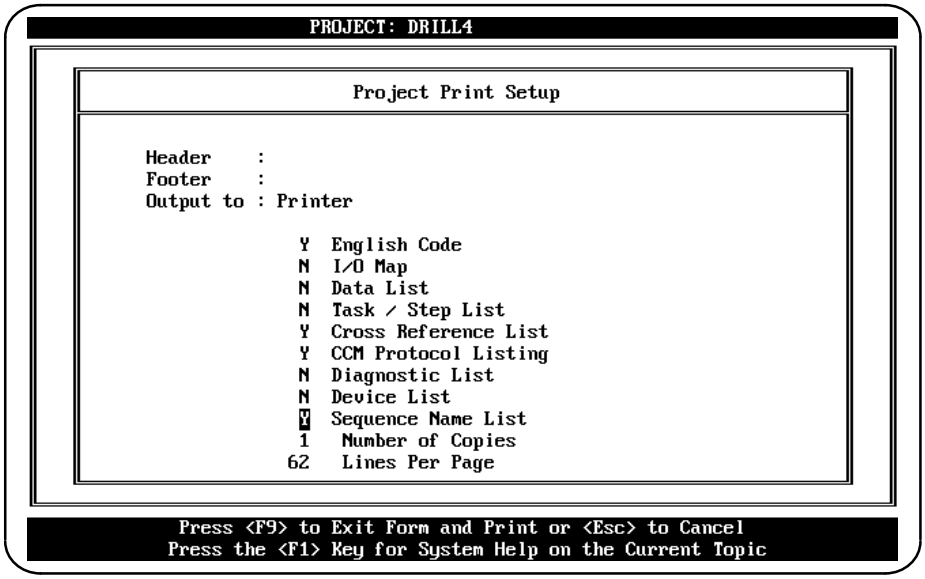
Download vs. Translate and Download

If the project has been changed, it must be translated before it is downloaded. If a changed project is just downloaded, a message is displayed notifying that the project in UPTime is more recent than the one in the Processor. Downloading a project that is already translated is faster than going through the translate and download operation.

Print Project Data

This option is used to create documentation for the current Diagnostic Strategy.

UPTime provides excellent self documenting features that are easily selected and printed from a Project Print Setup screen. To get into the Project Print Setup screen select <F3> from the Program mode of UPTime. From the menu select “Project” and then “Print” from the next menu. The following screen appears:



Header/Footer

The Header entry is printed on the top of each page of documentation. The footer entry is printed on the bottom of each page of documentation. Type the desired header and footer information in the highlighted box. The Header and Footer will accept up to 40 characters of information.

Directing the Output

The "Output To:" option specifies where the documentation is to be sent. If "Printer" is entered for this option, the documentation is sent to the printer connected to the parallel port. UPTime interprets any other name entered for this option to be a file name and the selected documentation options are sent to a disk to be stored in a file.

Documentation Options

This section describes each of the documentation options and shows a sample of the documentation produced when the option is selected. The Up and Down arrow keys are used to move the highlighted cursor to the options. To have a selection printed with the documentation package type a <Y> next to the option to be printed or a <N> to suppress a selection from being printed. Actually pressing any other key causes the form to toggle the display between Y and N for the highlighted option. For the "Number of Copies" and lines per page options type in the appropriate number. Press <ENTER> after selecting the number of packages to print. Press <F9> to start printing the documentation out the host computer's parallel port.

English Code

The Sequence text is printed when this option is selected. The name of the project and the date and time that the program was last modified is printed at the top of the page. A page number is printed at the bottom of the page. An example is listed below:

```
   English Code   Project: DRILL4      Last Modified: 6–15–93   11:59
   Sequence: INDEXING_CONVEYOR

Task: INDEXING_CONVEYOR_
    Step: WaitingToSTart
       Disable_Diagnostic PartAtLoad.
       If ConveyorMotor is ON, go to MovingToDrill Step.
       If 8 seconds have passed, go to the Delayed Step.

    Step: Delayed
       Turn on PartDelayed.
       If ConveyorMotor is ON, go to MovingToDrill Step.

    Step: MovingToDrill
       Enable_Diagnostic PartAtLoad.
       If ConveyorMotor is OFF, go to PartIsAtDrill Step.

    Step: PartIsAtDrill
       If ConveyorMotor is ON, go to MovingToUnload Step.

    Step: MovingToUnload  Max_Time 8
       If ConveyorMotor is OFF, go to WaitingToStart.

                Page 6
```

English Code

I/O Map

The I/O Map is a list of all of the names of CPU memory types defined in the State Logic program. The types included are %I, %Q, %AI, %AQ, %T, %M, %G, %S, %SA, %SB, %SC, and %R. The defined names are listed in numerical order according to type. The reference number address for each name is also listed as follows.

Data List

The data list option prints all of the names of data elements that have been defined in the program. The data list groups the names according to the type of element and each group is listed in alphabetical order. The types are listed in the following table:

Digital Point Name
Analog Channel Name
Floating Point Variable
Integer Variable
String Variable
Character Variable
Internal Flag

Types of Data Elements Included in Data List

Task / Step List

The Task / Step option prints each Task name of the program followed by all of the Step names in that Task. This listing produces a good overview of the process especially when the Task and Step names are descriptive of the operation performed.

Cross Reference List

The Cross Reference List produces a list of all the data elements of the program. Indented below each element is printed every Task/Step combination where that element is used in the program.

This print out is a valuable tool when debugging or troubleshooting a program. An example of a Cross Reference List is shown below:

CCM Protocol Listing

The "CCM Protocol Listing" option on the print setup form produces data for use in communicating to the State Engine using the CCM protocol. This protocol is typically used by host computers and computers running Graphical User Interface (GUI) programs to extract or change information in the State Engine. The CCM information printed lists the CCM type, the CCM type number, and the CCM number of the elements of the UPTime program.

Diagnostic List

This option lists all of the Diagnostics defined in the project. All of the selections of the Diagnostic are listed including name, type, limits, custom messages, and action.

Devices List

This opption lists all of the devices defined in the project. All of the parameters are listed in the printout.

Sequence Name List

Thsi option lists all of the Sequence names. The Sequence type is also specified.

Quit

This option exits the Create Mode so that the main menu is again displayed.

Debug Mode

The Debug Mode displays messages from the Diagnostic Processor and provides functions to monitor and control the execution of the Diagnostic Strategy running in the Diagnostic Processor.

Before starting the Debug Mode, UPTime compares the date and time of the last changes the program with the date and time of the program in the Diagnostic Processor. If the program in the Diagnostic Processor is not the same version as the one in UPTime, an error message is displayed. The version of the program used by UPTime must match the version of the one in the Diagnostic Processor.

The Debug Mode menu is accessed by pressing <F3>. The following sections describe each of the menu options.

Project

These functions are used to manage the Diagnostic Processor, manage program execution (start or stop or simulation mode), manage serial ports and logs, change Diagnostic Processor configuration, and download a new program. The options requiring explanation are described below.

Start Printer Output

This function allows the user to send all of the data, that is displayed in the terminal log, out to a printer attached to the parallel port (LPT1). When the printer log is active, the word **PRINT** appears in the top bar of the screen. If there is a printer error, the printer log is no longer active and an error message is displayed. To stop the printer log select that option from the menu.

Activate Log File Output

This function allows the data sent to the terminal log to also be sent to a disk file with a user given name of up to 8 characters. The log file name is given an extension of .LOG. The log file is limited in size to 100K bytes. When the file size reaches 100k the logging of data is terminated.

Diagnostic Processor Configuration

This option is used to set up the Diagnostic Processor. Several setup options are displayed on the following menu.

Enable/Disable CCM Protocol Port

The CCM port is the one not designated to be the programming port. When enabled, the CCM port responds to commands using the CCM protocol. The default state of the CCM port is disabled. See the Series 90–30 State Logic Control System section of this manual for more information of the serial ports.

Set CCM Protocol Station Address

Setting the CCM station number for the Diagnostic Processor, gives the Diagnostic Processor a number in the range 1 – 89. The Diagnostic Processor then only responds to commands using the specified station address.

Enable/Disable Automatic Program Execution on Power Up

The default Diagnostic Processor response to power up is to be in halt mode. If automatic execution is enabled, the program starts running automatically when power is applied to the system. The Diagnostic Processor configurations active when power was lost are still active when automatic execution is enabled. These options include serial port setups, error handling configuration, CCM setup, and programming port designation.

Error Handling Setup

There are two classes of run–time errors generated by the Diagnostic Processor; critical and non–critical. This option allows the Diagnostic Processor to be configured to either halt the program or continue running when an error in either one of these two classes occurs.

When this option is chosen, the first the response to critical and next to non–critical errors is chosen. This setup may be changed at any time.

Change Programming Port

This option allows serial port chosen to be the programming port to be changed. The programming port is the one that connects the Diagnostic Processor to UPTime. The default programming port is port 1.

Enter the number of the port that is the new programming port. If the programming port is changed, the current connection between the Diagnostic Processor and UPTime is no longer valid. UPTime must now be connected to the new port.

Change Diagnostic Message Port

This option allows the serial port chosen to be the Diagnostic Message Port to be changed. The Diagnostic Port is the one that displays any messages generated by Diagnostics that are tripped. The default Diagnostic Port is port 1.

Simulation Mode

When the Diagnostic Processor is in Simulation Mode, it does not communicate with the I/O The Diagnostic Processor does execute its program, but there is no transfer of inputs or outputs. To test a program in Simulation Mode, digital I/O and flags may be forced and analog variable values may be changed. The current Step of a Task may also be changed.

Automatic Execution

This options is used to control whether the Diagnostic Strategy starts execution when power is applied to the system. If this selection is not enabled the Strategy starts up HALTED.

Download Project to the Processor

Use this option to download a new program to the Diagnostic Processor. First the option to select a new directory is presented and then a list of projects from the current directory. Select the project to be downloaded to the Diagnostic Processor. A new project cannot be downloaded when the program is running. When the program is running, UPTime asks whether to stop the program or cancel the download.

Communication Port Reset

The communication port may become disabled by improper data transfer or noise on the serial cable so this function is available to reset the port and restore communications with the Diagnostic Processor.

Diagnostics

Use this option to view the Diagnostic definitions. A list of Diagnostic types is displayed and a form displays the definition plus the current values of the diagnostic variable.

Monitor

Monitor is an on–going display of the current values of all the selected elements. Up to 6 elements may be included in a monitor table at any time and up to 10 monitor tables may be defined. Monitor tables have user given names of up to 30 characters each. Spaces can be part of a Monitor Table name. To quickly move from one monitor table to another use the <Tab> and <Shift + Tab> keys.

View

The view option is used to display information about Sequences including the English text, the current status, and can also be used to start an Instructional Sequence. In addition, the current Step of every Task, Digital Device definitions, and System Status can be displayed.

While viewing the English text, the current value of a name can be displayed by putting the cursor on the word and pressing <F4>.

Trace

The Trace is a very useful debugging tool. A Trace display shows the most recent program Step transitions. This option allows the user to display the history of Step transitions on the screen for all Tasks or selected Tasks. The most recent transitions appear at the top of the screen. There are functions available for searching for a string of characters in the Trace.

Force

These functions allow a user to force Digital points and Internal Flags and analog channels. Forced items remain in the forced state independent of real world conditions

or program actions. The only way that a forced condition changes is for the force option to be used again to either change the forced value or clear the force. Once the force is cleared, outputs go to whatever conditions specified by the program and inputs go to whatever value is set by the CPU. The Force function does not affect the values stored in the CPU unless the variable is defined as an output.

Alarm

This option displays alarm information about the Diagnostics. The Alarm List shows all Diagnostics that are currently tripped or in the alarm state including the time of the event. The Histogram display shows a history of Diagnostics being tripped and then cleared plus when the Diagnostic Strategy started or halted execution and the times of these events. The previously uploaded histogram can be displayed or saved in a file. The histogram shows the most recent events at the top of the list.

Display

The display function displays the current value of the selected element together with the current time in the Terminal Log. When the type of data to display is selected, a list of names of that type are displayed. Select the name to display from this list.

Change

The change function allows the user to enter a new value for the selected item. After the new value is entered, it is sent to the Diagnostic Processor immediately. Also the change is reported to the Terminal Log together with the current time. One of the things that can be changed is the current Step of a Task.

PID Loops

This option appears on the debug main menu only if the program has defined a PID loop. Choose this option to tune a PID loop while the process is running. To tune a PID loop, change the parameters displayed.

PLC I/O

This option is used to view definitions of program names referring to CPU memory locations. Analog point, discrete point, and register variable definitions are all displayed using this option.

Fault

Use this option to display and clear both PLC and I/O faults. When the option to display a fault table is displayed that table is then displayed on the screen. To clear a fault table, just select that option from the menu.

The display lists the current time, the last time faults were cleared, total number of faults in the table, and faults since the last time faults were cleared.

The PLC Faults Table displays general system faults for different slots in the Series 90–30 chassis. The I/O Fault Table displays specific circuit faults. Both tables list the date and time of the fault.

Setup

Use this option to configure the UPTime software package. By changing the maximum number of Variables and Steps, the amount of memory required to execute UPTime is changed. To use UPTime at the maximum settings, use DOS 5 or better with DOS loaded HIGH, and delete all uneeded TSR programs and device drivers. It is recommended that the lowest maximums are used to increase the software performance and use the least amount of memory.

The other option of the setup form is a password for UPTime. If a password is entered, it must be used when starting UPTime. If the password is not entered, UPTime does not execute.

Keyboard Definitions

Table 6-1. Function Key Definitions

Key	Functions			
	<Key>	<Alt + Key>	<Ctrl + Key>	<Shift + Key>
<F1>	Help	Toggle Insert Overtype Mode	Project Error Help	
<F2>	Save File to Disk	Send to Controller	Retrieve from Disk	Check for Errors
<F3>	Menu			
<F4>	Define / View Current Word	Define All Undefined Words		
<F5>	Find / Replace Next	Find Text	Replace Text	Replace All
<F6>	Add Step	Add Task		
<F7>	Go to Another Task	Last Project Error		
<F8>	Mark a Block	Copy a Block	Move Block	Remove a Block
<F9>	Save / Exit Form	Change Frame / Loop in Form		
<F10>	Toggle Program / Debug Modes			

Table 6-2. Hot Key Definitions

Key	Function	Mode
<Ctrl + D>	List Digital Points	Program / Debug
<Ctrl + A>	List Analog Channels	Program / Debug
<Ctrl + N>	List Numeric Variables	Program / Debug
<Ctrl + S>	List String / Character Variables	Program / Debug
<Ctrl + Q>	Quit Current Mode	Program / Debug
<Ctrl + K>	List Keywords	Program
<Ctrl + W>	List Filler Words	Program
<Ctrl + P>	List PID Loops	Program
<Ctrl + U>	Undelete the last Deleted Block, ie. Paste.	Program
<Ctrl + R>	Communication Port Reset in Debugger	Debug
<Ctrl + V>	View English in Debugger	Debug
<Ctrl + E>	Enable Monitor Display in Debugger	Debug
<Ctrl + F>	Force Table	Debug
<Ctrl + T>	Trace Upload / Display	Debug

Table 6-3. Miscellaneous Key Definitions

Key	Functions	
	<Key>	**<Ctrl + key>**
<Insert>	Add / Paste	
<Delete>	Remove / Cut Character / Item	
<Home>	Left side of Screen / Field	
<End>	Right side of Screen / Field	
<Pg Up>	Scroll Up	Top of Project / List / Menu
<Pg Down>	Scroll Down	End of Project / List / Menu
<Up>	Up 1 Line / Field	
<Down>	Down 1 Line / Field	
<Left>	Left 1 Character / Field	
<Right>	Right 1 Character / Field	
<Tab>	Insert 4 spaces / Next Item in a List / Menu	
<Enter>	Carriage Return / Select Item	
<Esc>	Cancel Operation	

Chapter 7 Fill-in-the-Blank Diagnostics

This chapter describes how to create, control, and monitor Diagnostics. The different Diagnostic types are explained with reference to the different forms for creating each type. Also explained are the debug mode functions used to view Diagnostic definitions and create a record of alarms.

Creating Diagnostics

Diagnostics are created by entering information into an UPTime form. The first blank in the form is the name of the item being monitored. The next information specifies the conditions causing the Diagnostic to trigger. The last type of information indicates the action to take when the Diagnostic trips.

Name Blank in Diagnostic Forms

Names are defined before being used in a Diagnostic form. A name identifies an I/O point or variable memory location. The name can be one defined in UPTime, a nickname defined in Logicmaster 90–30 for the Ladder Logic program, or a direct memory reference such as R12 to represent the %R12 register.

UPTime Names

A name defined in UPTime can be up to 20 characters, either letters or numbers and the underscore character (_), but UPTime names must start with a letter. UPTime displays a list of names defined for the type of data used in this Diagnostic, when the <Ins> key is pressed and the cursor is in the name blank. This function is available if a blank in any UPTime form displays the double greater–than signs (>>).

A name can be selected from the list by highlighting the desired name and pressing the <Enter> key. To quickly locate a name on the list type in the first letter of the name, and UPTime highlights the first name starting with that letter. Typing in the second letter locates the first name starting with these two letters.

Logicmaster Names

To use nicknames defined in Logicmaster 90–30 follow these steps:

1. Create a nickname file using the Logicmaster print functions:

   From the Logicmaster Programming Package
   Select “Print” <F10> from the main menu
   Select “Logic” <F4> from the print function menu
   Select to have only the Variable Table printed
   Set Port to FILE by hitting the <Tab> key until that option appears
   Enter the file name

2. Tell UPTime where to find the file

   Select “Project” from the main menu
   Select “Set Path/Drive to Logicmaster Nickname File”
   Enter the drive, directory, and filename of the file created in step 1

3. Finally just use the Logicmaster Nicknames in Diagnostics or State Logic just as UPTime names are used.

When UPTime searches for the definition of a name, it first looks for names defined in UPTime. If the name is not found, UPTime searches the Logicmaster file of nicknames to identify the name.

Default nicknames used with the status bits, %s, %SA, %SB, %SC can also be used such as FST_SCN for the first scan bit, %S1 or LOW_BAT for the low battery indicator, %SA11. See the file DEFNICK is the \UPTIME\S90–30 directory for a full list of default status bit nicknames available.

Direct Memory Reference

Both Diagnostics and State Logic programming can use direct references to the CPU memory location. The memory types that can be accessed are %I, %Q, %AI, %AQ, %R, %M, %G, %T, %S, %SA, %SB, and %SC. For example Q23 refers the digital output memory location %Q23.

Wherever a name would normally be used, a direct memory reference is allowed. UPTime creates an internal name in its name table when a direct memory reference is encountered. If Q23 is specified, UPTime creates a name Q23 and assigns it to %Q23 in its tables. UPTime checks that there is not another name for that memory location before making the assignment.

Triggering Value Blank in Diagnostic Forms

The blanks in the Diagnostic forms for specifying the value that causes the Diagnostic to go to the alarm state are different for the different types.

Analog Diagnostic

Analog Diagnostic forms have both a high limit and a low limit blank. The Diagnostic is in alarm state when the analog value is outside these limits.

The form allows a floating point value to be specified for the high limit and low limit values. If an analog point has been scaled, the Diagnostic Processor represents the analog value in floating point format. If the analog channel has not been scaled, the values used for the high and low limits must be integer values between –32768 and +32767. See the section titled Analog Scaling and Update Rates in the State Logic Programming chapter of this manual.

If high or low limits are specified using floating point values in Diagnostic that refers to an unscaled analog channel, an error is displayed when the project is translated. The same error is displayed if an integer is out of range.

Digital Diagnostic

Digital Diagnostics can be configured to trip when the digital point goes ON, OFF, or every time the point changes state. Move the cursor to the blank and hit any key, then UPTime displays the three options. Select one of the options from the list.

Variable Diagnostic

Variable Diagnostics can monitor either Register variables which are stored in the CPU or variables stored in the Diagnostic Processor called Numeric Variables. When creating a Variable Diagnostic, UPTime first asks whether the data type is integer or floating point. Both CPU and Diagnostic Processor variables can be either integer or floating point values.

If the data type of the variable is integer, the high limit and low limit blanks accept only integer data from –32,768 to +32,767. UPTime indicates an error if an illegal value is entered.

If the data type is floating point, the high and low limit blanks allow for floating point data to be entered. There can be 6 digits to the left of the decimal point and two to the right.

Device Diagnostic

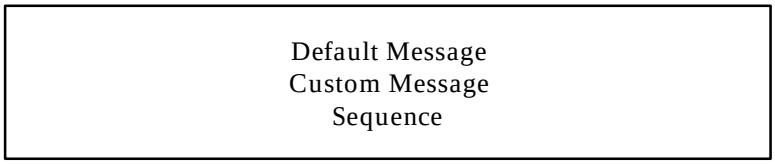
The triggering values for the Device Diagnostic is a time value. The two blanks in this form specify the maximum allotted time for the device to open and to close. See the description of the Device Structure in the Diagnostic Strategy Overview chapter of this manual.

This time is specified in seconds and can be any value between 1 and 65535 seconds, which is a little over 18 hours. For longer times use the Variable Diagnostic in conjunction with the Time_Counter Perform Function.

A timer starts when the open digital point is in the state which starts the open or close function. If the open complete or close complete signal is not received before the allotted time, the Diagnostic trips.

Action Blank

The action blank of the Diagnostic forms specifies what the Diagnostic Processor does when the Diagnostic trips. There are three possible actions:



The Digital Diagnostic has only one action blank, but the others all have more than one action blank. Variable and analog Diagnostics have two actions, one for the high limit and the other for the low limit. The Device form provides four blanks, two for exceeding the time for opening and closing and two that trigger when the Device either opens or closes.

Default Message

If Default Message is the chosen action, the Diagnostic Processor automatically constructs a complete message indicating why the Diagnostic tripped, the element being monitored, and for some types the current value and the limit value.

The Default Message text can be modified by selecting the "Default Message Text" option from the Define menu. The modified text can be any string of characters up to 80 characters in length.

Custom Message

When Custom Message is the chosen action, UPTime provides a window where any message up to 128 characters can be entered. The message can also include variable and I/O values and control characters within the message. To include analog and variable values, use the element's name in the message, preceded by the per cent sign (%). Instead of the name the Diagnostic Processor prints out the value of the named item.

Sequence

If Sequence is the selected action, UPTime prompts for the name of the Sequence. When the Diagnostic trips, the Diagnostic Processor starts that Sequence executing. The Sequence is created as a Continuous Sequence, but it becomes a Triggered Sequence when specified as an action of a Diagnostic. See the discussions of Sequences in the chapters describing the Diagnostic Strategy in this manual.

Triggered Sequences must have an End_Sequence instruction in one of the Steps. When this keyword is executed all of the Tasks in the Sequence go to the inactive Step. If a Sequence is already executing when a Diagnostic action tries to start it executing, no action is taken. There are two ways that a Sequence can be stopped, the End_Sequnce instruction executed from within the Sequence or the Stop_Sequence instruction executed in another Sequence.

Controlling Diagnostics with State Logic

Diagnostics are active or enabled by default. Often it is necessary to disable a Diagnostic so that frivolous alarms are not reported when they are not wanted. For example a Diagnostic might be set up to monitor for low hydraulic pressure, but such an alarm would be a nuisance when the machine were first starting up and the hydraulic pressure is expected to be low.

State Logic programming is used to disable and reenable Diagnostics. The keywords that control the Diagnostic status are DISABLE_DIAGNOSTIC and ENABLE_DIAGNOSTIC. These keywords are followed by the name of the Diagnostic being controlled.

Once a Diagnostic is enabled or disabled, it stays in that state until another Statement changes its state. Therefore the status of a Diagnostic does not change simply because a Task changes Steps.

If a Diagnostic Strategy is composed of Diagnostics only with no programming, all of the Diagnostics are always active. Therefore no programming is required to activate Diagnostics if they have not been previously disabled.

Online Diagnostic Features

The Debug Mode features using the Diagnostics display Diagnostic definitions, list Diagnostics that are in alarm status and presents a history of when Diagnostics went into alarm and when the alarm was cleared.

Terminal Log

One of the main uses of Diagnostics is to display information that it has tripped in the form of messages sent out a serial port. When UPTime is connected to the Diagnostic Port, the messages are displayed in the Terminal Log which is the main display screen of the Debug Mode. A message is sent to the Diagnostic port each time the Diagnostic tripped and each time the alarm conditions is cleared.

All of the messages displayed are saved even though the messages have scrolled off the screen. To display scrolled messages press the <Up> key. These messages can also be logged as they occur to a disk file and / or to a printer connected to the parallel port, LPT1. These messages can be displayed by any serial display device such as dumb terminals and marquee displays.

Menu Options

The DIAGNOSTICS menu option displays Diagnostic definitions and the current values of key Diagnostic elements. This function displays any of the four types of Diagnostics.

The ALARM menu option displays alarm information about Diagnostics. The Alarm List shows the Diagnostics that are currently in the alarm state. The list includes the date and time that the Diagnostic was tripped and the type of alarm.

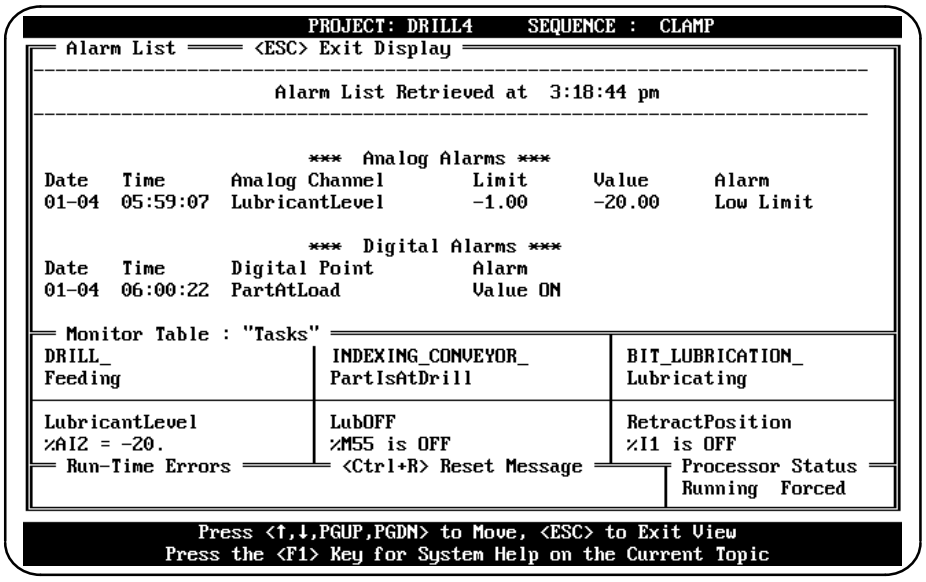


Figure 7-1. Alarm List Display

The Alarm Histogram provides a history of when Diagnostics tripped and when they cleared. Included in this list are the times that the program started and stopped executing. The most recent events are displayed at the top of the list.

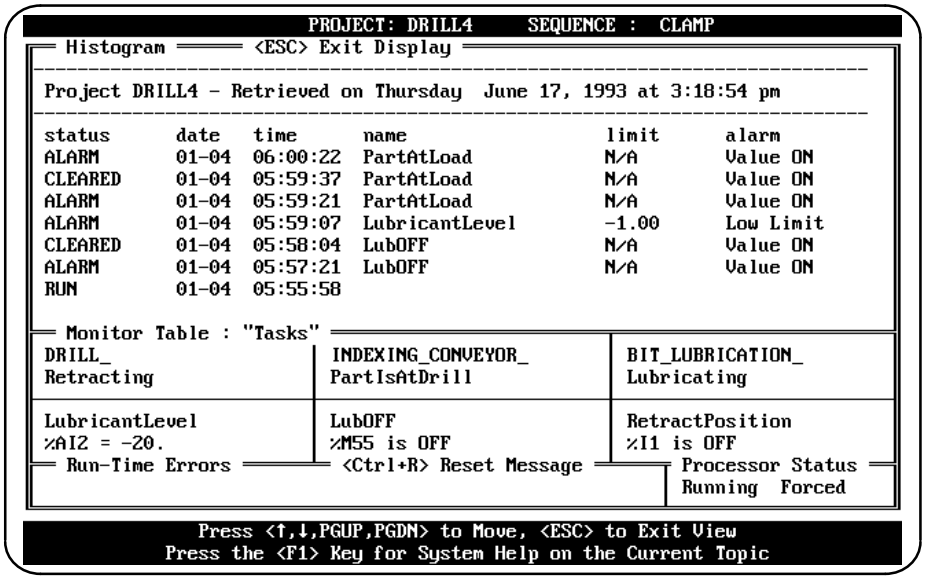


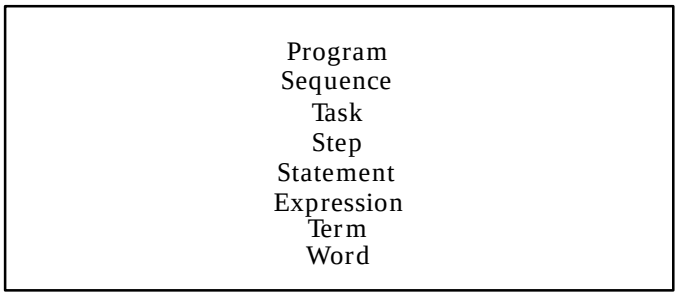
Figure 7-2. Histogram Display

Chapter 8 State Logic Programming

This chapter describes the details of creating the State Logic programming part of the Diagnostic Strategy. The detailed information describes how to use the State Logic structure and the English keywords, filler words, names, references to I/O, programming PID Loops, grammatical rules, and Perform functions.

Program Structure

There is a hierarchy in the structure of the State Logic program. Each program is divided into one or more Sequences, and each Sequence may have one or more Tasks, each Task is divided into one or more Steps, etc. The figure below lists each element of the hierarchy in descending order of significance.



UPTime Program Hierarchy

This hierarchy of Sequences, Tasks, Steps, and Statements are explained in the Overview and Creating a Diagnostic Strategy chapters of this manual. There may be many Tasks all executing simultaneously. Each Step is described by one or more Statements and each Statement consists of Expressions. Expressions are constructed from Terms which are composed of words.

The UPTime editor is invoked when a Sequence is edited. One Sequence is edited at a time. To edit or create a Sequence select “Sequences” from the main menu and then the type of Sequence to edit

When editing a Sequence each Task begins with the keyword, **Task**, followed by a colon then a Task Name.

Task: Assembly

Each Task includes all of the Steps from the start of the Task to the beginning of the next Task.

Similarly, each Step begins with the keyword, **Step**, followed by a colon then a Step Name.

Step: Attach_Arm

Each Step includes all of the Statements from the start of the Step to the beginning the next Step. There is no limit to the number of Statements in a Step.

Statements, like English sentences, are terminated by a period. The first Statement of a Step begins right after the Step name and includes all of the expressions appearing before the period.

There are two types of **expressions**: Conditional and Functional. Functional expressions describe some action that the Diagnostic Processor executes. Conditional expressions describe a condition which must be true for the functional expression in the statement to be executed. Expressions consist of one or more Terms which may be combined with logical(AND / OR)operators.

Terms express the fundamental actions and status tests of the program. Terms are built from words.

All words are classified into one of three categories: Keywords, Names, and Filler words. All filler words are ignored by UPTime when the program is translated, therefore the keywords and names are the only important words in a Term. Words are separated by spaces.

UPTime comes with a full set of keywords which are the instructions that get executed when the program is running. The programmer can change the keywords an add synonyms for each keyword. The set of filler words can be similarly changed.

Names are words that the programmer defines to represent I / O points, variables, Tasks, Steps, or Devices. Names can be up to 20 characters long and must start with a letter but may contain both letters, numbers and the underscore character (_).

Language Notational Conventions

In the following sections there are several examples of how to use the various State Logic expressions. In addition there is a description of the different ways to use the expressions plus an extensive representation of all of the possible combinations of uses.

There are so many different combinations possible that it is necessary to use some notational conventions to rigorously define all of the possible structures. This notational representation is for those who need to know exactly how the different terms can be used, for most programmers the verbal description that accompanies the notational representation is sufficient. Use whichever description works best for you, most programmers use both.

Underline	– Identifies Keywords
[ ]	– Encloses terms which are optional
{ }	– Encloses terms which may be repeated
< >	– Encloses a generic description of a term
|	– Indicates that either the term before or after may be used at this point.
( )	– Group Terms Together

Table 8-1. Language Structure Notational Conventions

The State Logic hierarchy terms described in the previous section are now specified using these notational conventions.

Program = { **<Sequence>** }

A program is one or more Sequences.

Sequence = { **<Task>** }

A Sequence is one or more Tasks.

Task = Task: <Task Name> [Startin_Last_Step] { <Step> }

A Task is the keyword Task followed by a colon, then a Task name and one or more Steps. The keyword Start_In_Last_Step is optional and would follow the Task Name.

Step = Step: <Step Name> [Max_Time Numeric Constant][{ <Statement> }]

A Step is the keyword Step followed by a colon, then a Step name and optionally one or more Statements. A Step does not require any Statements. Optionally the Max_Time keyword followed by a number may follow the Step name.

Statement = ([<Conditional Expression>] <Functional Expression>) |
(<Functional Expression> [<Conditional Expression>])

Every Statement must have a Functional Expression and may or may not have a Conditional Expression. The Conditional Expression may come before or after the Functional Expression.

Functional Expressions

Functional Expressions are composed of one or more Functional Terms. There is no limit to the number of Functional Terms used in a functional expression.

Functional Expression = { **Functional Term** }

Functional Terms describe any action or changes in the diagnostic system; enabling or disabling Diagnostics, starting and stopping Sequence execution, turning on outputs, sending messages out a serial port, making a new Step the active Step of a Task, setting analog output values, and changing variable values, setting serial port parameters, starting and stopping PID loop execution, and doing specialized Perform functions are the operations controlled by Functional Terms.

```
Functional Term =  < Control Sequences Term > |
                   < Control Diagnostics Term > |
                   < Turn On Discrete Term > |
                   < Assign Values Term > |
                   < Change Active Steps Term > |
                   < Send Serial Information Term > |
                   < PID Control Term > |
                   < Change Serial Port Configuration Term > |
                   < Execute Perform Functions Term > |
                   WhenDone_Next_Step
```

Controlling Sequences

There are three keywords that are used to control Sequences. Start_Sequence and Stop_Sequence are used to control one Sequence from another. The End_Sequence keyword is used to control the Sequence that the keyword is in. Every Triggered and Instructional Sequence must have an End_Sequence keyword in one of its Steps.

```
Control Sequences Term = (Star_tSequence <Sequence Name>) |
                         ( Stop_Sequence < Sequence Name > ) |
                         End_Sequence
```

- Start_Sequence AssemblyMonitor.
- Stop_Sequence CrusherAlarms
- End_Sequence.

Controlling Diagnostics

The Fill–in–the–Blank Diagnostics are active when the project starts running, but they may be made inactive and reactivated by programming Statements. Once a Diagnostic has been enabled or disabled, it remains in that State until the execution of another control term changes its state. A disabled Diagnostic never goes to an alarm state or causes any actions to occur.

```
Control Diagnostics Term = ( EnableDiagnostic <Diagnostic Name> ) |
                           (DisableDiagnostic <Diagnostic_Name >)
```

- Enable_Diagnostic FeedTorque.
- Disable_DiagnosticHydraulicPressure.

Turning ON Discrete (Actuate Term)

The Actuate Term is used to turn on Digital I/O Points and Internal Flags. This Term starts with the keyword Actuate followed by one or more discrete names.

```
Turn On Discrete Term = Actuate        { <Digital I/O Name> |
                                         < Internal Flag Name > }
```

- Actuate the Ready_Light.
- Start Pump_1 and Pump_2.
- Energize Clamp_1, Clamp_2, Clamp_3 and Clamp_Flag.

Assigning Values (Make, Math–Assignment, Set_Bit/Clear_Bit)

To assign values use the Make Term, Math–Assignment Terms, Set_Bit / Clear_Bit Terms. These Terms assign values to variables and analog I / O points.

Assign Values Term = **< Make Term > |**
< Math–Assignment Term > |
< Set_Bit/Clear_Bit Term >

Make Term

The Make Term is used to assign a value to a variable or analog I / O point. The Term starts with the keyword Make and is followed by a variable or analog name, the keyword equal, then a number or a calculated value.

Make Term = Make **< Numeric Assignment Term > |**
< Character Assignment Term > |
< String Assignment Term>

Numeric Assignment Term = (<Numeric Variable Name> | <Analog I/O Name> | <PID Value>)
equal < Numeric Value >

Character Assignment Term = <Character Variable Name> equal <Character Value>

String Assignment Term = <String Variable Name> equal <String Value>

- Make Flow_Setpoint equal 25.
- Make Valve_Control = 67.89.
- Make Total_Defects equal Temperature_Failures + Stress_Failures.
- Make Output_String equal “Enter setpoint now”.
- Make Test_Character = ’$’.
- Make Tank_Level_PID Bias equal 34.456.

See the section “Calculated Values” for a description of how to do mathematical calculations.

Math–Assignment Term

This Term does a simple arithmetic operation on a variable value then assigns the new value to the variable. The four terms are add, subtract, multiply, and divide.

Math–Assignment Term =	**< Add Term > |** **< Subtract Term > |** **< Multiply Term > |** **< Divide Term >**

The Add Term is the keyword Add followed by a number or variable name then a variable name.

Add Term = Add	**(< Numeric Constant > | < Variable Name>)** **< Variable Name >**

- Add 1 to Parts_Count
- Add Second_Shift_Parts_Count to Total_Parts_Count

The Subtract Term is the keyword Subtract followed by a number or numeric variable name then a variable name.

Subtract Term = Subtract	**(< Numeric Constant > | < Variable Name >)** **< Variable Name >**

- Subtract 2.78 from Starting_Value
- Subtract Tare_Weight from Test_Weight.

The Multiply Term is the keyword Multiply followed by a variable name then a number or variable name.

Multiply Term = Multiply	**< Variable Name >** **(< Numeric Constant > | < Variable Name >)**

- Multiply Parts_Lost by 2
- Multiply Machine_Strokes by Strokes_Per_Cycle

The Divide Term is the keyword Divide followed by a variable name then a number or variable name.

Divide Term = Divide	**< Variable Name >** **(< Numeric Constant > | < Variable Name >)**

- Divide Right_Side_Length by 4.5
- Divide Box_Volume by Volume_Adjustment

Set_Bit/Clear_Bit Term

The Set_Bit / Clear_Bit Term is used to change individual bit values of an integer variable. The Set_Bit or Clear_Bit keyword is used followed by the variable name and then the zero based bit number to be changed.

Set_Bit/Clear_Bit Term =	**(SetBit | ClearBit)** **<Integer Variable Name> <Integer Number>**

- Set_Bit Transfer_Status 2.
- Clear_Bit Tac_Register 4.

Changing Active Steps Term

There are three ways of changing the active Step of a Task. A Task can change its own Step, change the active Step of another Task min the same Sequence, or cause a Task in the same Sequence to become inactive or resume its previous active Step.

Change Step Term =	**(Go <Step Name>) |** **(Make <Task Name> equal <Step Name>) |** **((Suspend_Task | Resume_Task) <Task Name>**

The GO term is the means by which a Task transitions to another Step. Only the Go and the Step name are mandatory, all other words are optional. The Go may appear in any Statement, but there may only be one Go per Statement. As soon as a GO term is executed, no other terms in the Statement and no other Statements in the Step are executed.

- Go to the Forward_Motion Step.

Tasks control other Tasks by merely setting the Task to a new Step value.

- Put the Assembly_Control Task into the Emergency_Stop Step.

In this example, put is a synonym for make and into is a synonym for equal. This operation may only be performed on a Task in the same Sequence.

The Suspend_Task and Resume_Task Keywords are also used to change the current Step of a Task. The Suspend_Task keyword puts the named Task into the Inactive Step. The operating system gives every Task a Step named Inactive. When a Task is in the Inactive Step, this Task performs no activity and the only way to exit this Step is for another Task to change its current Step. The inactive Step can also be used with the GO keyword.

The Resume_Task keyword causes the named Task to go to the Step that was active before being suspended. Resume_Task and Suspend_Task only operate on Tasks in the same Sequence.

- If Water_Level is above 45.5 feet then Suspend_Task Fill_Tank.
- If Water_Level is below 43.8 feet Resume_Task Fill_Tank.

Sending Character Data (Write Term)

The Write Term is the keyword Write followed by data to send inside double quotes. Optionally a communications port name or R–Register numbers may be specified following the data to be sent. If no port name or R–Register is specified the data is sent to the programming port.

Send Serial Data Term = Write " <Serial Data> "	**[<Port Name> | R_Register <Number>]**

- Write "Push Start Button" to Operator_Control.
- Write "ERROR Number 16" to R_Register 10.

The data is directed to either serial port which is named or to a series of R_Registers. If the data is directed to R_Registers the string of characters are stored in the Series 90 register reference table beginning at the %R register number which follows the R_Register keyword. In the example above the characters in the string, "ERROR Number 16", are stored in successive bytes starting at %R10.

The serial data can be a mixture of typed text, variable values, ASCII control characters, and formatting characters.

The **typed text** are any characters entered directly from the keyboard. The text may include carriage returns so that several lines can be entered in one Write Term. Multiple line messages can be formatted in the program exactly as they appear on an terminal screen.

```
Write "
Opening Operator MENU

1. Change Today's Date
2. Change the Current Time
3. Engage Startup Procedure
4. Restart the Process"
to Operator_Panel.
```

The menu from this Write Term appears on the operator screen just as it does in the program. The display on the operator screen is as follows:

```
Opening Operator MENU

1. Change Today's Date
2. Change the Current Time
3. Engage Startup Procedure
4. Restart the Process.
```

The limit of the number of characters between the quotes is 512, which is about 7 full (80 character) lines of text.

Variable values are sent out the port by preceding a variable name with a "%".

- Write "Current parts count is %Part_Count." to Operator_Terminal.

If Part_Count is 10 at the time the Write Term is sent, this Statement displays the following line on the screen connected to the port named Operator_Terminal.

Current parts count is 10.

Formatting Characters that are used with the Write Term follow:

%NOCRLF – Write Terms always send a carriage return line feed pair following each message. Use this formatting feature suppresses these terminating characters.
%CRLF – sends a carriage return line feed character pair
%CRLF(X) – X number of carriage return, line feeds
%CLS – Clear the Screen, sends 25 carriage return, line feeds
%SPACE(X) – X number of spaces
%CHR(X) – The ASCII character for the value in the "()" is sent
%#X – The ASCII character for the hexadecimal value X is sent

- The " | " character is used to place two words together without any spaces in between them. For example, "%Pressurepsi" would look like one long variable name to UPTime and would yield an error message. But "%Pressure | psi" would yield the desired result of the value directly followed by the character string, "psi".
- To send a double quote sign use "%#22" or "%CHR(34)". To send per cent sign use "%%". All other keyboard characters are sent by simply typing them between the quotes.

PID Loops Control Terms (Start_PID, Stop_PID)

PID Loop control Statements start with the keywords Start_PID or Stop_PID, followed by the PID Loop Name. If stopping a PID loop, a value which sets the value of the control variable can follow the PID loop name.

PID Loop Control Term = Start PID Term | Stop PID Term

Start PID Term = Start_PID <PID Loop Name>

Stop PID Term = Stop_Pid <PID Loop Name> [with <Numeric Constant>]

- Start_PID Oven_1.
- Stop_PID KILN5 with 456.29.

Change Serial Port Configuration Term

The Term to change the configuration of a serial port is the Set_Commport keyword followed by a port name and then a list of parameters and their values.

>Port Configuration Term =	**Set_Commport <Port Name>**
	<Parameter Value List>

This Functional Term is automatically entered into the program by using the "Communication Ports" option on the LIST menu. Select the port and press the right arrow key "–>" to change the configuration options. When configuration is complete, select the "Insert Reconfiguration Data for the Port" option from the next menu. The entire Term is entered into the program at the current cursor location.

Perform Function Term

This Term is the Perform keyword followed by the function name, the keyword with and then a list of parameters and values.

Perform Function Term = Perform	**<Function Name>**
	with <Parameter Value List>

This Term is entered automatically by UPTime at the end of the Step where the cursor is located. First select the "Add" option from the menu, then the "Add a Perform Function" option. Fill in the blanks that are displayed after selecting the function desired.

When_Done_Next_Step

This keyword must occur in a Statement by itself, ie. with no other terms in that Statement. This keyword is most often used in Instructional Sequences to allow the user to interact with the program displaying the Steps of an instruction.

When this term is executed, the following prompt is displayed:

Enter R to repeat, S to Stop, or C to continue –>

The user has control of whether to have the current step of instructions repeated, stop the sequence of instructions, or continue with the next Step of the program. If continue to the next Step is selected, the Step below the current one in the program is executed.

ConditionalExpressions

Conditional Expressions are used for receiving information in a serial port and for testing a value or status. The Functional Expression in a Statement is executed only if the Conditional Expression in the Statement is true. The usual action that is dependent on a conditional is a transition to another Step.

Conditional Expression =	**< Test Conditional > |**
	< Character Input Conditional >

Test Conditional

The test conditional starts with the keyword, If, and is followed by one of the four types of Test Conditional Terms. The keyword NOT can precede the conditional term. Several Conditional Terms can be joined by the AND and OR keywords. The AND Terms have lower precedence and are therefore executed first. The order of execution can be changed by use of parenthesis and parenthesis can be nested.

Test Conditional = If **[NOT] <Conditional Term>**
[{ (OR | AND) [NOT] <Conditional Term> }]

The type of Conditional Test Terms are Digital, Timer, Relational, and Current Step.

Conditional Term = **<Digital Conditional> |**
<Timer Conditional> |
<Relational Conditional> |
<Current Step Conditional>

Digital Conditional

The Digital Conditional Term tests the status of digital I/O circuits, digital devices, and internal flags. This term is a discrete name followed by the keyword ON or OFF.

Digital Conditional = **(<Digital I/O Name> | <Flag Name> | <Device Name>)**
[{ (AND | OR)
(<Digital I/O Name> | <Flag Name> | <Device Name>) }]
(ON | OFF)

- The following notational conventions are used throughout this section to rigorously define the required structure:
- If **Forward_Limit_Switch is on . . .**
- If **Part_Ready_Flag is off . . .**

Several digitals can be specified in the same expression joined by AND or OR keywords as follows:

- If **Top_Limit_Switch or Bottom_Limit_Switch and Counter_Weight_Switch are OFF . . .**

The ANDs have a lower precedence and are therefore executed first.

Timer Conditional Term

The Timer Conditional is a number or variable followed by the keyword SECONDS. The timer has a resolution of 1/100 of a second and the number used to indicate the number of seconds can be a floating point number.

Timer Conditional = (<Numeric Constant> | <Integer Variable Name>) seconds

- If **3.76 seconds have passed, then . . .**

An integer variable can also be used to specify the number of seconds. The value of the variable indicates the number of hundredths of a second, so

that a value of 100 would indicate a time of 1 second.

- If W**ait_Time seconds, then . . .**

Timers always refer to the amount of time that the Step has been active. A common mistake is to assume that the timer starts when it is scanned.

- If **Track_Monitor is ON and 5.3 seconds have passed . . .**

The timer above refers to the time that the Step has been active and is not influenced by the condition of the Track_Monitor.

The timer number must be in the range of 0.01 to 600.00 seconds or 10 minutes. When using an integer variable, the variable value must be between 0 and 32767.

There are several ways to make a timer that uses a period of time greater than 10 minutes. The common methods use Step transitions to reset a Step timer.

Step: Heater_On_One_Hour
Actuate Vat_Heater.
If Ten_Minute_Counter is >= 6, go to Start_Process Step.
Wait 600 seconds then go to Timer_Counter Step.

Step: Timer_Counter
Add 1 to Ten_Minute_Counter and go to Heater_On_One_Hour Step.

Relational Conditional Term

Relational Terms test a variable or analog values. The Term is a value, followed by a relational operator, then another value. The values tested can be numbers, calculations, variable names, and analog names.

Relational Conditional = <Numeric Relational Term> |
<Character Relational Term> |
<String Relational Term>

Numeric Relational Term = <Numeric Value> <Relational Operator> <Numeric Value>

Character Relational Term = <Character Value> (**equal** | **not_equal**) <Character Value>

String Relational Term =	<String Value> (**equal** | **not_equal**) <String Value >

- If **Parts_Count = 500 . . .**
- If **Flow_Meter_Input is above Flow_High_Limit . . .**
- If **Canister_Pressure – Atmosphere <= Pressure_Limit – Safety_Margin . . .**
- If **String_Entry equal "Formula 1" . . .**
- If **Test_Char not_equal '@' . . .**

See the section, Mathematical Calculations, for a discussion on how to use calculations.

Current Step Conditional Term

The Current Step Conditional is a Task Name followed by the keywords EQUAL or Not_Equal and then a Step Name. This conditional is used to test the current Step of another Task. This term can test the current Step of any other Task whether or not the other Task is in another Sequence.

Current Step Conditional =	**<Task Name>** **(equal | not_equal)** **<Step Name>**

- If Pump_Monitor Task is in the Backwash Step . . .

Complex Conditionals

Conditionals can be preceded by the NOT keyword and several can be joined by the AND or OR keywords and parenthesis can be used to change order of evaluation.

- If Hydraulic_Pump_Control Task is in the Over_Pressure Step or Hydraulic_Pressure is above 23.56
- If 1 seconds and not Temperature_Setpoint greater than 4.67 / Settling_Value
- If Spin_Drive is ON and (Pour_Ladle is not_in Pouring Step or not Mold_Number above 67

Character Input Conditional

The syntax for this conditional is the keyword READ followed by a variable name. This conditional is true when a character input message is completed. The character input is stored in the variable listed.

Optionally this conditional can specify the port from which the input is received. If this option is used the keyword FROM follows the variable name and then a communications port name is listed.

Another option is to receive the input from R Registers of CPU. The syntax for this option is again to use the keyword from followed by the keyword R Register then the starting register number, the keyword for and then the number of registers to read. When using this option, the variable receiving the data must be a string variable. Each byte of the R Registers read is stored in the string variable.

Character Input = Read	**<Variable Name>** **[from <Communications Port Name> |** **(R_Register <Register Number> for** **<Integer Number>)]**

- Read Menu_Choice from Operator_Station . . .
- Read Error_String from R_Register 46 for 20 R_Registers . . .

The first example reads data into the variable named, Menu_Choice, from the communications port named Operator_Station. The second example reads 40 characters into the string variable, Error_String, from the 20 CPU R Registers starting at %R46.

A GO Functional Term must always follow the character input conditional and there cannot be any other Terms in the Statement. If two character input conditionals for the same port are written in the same Step it is unknown which conditional receives a message from the port.

The types of variables used with the Read are:

Integer Variables
Floating Point Variables
String Variables
Character Variables

If the type of data received does not match the variable type, the input is ignored and the conditional is not satisfied. An example of invalid data is entering string of characters to a numeric variable.

The input is completed and the conditional is true, when an end of message character is received at the port. The default end of message character is the carriage return, so that normally the input is completed when the <Enter> key is pressed. The end of message character may be changed by the Set_Commport Functional Expression.

Input to a character variable is complete as soon as one character is received, so that a character is stored and the GO executed as soon as any character is received.

IMPORTANT: Character variables cannot be used to receive input through the programming port when connected to UPTime. Character variables work well if some other serial device is connected to the programming port or the other port also.

The Words of the State Logic Program

All words in State Logic programming fall into one of three categories, keywords, names, and filler words. Names are defined by the user and there a set of predefined keywords and filler words. The keywords and filler words can be changed by the user.

Names

Names are used to refer to I/O points, variables, memory locations, serial ports, digital devices, flags, Sequences, Tasks, and Steps. A name may be define in UPTime before, during, or after program development. Nicknames used in the ladder logic program and direct memory references may also be used.

UPTime Names

A name defined in UPTime can be up to 20 characters, either letters or numbers and the underscore character (_) but must start with a letter. A name can be defined through many menu options. The VARIABLES option from the Create Mode main menu and the LIST and DEFINE menus from the Sequence editor. UPTime can search for any undefined words so they can be defined one at a time when highlighted.

A name can be selected from the list by highlighting the desired name and pressing the <Enter> key. To quickly locate a name on the list type in the first letter of the name, and UPTime highlights the first name starting with that letter. Typing in the second letter locates the first name starting with these two letters.

Logicmaster Names

To use nicknames defined in Logicmaster 90–30 follow these steps:

1. Create a nickname file using the Logicmaster print functions:
   From the Logicmaster Programming Package
   Select “Print” <F10> from the main menu
   Select “Logic” <F4> from the print function menu
   Select to have only the Variable Table printed
   Set Port to FILE by hitting the <Tab> key until that option appears
   Enter the file name
2. Tell UPTime where to find the file
   Select “Project” from the main menu
   Select “Set Path / Drive to Logicmaster Nickname File”
   Enter the drive, directory, and file name of the file created in step 1
3. Finally just use the Logicmaster Nicknames in Diagnostics or State Logic just as UPTime names are used.

When UPTime searches for the definition of a name, it first looks for names defined in UPTime. If the name is not found, UPTime searches the Logicmaster file of nicknames to identify the name.

Default nicknames used with the status bits, %s, %SA, %SB, %SC can also be used, such as FST_SCN for the first scan bit, %S1 or LOW_BAT for the low battery indicator, %SA11. See the file DEFNICK in the \UPTIME\S90–30 directory, for a full list of default status bit nicknames available.

Direct Memory Reference

Both Diagnostics and State Logic programming can use direct references to the CPU memory location. The memory types that can be accessed are %I, %Q, %AI, %AQ, %R, %M, %G, %T, %S, %SA, %SB, and %SC. For example Q23 refers the digital output memory location %Q23.

Wherever a name would normally be used, a direct memory reference is allowed. UPTime creates an internal name in its name table when a direct memory reference is encountered. If Q23 is specified, UPTime creates a name Q23 and assigns it to %Q23 in its tables. UPTime checks that there is not another name for that memory location before making the assignment.

Keywords

This section displays the set of Keywords supplied with UPTime. Up to 10 synonyms may be added for each Keyword. The Default Keyword may also be changed. The Keywords are broken into four categories, Conditional terms, Functional terms, Operators and miscellaneous words that modify the meaning of a Statement. In the following tables the default keyword is displayed in **bold** print with some suggested synonyms in normal print.

Table 8-2. Conditional Terms

Keyword, Synonyms	Meaning / • Examples
If, When	Test conditions and values, actions executed when test returns TRUE condition. • **When** the Forward_Limit_Switch is ON, go . . . • **If** Count is > 1, go . . .
For, Until	Test conditions and values, actions are executed when the test returns a False condition. • Actuate Stop_Light **until** Parts_Count = 23. • Turn on Alarm_Light **for** 2 seconds then go to Reset.
Read	Get input from comm port or CPU Registers, must be used with GO term. • **Read** Name from Port_1 then go Display_Name. • **Read** Status_String from R_Register 69 for 23 R_Registers, then go to Display_Status Step.
Wait	Time Delay, must be used with GO term. • **Wait** 2.34 seconds then go to Restart Step.

Table 8-3. Functional Terms

Keyword, Synonyms	Meaning / • Examples
Energize, Start, Actuate, Turn, Run, Open, Turn_On	Turn on a Digital Point(s) • **Start** Conveyor_Motor. • **Energize** Forward_Solenoid. • **Actuate** Backwash_Pump, and Backwash_Pump_Light.
Add	Add a value to a variable • **Add** 2 to Count.
Divide	Divide a variable by a value • **Divide** Count by 2.
Go	Make another Step the Active Step • If Switch1 is On, **go** to the Motion Step.
Halt	Stop the process immediately • If Alarm is On, **Halt**.
Make, Put, Place, Set	Assignment operator initiator • **Make** Total = 56.
Multiply	Multiply a variable by a value • **Multiply** Count by 2.
Perform, Execute	Invoke an UPTime "Perform" function • **Perform** Display_Date_Time with...
Set_Commport	Change comm port settings while running • **Set_Commport** Port_1 with Baud_Rate=9600, Data_Bits=8, Parity=N, Stop_Bits=2, Auto_Echo=Y, Xon_Xoff=Y, Receiver_On=N, End_of_Message_Char=h0d.
Start_PID	Invoke a PID Loop • **Start_PID** Main_Loop.
Step	Identifies the Name of State Logic Steps •**Step**: PowerUp • go to the Motion **Step**.
Stop_PID	Halt a PID Loop and set the output value. • **Stop_PID** Main_Loop with 234.5.
Subtract	Subtract a value from a variable • **Subtract** 1 from Count.
Task	Identifies the Name of State Logic Tasks • **Task**: Main • If the Main **Task** is in the PowerUp Step, go...
Set_Bit	Set a condition TRUE or a bit of a 16 bit integer value to 1 • **Set_Bit** Flowmeter_Counter HSC_OUTPUT_ENABLE. • **Set_Bit** Integer_Variable_1 0.
Clear_Bit	Set a condition FALSE or a bit of a 16 bit integer value to 0 • **Clear_Bit** Resolver_Counter HSC_RESET_PRELOAD. • **Clear_Bit** Integer_Variable_2 15.
Suspend_Task	Stop the execution of a Task – no Step is active. • If Level > alarm, **Suspend_Task** Automatic.

Table 8-3. Functional Terms (continued)

Keyword, Synonyms	Meaning / • Examples
Resume_Task	Restart the execution of a task – the Step that was active when the Task was suspended is made the active Step. • If Level < alarm, **Resume_Task** Automatic.
Write	Send data out the comm port • **Write** "Error Message" to Operator_Console.
Start_Sequence	Start all Tasks in the named Sequence running in their first Step. • Start_Sequence FaultAnalysis.
Stop_Seqeunce	Put all Tasks into the Inactive Step in the named Sequence. • Stop_Sequence ReportFaults.
End_Sequence	Causes all Tasks in the current Sequence to go to the Inactive Step. • End_Sequence.
When_Done_Next_Step	Writes a prompt for the user to asks that the last instruction be repeated, continue to next Step in the program, or stop the Instructional Sequence. • When_Done_Next_Step.

Table 8-4. Mathematical Operators

Keyword, Synonyms	Meaning / • Examples	Precedence
()	Parentheses – Used to group terms to change order of operation. Up to 18 levels of parentheses are permitted. Parentheses may be used in mathematical expressions and with relational conditional terms.	1
ARCTAN(exp)	Arctangent: where –65535 <= exp <= 65535 • Make Overlay = **ARCTAN**(Hyp * 2).	2
COS(exp)	Cosine: where –65535 <= exp <= 65535 • Make Near = **COS**(Test_Value).	2
EXP(exp)	e to a power: • Make Inverse = **EXP**(Transfer)	2
LN(exp)	Natural logarithm (base e): • Make Test_Value = **LN**(Input – 3.4)	2
RANDOM	Random number generator: Generates a random number (0 – 1) • Make Simulated_Inp = Set * **Random**	2
SQRT(exp)	Square Root: • Make Out_Pot = **SQRT**(Flow_Meter).	2
SIN(exp)	Sine: where –65535 <= exp <= 65535 • Make Vector1 = **SIN**(Strain_Gauge).	2
TAN(exp)	Tangent: where –65535 <= exp <= 65535 • Make Slope = **TAN**(TRIM * In_Flow).	2
^	Exponential Operator • Make Count = Amount ^ 2.	2
%, Modulus	Modulus Operator – integer operands only • If Count **%** 5 = 0, go...	3
***, Times	Multiplication Operator • Make Count = Amount * 2.	3
/, Divided_By	Division Operator • Make Count = Amount / 2.	3
+ , Plus	Addition Operator • Make Count = Amount + 2.	4
– , Minus	Subtraction Operator • Make Count = Amount – 2.	4
Bitwise_And	AND bits operator • Value = Code Bitwise_And Mask.	4
Bitwise_Or	OR bits operator • Setup = Code Bitwise_Or Mask.	4
= , Is, Equal, Equals, Into	Assignment Operator • Make Count = 5,	5

Table 8-5. Relational Operators

Keyword, Synonyms	Meaning / • Examples	Precedence
= , Is, Equal, Equals, Into	Comparison Operator • If Count = 5, go...	5
< , Less, Under	Less than Operator • If Count < 5, go...	5
<= , =<	Less then or equal to Operator • If Count <= 5, go...	5
> , Greater, Above, More	Greater than Operator • If Count > 5, go...	5
>= , =>	Greater than or equal to Operator • If Count >= 5, go...	5
<> , Not_equal	Not Equal Operator • If Count <> 5, go...	5
AND	AND Operator for Conditional and Functional Terms • If Count > 5 **AND** Top_Switch is ON • Actuate Pump_5 **AND** Pump_6.	7
OR	OR Operator for Relational	8
	Conditional Terms Only • If Vat < 98 degrees **OR** Fuel < 12	
NOT	NOT Operator for Relational Conditional Terms Only • If **NOT** Inlet_Pressure > 100 psi	6

Table 8-6. Miscellaneous Keywords

Keyword, Synonyms	Meaning / • Examples
AM	Time Suffix • If Time is past 3:00 **AM**, go...
Friday	Day of week number 5 • If day_of_week = **Friday**, go...
From	Used in "Read" Terms to identify a port name. • Read Name **from** Port_1 then go to the Stretch Step.
Max_Time	Used to set the maximum time diagnostic for a Step • Step: PowerUp **Max_Time** 2.5
Monday	Day of week number 1 • If day_of_week = **Monday**, go...
Not	Logical "NOT" in a conditional expression • If **not** (Count > 1 or Count < 10), go...
Off, False, Not_True,Not_Tripped	Test for Digital I/O for not set state • If Switch1 is **Off**, go...
On, True, Tripped	Test for Digital I/O for set state • If Switch1 is **On**, go...
Or	Logical "OR" in a conditional expression • If Count > 1 **or** Count < 10, go...
Inactive	Name of the Step in Which No Actions Occur • Put the Manual Task into the **Inactive** Step.
PM	Time suffix • If Time is past 3:00 **PM**, go...
Saturday	Day of week number 6 • If day_of_week = **Saturday**, go...
Seconds	Used for comparison in a "Wait" Term • Wait 3.2 **seconds**, go... • If 3.2 **seconds** have passed, go...
Sunday	Day of week number 7 • If day_of_week = **Sunday**, go...
Thursday	Day of week number 4 • If day_of_week = **Thursday**, go...
Tuesday	Day of week number 2 • If day_of_week = **Tuesday**, go...
Wednesday	Day of week number 3 • If day_of_week = **Friday**, go...
With	Prefix for data that is needed by a function • Stop_PID Kiln_Temperature with 45.679 • Perform Get_User_Input **with** Clear_Screen = Y...
R_Register	Used to indicate %R registers in the Read Term • Read Status_String from **R_Register** 69 for 23 R_Registers

Filler Words

Filler words have no functionality, ie. they do not change the meaning of any of the statements in which they appear. Filler words are only used to increase the clarity of the English text. For example, “go to the Motion Step” looks better and sounds better than “go Motion”. To some programmers, however, typing fewer words is better.

Perform Functions

The Perform functions implement operations which are more complicated than the common State Language Terms. UPTime presents a form which has a template of information that is entered by the programmer. When this form is completed UPTime enters the English text for the proper execution of the function.

The Step where a Perform function appears should usually be structured so that the Perform is executed for only one scan. A common error in using Performs, is that the program is structured such that it is executed several times, ie. each time the Step is scanned.

Perform functions are Functional Terms and must be the only Terms in the Statement.

Table Functions

The set of table functions (individual descriptions to follow) allow the User to perform several functions using data in a table or array type fashion.

Tables in General

All of these functions work on the same tables or arrays of data. A Table is a two dimensional array of values that can be either floating point numbers (float), integer numbers (integer), sequence of characters (string) or binary numbers (digital I/O status). Every element of that table must be of the same type and when the Table is defined the type of variable is established.

There can be a maximum number of 100 Tables, each assigned a unique number from 1 to 100. These tables can be of any size determined by the number of rows and columns assigned to them until the maximum amount of memory allocated for Table use is consumed. There is 20K bytes of memory reserved for use with table functions.

A Table must first be defined before it can be used. This is done using the four Table Define functions. The Swap_Table_Value functions named Swap_Table_Value_Int, Swap_Table_Value_Float, Swap_Table_Value_Dig, and Swap_Table_Str allow the User to either write a value from a variable of the same type into a Table element, or read a value from a Table element into a variable of the same type.

The Init_Table functions named Init_Table_Integer, Init_Table_Float, and Init_Table_Digital allow the user to store multiple values into a Table at one time. There is no initialization function for a string table. The Copy_Table_To_Table function allows the User to copy one Table’s value into another Table.

All Table functions check to make certain that when a Table is selected for use, it fits the definition that has been previously creates. Thus a Swap_Value to a Table defined as

integer must have an integer variable, or both Tables selected in a Copy_Table_To_Table must be of the same type, or a Swap_Value cannot refer to a row number for a table that is greater than the total number of rows defined for that Table.

In the event of misuse of the functions being detected either UPTime will produce an error at download time or (and this is most likely) an error message will be generated at run time.

It should be noted for all the Table functions the row number comes first followed by the column number.

Define_Table

The User must define every Table he Uses with this function. When this function is selected from the Perform menu, a menu will display the following for the User to select:

Number_of_table	the Table number from 1 to 100
Type_of_table	the type of variables (I for integer, D for digital or F for float, S for String) stored in the Table elements
Number_of_rows	the number of rows of the table
Number_of_columns	the number of columns
Save_value_over_halt	indicates whether the Table should be saved through a halt−run cycle

At Diagnostic Processor run time, a non–critical error with a message will be generated if the Table number specified has already been defined. The Table will not be re–defined and the original definition will be retained. To avoid this error, the User should place this function in a Step that only executes once.

The only other non–critical run time error that can occur is if the table number specified is greater than 100.

Entering and Retrieving Table Values

There are four Swap_Table_Value functions that are exactly the same except they work on the three different types of Tables.

Swap_Table_Value_Int
Swap_Table_Value_Flt
Swap_Table_Value_Dig
Swap_Table_Value_Str

These functions allow the User to write a value from a variable of the same type into a Table element or to read a value from a Table element to a variable. There are four distinct functions of different types so that UPTime can check to make sure the variable named is of the same type as the Table specified.

When the User selects one of these functions from the Perform menu the following information will be requested:

Number_of_table	the Table number from 1 to 100
Type_of_operation	the type of operation (R for read or W for write) to be performed with R meaning to take a value from the table and place it in the variable, and W to enter the value of the variable into the table.
Row_number	the row number of the element to be read from or written into
Column_number	the column number of the element to be read from or written into
Variable	the name of the variable to be used to store the Table value during a read or get the value during a write

The Diagnostic Processor will generate run time critical errors if the Table selected does not match the type of Swap being used or if the row and column numbers are out of range for the selected Table.

Initializing Tables

The three Init_Table functions allow the User to set values for multiple elements in a Table at one time. There are three distinct functions of different types so that UPTime can check to make sure the variable named is of the same type as the Table specified. There is no initialization function for String Tables.

Init_Table_Int
Init_Table_Flt
Init_Table_Dig

When the User selects one of these functions from the Perform menu the following information will be requested:

Number_of_table	the Table number from 1 to 100
Row_number	the row number of the first element where the values listed are to be stored
Column_number	the column number of the first element where the values listed are to be stored
Number_of_values	the number of values that will be stored in the following consecutive Table elements
Value_1	a constant or the name of a variable (of the same type as the Table) to be stored in the first Table element identified by the Row_number and Column_number
Value_2	a constant or the name of a variable (of the same type as the Table) to be stored in the first Table element after the Table element identified by the Row_number and Column_number
..	
..	
Value_28	a constant or the name of a variable (of the same type as the Table) to be stored in the last Table element identified by the Row_number and Column_number

There can be up to 28 values initialized with each individual function and they can begin at any Table element location. Each column element of a row is filled in before the next value is placed in the next row.

The Diagnostic Processor will generate run time critical errors if the Table selected does not match the type of Swap being used or if the row and column numbers are out of range for the selected Table or if the number of values added to the starting element position would go beyond the last element defined for the Table.

Copy_Table_To_Table

The Copy_Table_To_Table function allows the User to copy one Table's values into another Table. The Tables must be of the same type and the Table that the values are copied from must have a less or equal number of rows and columns than the other Table.

The Table's elements will be copied into the corresponding Table's elements. If the first has less rows or columns than the second, then the elements that do not exist in the first Table will be left unchanged in the second Table.

When the User selects one of these functions from the Perform menu the following information will be requested:

Table_To_Copy_From	the number of the Table from which the values will be copied. Its row number and column number must be less than the other Table identified
Table_To_Copy_Into	the number of the Table the values will be written into.

The Diagnostic Processor will generate run time critical errors if the two Tables selected are not the same type or if the Table to copy from is larger than the other Table.

Table Uses

There are many uses for the Table functions. As an example, the Table functions are valuable in applications where the set up of parameters varies depending on the product under manufacture on the process line. Batch process recipes or flexible manufacturing assembly lines are examples.

The UPTime program can be written using English name variables for parameters throughout with statements such as:

If Oven_temp_1 is greater than Melting_point ...

used throughout the program description of the process. Then in a Step, lets call it the Select_Product Step, by using the Swap_Table_Value_Flt function, the variable Melting_Point can be made equal to one of the elements of Table 1, where Table 1 contains the parameters for this particular product run.

Using the Init_Table_Float, Tables containing parameters for each style of product that can be made on the line can be initialized. When the Operator selects a style of product in the Select_Product_Style Step, the Copy_Table_To_Table function can be used to move those parameters into Table 1, the Table in which Melting_Point finds its values for this style and product run.

BCD I/O Representation

General

At times input and output devices are used for data entry or display that use BCD representation. Thumb wheel switches and LCD displays are possible examples.

The devices are connected either to digital inputs or digital outputs where 4 hardware inputs or outputs represent 1 digit of the display. The display or switch then uses a binary code representation from 0 to 9. There are 16 total possibilities (4 outputs or inputs represent 2 to the 4th or 16 possible combinations), and the remaining 6 are used for such things as minus sign, decimal point and null or space character on a display.

The two functions, BCD_In_Conversion and Output_BCD_Conversion allow the User to designate a series of consecutive digital inputs or outputs to be treated as if they are groups of 4 BCD digits. The functions then translate between the I/O and either Diagnostic Processor integer or float variables.

BCD_In_Convert

The BCD_In_Convert translates between a series of digital inputs and an integer or float variable. The User specifies the name of the first digital input in the series and the number of BCD digits (4 inputs per digit) that represent the variable. The User also specifies the type of variable and the name of the variable to store the converted value.

The inputs will be taken in hardware consecutive order, and the number of digits can be on more than one block or card as long as the cards have consecutive addresses. The User only need define with an English name the first digital input and initialize the blocks or cards involved.

BCD uses the standard binary representation for the numerical digits 0 to 9. There is no true standard for the minus sign or decimal point character. Therefore the function has the provision for the User to optionally specify the hexadecimal number, #A, #B, #C, #D, #E, or #F, (where # means hexadecimal number to UPTime), that is the pattern for these two characters.

The function parameters are:

Starting_input	the English name of the first digital input in the string of consecutive inputs that form the BCD digits
Number_of_BCD_digits	the number of BCD digits. The number of digital inputs in the string will be 4 times the number of digits.
Variable_name	the name of the variable to store the translated value
Variable_type	the type of variable either integer or float
Minus_sign_pattern	a hexadecimal number from A to F that gives the pattern for a digit that should be recognized as the minus sign.
Decimal_point_pat	a hexadecimal number from A to F that gives the pattern for a digit that should be recognized as the decimal point. Note the minus sign and decimal point are optional. A minus sign can be specified without a decimal point, but if a decimal point is specified the minus sign must also be specified.

Output_BCD_Convert

The Output_BCD_Convert translates between a an integer or float variable and a series of digital outputs. The User specifies the name of the first digital output in the series and the number of BCD digits (4 outputs per digit) that represent the variable. The User also specifies the type of variable and the name of the variable where the value to be converted is stored.

The outputs are in hardware consecutive order, and the number of digits can be on more than one block or card as long as the cards have consecutive addresses. The User only need define with an English name the first digital output and initialize the blocks or cards involved.

BCD uses the standard binary representation for the numerical digits 0 to 9. There is no true standard for the minus sign or decimal point or null (space) character. Therefore the function has the provision for the User to optionally specify the hexadecimal number, #A, #B, #C, #D, #E, or #F, (where # means hexadecimal number to UPTime), that is the pattern for these three characters.

The function parameters are:

Starting_output	the English name of the first digital output in the string of consecutive outputs that form the BCD digits – all of the other outputs used must be defined
Number_of_BCD_digits	the number of BCD digits. The number of digital outputs in the string will be 4 times the number of digits.
Variable_name	the name of the variable that is to be translated and output
Variable_type	the type of variable either I for integer or F for float
Minus_sign_pattern	a hexadecimal number from A to F that gives the pattern for a digit that should be recognized as the minus sign.
Null_character_pat	a hexadecimal number from A to F that gives the pattern for a digit that should be recognized as the null or space character.
Decimal_point_pat	a hexadecimal number from A to F that gives the pattern for a digit that should be recognized as the decimal point.
Number_decimal_dig	the number of decimal digits; the digits to the right of the decimal point, that should be output. Note: If the Number_decimal_dig parameter is used then the Decimal_point_pat is not optional and must also be used.

Note: if the value is too large to display in the Number_of_BCD_digits specified, but there is enough room for all the digits to the left of the decimal point, those digits will be displayed and no error will be generated. If there is not enough digits for all the numbers to the left of the decimal point, the display will not be output and a non–critical error will be generated.

Shift_Register

The Shift_Register function allows the User to shift values from one integer variable to another by a User selected number of bits.

The User can define up to 28 integer variables and connect them together to form a shift register. The User then can shift the contents of each variable by a selected number of bits (up to 64) to the right or left.

The User can choose to have the shift behave in a circular fashion where the bits from the variable farthest to the right shift to the farthest variable to the left. Or the shift can be a fill type shift where the bits fall off the end and do not circle back to the first integer. In the case of a fill type shift, the User can choose the value (0 or 1) that is placed in the locations left empty by the shift.

If more than 28 integers are to be connected, two shift registers can be cascaded by having variable_28 of the first shift be the same as variable_1 of the next shift.

The User defined function parameters are:

Number_of_bits	the number of bits to shift the registers, 0 to 63
Shift_direction	either R right or L for left. The direction of the shift.
Type_of_shift	either C for circular or F for fill, The way the shift will act when the last variable is reached.
Fill_value	either 1 or 0. The value that will be placed in the bits of the variables that have been shifter in a F or fill type shift. No meaning in a circular shift.
Variable_1	the name of the first integer variable and therefore the variable farthest to the left in the shift register.
Variable_2	the name of the second integer variable in the shift register.
..	
..	
Variable_28	the name of the 28th or last integer variable and therefore the variable farthest to the right in the shift register.

Note: only Variable_1 must exist, all others are optional.

String Manipulation

The String_Manipulation function allows the User to perform various functions upon a string variable. String variables can be up to 80 characters long and often are used for inputting data from an ASCII oriented device such as a bar code reader, or outputting to a similar device such as a scale or robot.

General

In many cases the 80 characters is not one piece of data but a series of sub–strings each containing unique data. Thus the ability to extract the sub–strings, convert the data to integer or float and store in a variable and store a sub–string into a larger string is sometimes needed.

The string manipulation function is in reality several functions in one. The User defines the name of the string to be manipulated and the number of the starting and ending character, if a sub–string within that string is of interest. The start and end must be less than 80 , the first character is number 1, and the start must be smaller than the end or a run time error is generated.

The User also defines the type of manipulation (see following sections) and a reference variable that is used by the operation. The parameters defined by the User for the function are:

String_name	the name of the string to manipulate
Start_character_num	the number of the starting character used in the string. The first character is 1
End_character_num	the number of the ending character used in the string. The last character is 80.
Operation	a character code that defines the operation to be performed (see following section for definition)
Reference_variable	the name of the reference variable to be used in this manipulation. The variable type required depends on the Operation and the Diagnostic Processor generates a critical error if a mismatch occurs.
Search_Character	the name of a Character_Variable or the actual character to be matched by this operation. This is an optional parameter which only needs to be entered when the match (M) operation is chosen.

Operations

E for Extract

If the Operation character is an E, the function will extract the sub–string defined by the starting and ending characters and copy those ASCII character numbers to the string variable named in the Reference_variable.

s For store in sub–string

If the Operation character is an 's', the function will use the string variable named in the Reference_variable as a sub–string, and store those ASCII characters in the sub–string defined by the starting and ending character numbers. The characters in the string named as the Reference_value will be stored until the end of that string is reached or until the last character number in the main string is reached.

I for extract and convert to Integer

If the Operation character is an I, the function will extract the sub–string defined by the starting and ending character numbers and convert those ASCII characters into an integer value and store that value at the integer variable named in the Reference_variable.

i For Convert integer To ASCII And Store In String

If the Operation character is an i, the function will convert the value stored at the integer variable named in the Reference_variable to ASCII and store that value into the sub–string location defined by the starting and ending character numbers.

F for extract and convert to Float

If the Operation character is a F, the function will extract the sub–string defined by the starting and ending character numbers and convert those ASCII characters into a float value and store that value at the float variable named in the Reference_variable.

f For Convert float To ASCII And Store In String

If the Operation character is a f, the function will convert the value stored at the float variable named in the Reference_variable to ASCII and store that value into the sub–string location defined by the starting and ending character numbers.

C for Concatenate

If the Operation character is a C, the function will concatenate or add the string of characters named in Reference_variable to the main string. The resulting main string can not exceed 80 characters, so the addition of the Reference_variable characters will be truncated at that time.

L for string Length

If the Operation character is an L, the function will calculate the number of characters in the string, and put that number into the integer variable named by Reference_variable.

M for Match the Given Character with a Character in the String

If the operation character is an M, the function matches the character in the Search_Character parameter to the first character in the sub–string designated by starting character and ending character values. The position of the first match is returned in the Reference_Variable.

Errors in General

The functions check to make sure when conversions to or from ASCII are performed that legal values will result and produce errors if they do not.

Specialized Perform Functions

All off the above functions have specific parameters which are passed to the function. These parameters are all chosen by filling in similar forms which specify parameter type and whether or not it is required. The following performs have unique ways that the operations are specified.

Display Date and Time

This function displays the current date and time in the desired format. After choosing this option a form is displayed to enter the Name of the Step that is created, the format of the display, the communications port to send the information and the Step to branch to after the operation is completed.

Get User Input

This function enters program text used to retrieve information through a communications port. A form is displayed for entering the following options:

- Current Step – Name of the Step that is created.
- Clear Screen – Option of clearing the screen before the prompt is displayed.
- Screen Message – Prompt telling operator to enter some information.
- Input Variable – Variable that stores the input
- Comm Port – Which port is used.
- Branch To Step – Step that becomes active when this process is completed.

User Menu

This function will display a menu of up to 10 items and then wait for the user to enter a selection. If the selection is valid, it will branch to desired Step for that selection. This is very useful when creating a user interface for the Diagnostic Strategy.

Miscellaneous Programming Operations

This section discusses several different aspects of using Variables in State Logic programming. There is also a list of grammatical rules and a discussion of PID loops.

The VARIABLES option on the menu is used to define most of the variable types. The topics discussed here are numeric data types, variable types, and mathematical calculations.

Numerical Data Types

There are two numerical data types integer and floating point. Integer data is limited to the range of –32768 to 32767. Floating point data is limited to the range of +/–1.2E–38 to +/–3.4E+38 with an effective precision of seven decimal digits.

Integer constants are whole numbers in the range –32768 to 32767. Integer constants can be specified in hexadecimal format by preceding the number with '#', ie, #77FF. Floating point constants are numbers using decimal points, numbers outside the range for integers, or numbers using scientific notation.

Data types maybe mixed freely within expressions. If any of the operations in an expression require floating point notation, all of the data elements are converted to floating point values. If a floating point value is assigned to a variable of integer variable type, the floating point value is converted to an integer value observing the following rules:

1. All values are rounded to the nearest number.
2. Values outside the integer range are clipped to –32768 or +32767.

Floating point operations require more time to perform than integer operations. Therefore, refrain from floating point operations as much as possible if response time is critical to your application.

The data type of analog values is floating point. The data type of time values (seconds, minutes, etc.) is integer.

Numeric Value = <Numeric Constant> |
<Calculation> |
<Numeric Variable Name> |
<Analog I/O Name> |
<PID Value>

Calculation = < Numeric Value > < Operator > < Numeric Value > |
< System Functions > (< Numeric Value >)

System Functions = SIN | COS | TAN | ARCTAN | SQRT | EXP | LN | RANDOM

Numeric Constant = <Floating Point Number> | <Integer Number>

PID Value = <PID Loop Name> <PID Parameter Keyword>

Character Value = < Character Variable Name > | ' <Character> '

String Value = <String Variable Name > |
" < Character String > " – Up to 80 characters

Variables

Variables are used to store changeable information in memory. All variables are identified by a unique name that is assigned to a specific memory location. Each variable can be configured so that its value is saved when the program stops or be initialized every time the program is started. There are four main categories of variables, numeric, character, digital, and device. Calculations may refer to the value stored in any of the numeric variables.

Analog Channels

Analog Variables: The values of analog I/O channels connected to the PLC system and stored in the %AI and %AQ CPU memory locations. Analog variables may be scaled in UPTime so that engineering units are used instead of the raw data. The scaling option is selected when the analog variable is defined. Unscaled analog variables use the raw values used by the analog modules. Scaled values are floating point values while unscaled channels are integer values.

Digital Points

Digital points store only one bit of information as either ON or OFF. These variables are all stored in the CPU of the system. The types of digital points are %I, %Q, %M, %G, %T, %S, %SA, %SB, %SC.

Except for the status bits (%Sx), which must all be inputs, digital points can be either inputs or outputs. If the Diagnostic Processor controls the value of these points they must be defined as outputs. If some other device, such as the CPU or an input module, changes the value of these points, then they must be defined as inputs.

State Logic digital outputs are always OFF by default. A Digital output is only ON when a Statement in an active Step is turning it ON.

Another digital variable is the internal flag. These variables are stored in the Diagnostic Processor and are used to communicate a status condition from one Task of the State Logic program to another. Internal flags are defined when the Sequence editor is in use. Press <F3> for the menu and then select the LIST option and then the "Internal Flags" option. Flags can also be defined when searching for undefined words.

Register Variables

These variables are stored in the %R registers in the CPU of the system. The Register Variables can be configured to be either integer or floating point values. An integer type uses one %R register and floating point type uses 2 %R registers. The Diagnostic Processor accesses these values across the PLC backplane.

Register Variables can be either inputs or outputs, as can other variables stored in the CPU. Designated Register Variables as outputs only if the State Logic program changes that register value, otherwise the variable must be defined as an input. Register Variables defined as outputs cannot be written to be the WRITE term.

Numeric Variables

Integer and Floating Point Variables are stored in the Diagnostic Processor, therefore these variables are not designated to be inputs or outputs. When used in calculations, the variable types may be mixed freely. If an integer variable stores a floating point value the fractional part of the number is truncated. If a number outside the range of an integer values is assigned to an Integer Variable, a math overflow run–time error results.

Floating point calculations are slower than calculations with integer values. When the types are mixed, the calculation is done as a floating point calculation.

Digital Device Variables

A Device is a unique higher level UPTime structure that treats a group of Digital Points as one entity. A Device has many uses but is designed primarily to represent mechanisms that cycle between two states and usually including some travel from one state to the other.

Devices usually have two digital outputs, each causing the mechanism to go to a specific state. Normally Devices include two inputs, each signalling that the mechanism's travel from one state to another is complete. A Device Diagnostic monitors the time that it takes for the mechanism to change state once the forcing output starts the change. The Device Diagnostic can also be setup to trigger every time the mechanism reaches either state.

Mechanisms that can be represented by the Device structure are presses, hydraulic rams, motors, and clamps. This structure is very flexible and can therefore be used for several purposes. It is not necessary to use an output for the point causing a change or use an input for points that detect the change, in fact any of the different types of digital points can be used for any of the points that define the Device. The same digital point can be used more than once in the Device definition. Use Devices and the Device Diagnostic when the time from the point that one digital point changes state until another digital point changes state is critical.

A Device is defined by filling in the blanks in a form. The Device is given a name by filling in the name blank. The Device terminology is designed to describe an implement that opens and closes. The blanks for the two outputs are labeled Open Actuate and Close Actuate, and the blanks for the two inputs are labeled Open Complete and Close Complete.

Names must be defined before used in the form. The same naming capabilities are used in the Device form as in the Diagnostic forms. A name can is attached to an I/O point by using UPTime menus and forms. A list of defined digital points appears when the <Ins> key is pressed. The name blank may also be a direct memory address specification such as I76 or Q45, or a name may be a nickname used in Logicmaster when the Ladder Logic program controlling the process was created.

In the Device form there is a normally open or normally closed designation under the blank for each I/O name. For outputs this designation says "Actuate When On" or "Actuate When Off", and for the inputs the designation says "Complete When On" or "Complete When Off". The "Actuate When Off" designation is used for the many mechanisms that start the operation when the output is Off.

Besides being used in the Device Diagnostic, devices can also be used in the State Logic program. The program can test whether the Device is ON or OFF as in the following Statement referring to a Clamp defined as a Device:

If the Clamp is ON, . . .

ASCII Variables

The ASCII variable types are Character, String, and Register. Character variables store one character and use one byte of memory. String variables store up to 80 characters and use 80 bytes of memory. String variables store any characters including control characters. To use a control character in a string variable, enter the per cent character (%) followed by the pound sign character (#) and then two digits that are the hexadecimal number for that control character. For example:

Make Test_String equal "abc%#1Bxyz".

is an assignment to a string variable that stores the characters a, b, and c then the escape character and then x, y, and z.

The %R registers can also be used to store character information in the CPU. Use the WRITE term to send characters to the %R Register locations and the READ term to retrieve characters from those locations. Two characters are stored in every %R register. See the sections describing the READ and WRITE terms in this manual.

DO NOT WRITE to register locations used as Register Variables and designated as outputs. UPTime overwrites values in the output Register Variables every scan.

Time Variables

The CPU has a clock that maintains the current month, day, day of the week, hour, minute, and second. These values are always available to the Diagnostic Processor through the Time Variables Month, Day, Day_of_Week, Hour, Minute, and Second.

These variables represent an integer values that may be used by program statements or accessed using the DISPLAY debug mode menu option. These are READ–ONLY values that cannot be change from the Diagnostic Processor. To change the clock settings use Logicmaster to change the time values in the CPU.

CAUTION: Any changes to the Time Variables while the program is running, may affect the execution of any timing Statements currently active in the program.

The current time values may also be saved using the Time Variables:

Make Start_Hour = Hour, Start_Minute = Minute, and Start_Second = Second.

Use this flexible method of saving the current time to create any type of elapsed time timer needed. Subtract the saved variables from the current Time Variables to get the elapsed time.

Number_Of_Hours = Hour – Start_Hour.

This Statement gives the number of hours that the clock has changed since the Start_Hour variable has been set. Be careful to account effects of other Time variable rollover, such as minutes going from 59 to 00.

Mathematical Calculations

Calculations may use any of the Numeric variables, Register Variables, Analog Channels, and Time Variables. Calculations in Functional Expressions use any of the operators on the right side of the EQUAL sign and assign a value to the variable on the left side of the sign. In Conditional Expressions a numeric calculation may be used on either side of a comparison operator.

Mathematical calculations are used in Functional Terms as in this assignment Term:

Make Pointer_Position = Last_Position * (Forward_Pressure + 345.8)

and in Conditional Terms as in this comparison term:

If Advanced_Magnitude < SIN(Current_Angle) / 45.6 go to Reposition Step.

Numerical expressions may be much more complicated using any of the operators in any order and nested in parenthesis to change order of evaluation or make the expression more readable. Up to 18 levels of parenthesis may be used.

Calculations Using Analog Channels

Calculations can use analog channels as values in any calculation. A calculated value can also be directly assigned to an analog output channel. The following Statements use analog channels (underlined) in calculations.

If TankLevelSetpoint + 13.5 < TankLevel * 2, then go to the Restart Step.

Make MotorSpeed = Demand * SIN(CurrentLoad).

Operator Precedence

Operators are executed according to the precedence listed in the operator keyword table. The operators with the lowest precedence number are executed first. Operators with the same precedence are executed left to right. Use parenthesis to change the order of execution.

Grammatical Rules

- Every Triggered Sequence and Instructional Sequence must have an End_Sequence keyword in one of its Steps.
- The When_Done_Next_Step keyword must be the only word in the Statement.
- Every Task must begin with the word "Task:" followed by the Task name.
- Every Step must begin with the word "Step:" followed by the Step name.
- Every Statement must end with a period (".").
- Only one "Go" is allowed per Statement, but there may be several in a Step.
- If a "Read" Term is used in a Statement, it must be accompanied by a "Go" in the same Statement. There may be no other Terms in the Statement.
- Only one "Read" is allowed per Step.
- A "Perform" function must be the only Term in a Statement.

PID Loops

The Adatek Diagnostic Processor provides the capabilities of modulating control through the use of the PID algorithm. Each Adatek Diagnostic Processor provides the User with up to ten PID algorithms that are continuously executed at User selected time intervals. These PID algorithms can be connected to field inputs and outputs, or interconnected in cascaded and other fashions to implement the User's desired strategy.

Those who are already familiar with traditional PID control, might want to skip the **PID Algorithm Philosophy** section and go directly to the **PID Summary** section. The **PID Summary** section is a summary of the features available with the Diagnostic System and is probably the only section that those familiar with PID loops need read. For those Users needing or wanting a more detailed discussion, the **PID Algorithm Philosophy** section provides the details.

PID Algorithm Philosophy

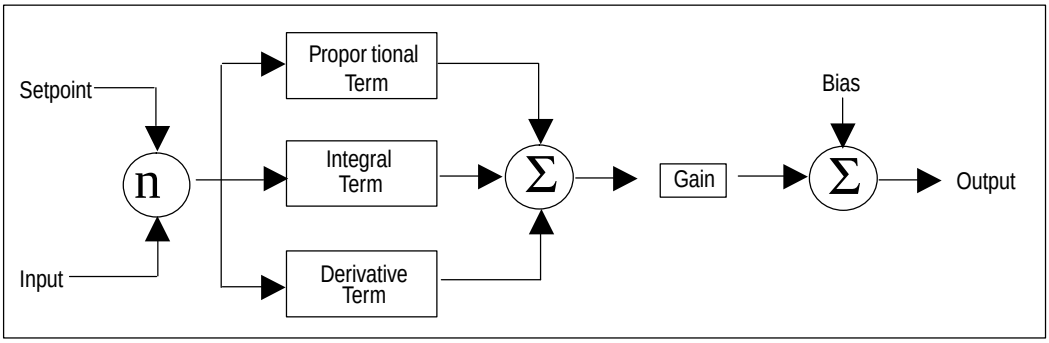
The Adatek Diagnostic Processor PID employs a traditional algorithm that compares a setpoint with a process variable to generate an error signal. The error signal is acted upon by any or all of three parts; proportional, integral, or derivative, and the resulting output is the action that should be taken by the process actuator.

Each of the three parts has a tuning constant associated with it that can be adjusted to affect how the control action occurs. The Proportional part or term uses the Gain tuning constant. Its result is simply the product of the error, the difference between the process input and the setpoint, multiplied by the Gain. It is an instantaneous value that changes as the error changes.

The Integral term uses the Reset tuning constant. Its result is an accumulation of the product of the error signal times the Reset over time. Even though an error signal is currently zero value, the integral portion may provide a result because previous error signals have accumulated.

The Derivative term uses the Rate tuning constant. The derivative term's result immediately allows an error signal to have its full effect, then returns the term's value to zero as time goes on. The amount of the Derivative term output for a given error and the rate it decays is affected by the value of the Rate tuning constant.

The total output of the PID is the sum of the results of the three terms. Figure 1 shows a simplified diagram of the algorithm. Typically the Proportional and Integral terms are used more often alone without Derivative because this provides a more stable control performance. The Derivative term allows more anticipation and quick response, but at a penalty of possible over response and undesirable process disturbances.



When the process variable differs from the setpoint, such as at the time of a step change in the setpoint, the proportional term immediately causes the output of the loop to change. As time passes the integral term integrates the output in the same direction. The action of the processor hopefully brings the process variable closer to the setpoint. This causes the error to become smaller and decreases the proportional term, but the integral term continues to increase as it adds on the error signal over time.

Ultimately the process variable equals the setpoint and the error is zero causing the proportional term's value to be zero. In addition, the integral portion is no longer changing because the error is zero, therefore the output remains constant equal to the value the integral term accumulated. Any changes in process variable or setpoint cause an error and the processor will integrate to adjust the output to bring the system to equilibrium.

The addition of the derivative term, makes the output react more extremely when the error is first detected. Then as a function of the Rate tuning constant, this reaction decays out allowing the integral term to bring the system into balance and remove the error.

Simple PID Loop

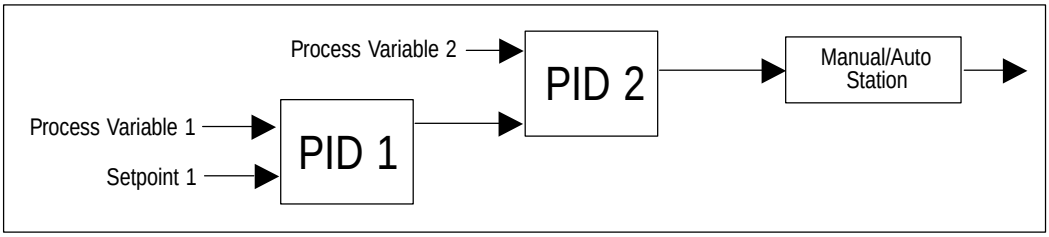
Many applications only require a simple PID loop to achieve the desired control results. A process variable represented by an analog input is compared with a setpoint with the output of the PID loop being directly sent to a field actuator by means of an analog output.

To set up this simple loop, the User needs only choose the "Define PID" function from the "Define" menu in UPTime. The User then enters the name of the analog input, the name of the analog output, and the various initial settings for the tuning constants.

The setpoint and tuning constants can be changed while the process is on line by using the "Tuning" function in the Debug mode of UPTime.

Complex PID Control

In some cases a more complex strategy using PID algorithms may be desired. Rather than the output of the PID going directly to an analog output, a cascaded PID strategy can be used in which the output of one PID goes to the input of another PID.



In this example PID 1 compares process variable PV1 with the desired setpoint. The output is then fed into PID 2 as a setpoint and is compared with process variable PV2. Generally the second PID (PID 2) called the downstream PID, will be tuned to have a faster response time. It will act first to move the output quickly in the right direction, and then the slower acting upstream PID will act to integrate out the error between the control variable and its desired setpoint.

The PID algorithm can use an analog input as its input, as in the case of PID 1 or any floating point variable. In the case of PID 2, the input to PID 2 can be defined as the output of PID 1. Likewise the setpoint can be a constant, or a named variable or as in this case, an analog input. To set this strategy just use the “PID Define” menu and name the input and setpoint.

Bumpless Transfer

Figure 2 also shows a Manual / Auto station between the PID output and the actual analog output card. This station allows the User to place the station in Manual, and then by means of Raise and Lower pushbuttons, change the actual value of the Manual / Auto output. In the Auto mode, the value of the Manual / Auto station is equal to the output of the PID.

When the M / A station is in Manual, the station value can and usually will be forced to a value other than one that will make the setpoint equal the process variable input. If the PID followed its normal operation, the integral term would continue to integrate because of the error signal between the process variable and setpoint. This would leave a difference between the PID output, which is the Auto input to the M / A station, and the actual M / A output. Then when Auto mode is selected, this difference would cause a jump or bump in the M / A output. This would upset the process and is desirable to avoid.

To prevent this bump from happening, the PID needs to have another mode of operation besides its automatic mode. In this mode, called tracking, the PID output will be maintained at what ever value it is set to, such as the M / A output in this example. The PID will not perform its normal arithmetic, but will instead set itself so that when tracking is removed, the PID output gradually goes to the proper value and avoids the bump. This is called bumpless transfer.

Each of the ten PID algorithms have logical signals associated with them that use the name Track_Mode and the actual PID name. As an example, if the User named the first PID algorithm Tank_Level then the logical signal would be called Tank_Level Track_Mode. When Track_Mode is made true, then the PID automatically discontinues its normal algorithm, and begins tracking its output and preparing for bumpless transfer.

The User can set the Track_Mode variable from any active Step. Using this variable, the User can create any tracking strategy he desires.

Anti–R eset Windup

Using a cascaded PID strategy can cause the User some subtle problems. In the example shown in Figure 2, if PID 2 has reached its maximum and PID 1 still has an error signal because the setpoint does not equal PV1, then PID 1 would continue to integrate. The output of PID 1 would continue to increase, but it would have no affect on PID 2 since it is already at its maximum. However when the error signal of PID 1 reversed direction and caused PID 1 to begin integrating in the opposite direction, PID 1 would have to integrate below the threshold value it was when PID 2 reached its maximum before it would have any affect on PID 2. This excess amount of output PID 1 has accumulated is commonly referred to as reset windup.

The upstream loop, PID 1, needs to be prevented from winding up. An input called Block_Up will transfer the PID algorithm into a mode such that it will not integrate in the Up, towards 100%, direction. This is called anti–reset windup.

The PID will still be able to integrate down if the error signal reverses direction. That is, if the process variable is less than the setpoint the PID will not integrate up. If the process variable becomes greater than the setpoint the PID will integrate down. The Block_Down input works exactly opposite.

In the case of Figure 2, if the setpoint for PID 1 is greater than PV1, PID 1 output will continually increase. This is the setpoint for PID 2 and assume it is already greater than PV2 and PID 2 has reached its high limit. There is no profit in PID 1 output getting larger since PID 2 can not respond to its demand, PID 2 is already at its limit. Therefore the User should set the PID_1 Block_Up input true and stop the PID from winding up further.

Then when either PV2 increases or the high limit on PID 2 is changed which will allow further action by PID 2, the PID_1 Block_Up input can be set false and PID 1 can resume integrating. Or if PV1 rises above the setpoint in response to the control action, PID 1 will begin integrating the output of PID 1 in the lower direction which is permissible. When PID 1 output falls below the input PV2 into the downstream PID 2, PID 2 will integrate down below its high limit and remove PID_1 Block_Up and return the complete loop to normal.

Using the Block_Up and Block_Down inputs any amount of cascading of PIDs can be accomplished without reset windup occurring and with bumpless transfers from one mode to another.

Tuning and Scaling

The Gain, Reset, and Rate constants can be adjusted to obtain the desired PID performance, such as speed of response and over shoot. In addition, each PID has a high limit and low limit that can be set to limit the output to less than its full range when desired. These limits automatically employ anti–reset windup.

Within the PID algorithm, all signals are treated as being 0 to 100%. Each input and the output and the high low limits, have scaling constants associated with them. Values given for this parameter are converted to a percentage of the range specified by the scaling constants.

If these constants are left blank at programming time, that input is assumed to be already 0 to 100%. Values given for this parameter are assumed to be a percentage which has already been scaled.

By using the scaling factors, the output can be scaled to have a live 0, that is go from –100% to +100%. This is a valuable tool at times when cascading PIDs and the upstream PID needs the ability to overcome and move the downstream PID across its full range.

PID Summary

The Adatek state engine provides up to ten PID algorithms. Each algorithm is identical and can be executed at User selected time intervals with the minimum interval being 1 second. The PID algorithms can each be independent or can be cascaded together with the output of one PID becoming the input to another. Each PID features built in bumpless transfer and anti–reset windup features.

Each PID is initialized in the Program mode of UPTime using either the LIST or DEFINE option from the program mode menu. PID Loops are tuned on–line in the Debug mode. Each PID loop is given an alpha numeric name which can be up to twenty characters in length. All references to the PID loop use this name.

To start a PID Loop use the Start_PID keyword. For example the statement:

Start_PID Tank_Level.

starts the PID Loop name Tank_Level.

To stop a PID Loop use the Stop_PID keyword. For example the Statement:

Stop_PID Tank_Level with 10.

stops the PID Loop named Tank_Level and sets the output to 10.

PID Parameters

Each PID loop is defined by the values of several parameters. These parameters are initialized when the PID loop is first defined, and can be changed by program statements during program execution or through the tuning forms provided in debug mode. The program refers to these parameters by specifying the name of the PID loop followed by the parameter keyword. For example, the UPTime program statement:

Make Tank_Level Setpoint = 45.

sets the setpoint of PID loop named Tank_Level to 45.

The following table lists each of the parameters with a description and keyword to identify the parameter in an UPTime program statement.

Table 8-7. PID Loop Parameters

Parameter	Keyword	Description
Action Direct or Inverse	Loop_Action	Set to D or I to make PID integrate from 0 toward 100% if the setpoint> process variable (direct acting) or process variable > setpoint (inverse acting.)
Update Time	Update	Time interval between updates for this PID.
Gain	Gain	Gain for the PID
Reset	Reset	Reset constant for this PID
Rate	Rate	Rate constant for this PID
Setpoint	Setpoint	Name of the variable acting as setpoint or a value to use as the initial value.
Setpoint Max Scale	SP_Max	Engineering unit value for 100% scale or blank to use 100%
Setpoint Min Scale	SP_Min	Engineering unit value for 0% scale or blank to use 0%
Process Variable	Process_Var	Name of the process variable
Process Var Max Scale	PV_Max	Engineering unit value for 100% scale or blank to use 100%
Process Var Min Scale	PV_Min	Engineering unit value for 0% scale or blank to use 0%
Control Variable	Control_Var	The output of the PID loop – analog output channel or floating point variable.
Control Var Max Scale	CV_Max	Engineering unit value for 100% scale or blank to use 100%
Control Var Min Scale	CV_Min	Engineering unit value for 0% scale or blank to use 0%
Bias	Bias	Amount to be added to the Output
High Limit	High_Limit	Maximum allowable value for the Output
High Limit Max Scale	HL_Max	Engineering unit value for 100% scale or blank to use 100%
High Limit Min Scale	HL_Min	Engineering unit value for 0% scale or blank to use 0%
Low Limit	Low_Limit	Minimum allowable value for the Output.
Low Limit Max Scale	LL_Max	Engineering unit value for 100% scale or blank to use 100%
Low Limit Min Scale	LL_Min	Engineering unit value for 0% scale or blank to use 0%

There are also PID status and command bits associated with each PID loop. These bits indicate information about the status of the PID loop or may be used to control the PID loop. To use these bits in an UPTime program, the PID name is used followed by the keyword for the bit.

The UPTime statement:

If Tank_Level High_Limit_Status is true, go to Manual_Mode Step.

makes Manual_Mode the active Step of the Task if the output of Tank_Level PID loop is at the high limit value.

The UPTime statement:

Set_Bit Tank_Level Track_Mode.

makes the Tank_Level PID loop tack the output so that there is no error signal.

Table 8-8. PID Command and Status Bits

Status or Command Bit	Keyword	Description
Block Up	Block_Up	When this bit is set, the PID does not integrate up for a positive error signal. Used for Anti_Reset Windup.
Block Down	Block_Down	When this bit is set, the PID does not integrate down for a negative error signal. Used for Anti_Reset Wind-up.
Track Mode	Track_Mode	When this bit is set, the PID tracks the output so that no error is calculated. Used for a bumpless transfer to automatic mode.
High Limit Status	High_Limit_Status	When this bit is set, the PID output has reached the high limit parameter or Block_Up for this PID is true. This is a read–only bit.
Low Limit Status	Low_Limit_Status	When this bit is set, the PID output has reached the low limit parameter value or Block_Down is true for this PID loop. This is a read–only bit.

PID Inputs

There are 5 inputs to the PID algorithm. Two of the inputs are the process variable and setpoint. These are either analog inputs or calculated real variables such as the output of another PID. The setpoint can also be a constant which can be adjusted on line from the PID tuning menu available in the UPTime Debug Mode.

The two analog inputs, process variable and setpoint, are treated as 0 to 100% signals inside the algorithm for mathematical purposes. Each input and output has a maximum and minimum scale parameter associated with it. When the User programs the PID, he sets the minimum engineering unit which will correspond to 0% and the maximum engineering unit that will correspond to 100% for that input. The PID algorithm will internally scale the inputs, generating a percentage of the scaling range before they are used. If the minimum and maximum scale parameters are left blank, the algorithm assumes that input is a percentage already scaled 0 to 100%.

The other three inputs are digital values and can be used to implement complex control strategies. They can be actuated by any active Step of a Task. They have been assigned names based upon the PID Loop with which they are associated. These names are Track_Mode, Block_Up and Block_Down. These inputs are controlled with the Set_Bit and Clear_Bit keywords.

If a PID Loop is named Tank_Level then the UPTime program statement:

Set_Bit Tank_Level Track_Mode.

puts PID loop Tank_Level into tracking mode.

The Track_Mode input when set, puts the PID loop into a track mode where the output is not changed by the PID, and the algorithm is set up to perform a bumpless transfer when the Track_Mode input goes back to being false. This signal would be used with a Manual / Automatic station to obtain Manual / Auto bumpless transfer.

The other two inputs, PID_Block_Up and PID_Block_Down limit the PID loop to integrating in only one direction and tracking in the other direction. For example, when Block_Up is true, the PID is allowed to integrate down, towards zero, when the error signal, the difference between setpoint and process variable, is negative. If the error signal is positive however, the output will not change and the integral portion will not be allowed to wind up. Block_Down works exactly opposite.

These two inputs can be used to provide Anti–Reset Windup when one PID is cascaded into another PID. If during the course of operation the down stream PID reaches the point it can no longer integrate in response to the upstream PID's output, as when it reaches full scale, then any further integrating of the upstream PID would be counter productive. These inputs can stop that excess integration in the counter productive direction while allowing immediate response in the opposite direction which is the direction that will have an affect on the down stream PID.

Using the Track, Block_Up and Block_Down signals allow the User to build very sophisticated controls. If all that is needed is a simple single loop controller, these inputs can be ignored. A more detailed discussion of these inputs and their use is in the **PID Algorithm Philosophy** Section.

PID Output

The PID outputs consist of a floating point value output that can either be assigned directly to an analog output, or to a floating point variable for use such as the input to another PID or in some calculation. Internal to the PID, the output's range is 0 to 100%. Like the inputs, the output also has a scaling constant that allows the User to set the maximum and minimum scale. This allows the User to make the output be a –100% to +100% controller, or any other engineering units desired. If the maximum and minimum scale constants are left blank, the PID output is 0 to 100%.

The other three outputs are digital variables that indicate the status of the PID algorithm. They are Track_Mode, HL_Status and LL_Status. The two limit outputs are true when either the PID has reached its limit or the Block_Up or Block_Down inputs are true.

The outputs may be used in the UPTime program by specifying the PID name followed by the appropriate keyword. For example, the UPTime program statement:

If Tank_Level LL_Status is true, go to the Reset Step.

makes Reset the active Step of this task.

Tuning Constants

The PID algorithm also has several adjustable tuning constants. These include the Gain, Reset and Rate, the high limit and low limit, and a bias value. The bias is a value that is added to the output at all times and can be used to insure a minimum output from the PID.

The high and low limits set maximum and minimum values, within the scale maximum and minimum, that the loop output will not exceed. When these limits are reached, the

appropriate status output is set, and anti–reset windup techniques automatically go into affect for that PID.

Every PID also can be selected to be either a direct acting or inverse acting loop. This is a parameter selected at programming time. A direct acting loop will integrate from 0 to 100% if the setpoint is greater than the process variable. A reverse acting loop will integrate from 0 to 100% when the process variable is greater than the setpoint. All other features are exactly the same whether the PID is in the direct or inverse mode.

The following is a summary of User selected values. These values are all entered in the PID Loop Configuration form displayed for initializing a PID loop. PID loops are initialized in program mode using the LIST and DEFINE options from the program mode menu.

Language Structure Summary

Notational Conventions

The following notational conventions are used throughout this section to rigorously define the required structure:

Table 8-9. Language Structure Notational Conventions

Underline	– Identifies Keywords
[ ]	– Encloses terms which are optional
{ }	– Encloses terms which may be repeated
< >	– Encloses a generic description of a term
|	– Indicates that either the term before or after may be used at this point.
( )	– Group Terms Together

Program Hierarchy

Term	Syntax
Program	{ <Sequence> }
Sequence	{ <Task> }
Task	Task: <Task Name> [StartIn_Last_Step]{ <Step> }
Step	Step: <Step Name> [Max Time <Numeric Constant>][{ <Statement>}]
Statement	([<Conditional Expression>] <Functional Expression>) (<Functional Expression> [<Conditional Expression>])

Functional Structures

Term	Syntax
Functional Expression	{ Functional Term }
Functional Term	Functional Term =< Control Sequences Term > | < Control Diagnostics Term > | < Turn On Discrete Term > | < Assign Values Term > | < Change Active Steps Term > | < Send Serial Information Term > | < PID Control Term > | < Change Serial Port Configuration Term > | < Execute Perform Functions Term > | **When_Done_Next_Step**
Turn On Discrete Term	**Actuate** { <Digital I/O Name> | < Internal Flag Name > }
Assign Values Term	< Make Term > | < Math–Assignment Term > | < Set_Bit / Clear_Bit Term >
Make Term	Make (< Numeric Assignment Term > | < Character Assignment Term > | < String Assignment Term>)
Numeric Assignment Term	(<Numeric Variable Name> | <Analog I/O Name>) **equal** < Numeric Value >
Character Assignment Term	<Character Variable Name> **equal** <Character Value>
String Assignment Term	<String Variable Name> **equal** <String Value>
Math–Assignment Term	< Add Term > | < Subtract Term > | < Multiply Term > | < Divide Term >
Add Term	Add (< Numeric Constant > | < Variable Name>) < Variable Name >
Subtract Term	Subtract (< Numeric Constant > | < Variable Name >) < Variable Name >
Multiply Term	Multiply < Variable Name > (< Numeric Constant > | < Variable Name >)
Divide Term	Divide < Variable Name > (< Numeric Constant > | < Variable Name >)
Set_Bit/Clear_Bit Term	**(Set_Bit | Clear_Bit)** (<Integer Variable Name> <Integer Number>)
Change Step Term	(**Go** <Step Name>) | (**Make** <Task Name> equal <Step Name>) | ((**Suspend_Task | Resume_Task**) <Task Name>

Functional Structures (continued)

Term	Syntax
Send Serial Data Term	Write" <Serial Data> " [<Port Name> | **R_Register** <Number>]
PID Loop Control Term	<Start PID Term> | <Stop PID Term>
Start PID Term	**Start_PID** <PID Loop Name>
Stop PID Term	**Stop_Pid** <PID Loop Name> [**with** <Numeric Constant>]
Port Configuration Term	**Set_Commport** <Port Name> <Parameter Value List>
Perform Function Term	**Perform** <Function Name> with <Parameter/Value List>
Control Sequences Term	(**Start_Sequence**<Sequence Name>) | (**Stop_Sequence** < Sequence Name >) | **End_Sequence**
Control Diagnostics Term	(**Enable_Diagnostic** <Diagnostic Name>) | (**Disable_Diagnostic** <Diagnostic_Name >)

Conditional Structures

Term	Syntax
Conditional Expression	< Test Conditional > | < Character Input Conditional >
Character Input	**Read** <Variable Name> [**from** <Communications Port Name> | (**R_Register** <Register Number> **for** Integer Number>)]
Test Conditional	**If** [**NOT**] <Conditional Term> [{ (**OR** | **AND**) [**NOT**] <Conditional Term> }]
Conditional Term	<Digital Test Conditional> | <Timer Test Conditional> | <Relational Test Conditional> | <Current Step Test Conditional>
Digital Test Conditional	(<Digital I/O Name> | <Flag Name> | <Device Name>) [{ (**AND** | **OR**) (<Digital I/O Name> | <Flag Name> | <Device Name>) }] (**ON** | **OFF**)
Timer Conditional	(<Numeric Constant>|<Integer Variable Name>) seconds
Current Step Conditional	<Task Name> (equal | not equal) <Step Name>
Relational Test Conditional	<Numeric Relational Term> | <Character Relational Term> | <String Relational Term>
Numeric Relational Term	<Numeric Value> <Relational Operator> <Numeric Value>
Character Relational Term	<Character Value> (**equal** | **notequal**) <Character Value>
String Relational Term	<String Value> (**equal** | **notequal**) <String Value >

Value Expressions

Term	Syntax
Numeric Value	<Numeric Constant> | <Calculation> | <Numeric Variable Name> | <Analog I/O Name> | <PID Value>
Calculation	(< Numeric Value > < Operator > < Numeric Value >) | < System Functions > (< Numeric Value >)
System Functions	**SIN | COS | TAN | ARCTAN | SQRT | EXP | LN | RANDOM**
Numeric Constant	<Floating Point Number> | <Integer Number>
PID Value	<PID Loop Name> <PID Parameter Keyword>
Character Value	< Character Variable Name > | ‘ <Character> ’
String Value	<String Variable Name > | “ < Character String > ” – Up to 80 characters

Appendix A Specifications

This sections has specifications for the Diagnostic Processor Module, and the UPTime software. In addition are lists of standards met by the Diagnostic System and the I/O and register capabilities of the Diagnostic System.

Table A-1. UPTime Specifications

Sequences	254
Tasks	254
Steps Per Task	255
Maximum Number of Variables	3000
Maximum Number of Steps	3000
Statements per Step	Unlimited
Devices	100
Integer Variables (range –32768 to +32767)	1000
Floating Point Variables (range +/– 1.175494E–38 to +/– 3.402823E+38)	1000
String Variables	100
String Variable Size	80 Characters
Character Variables	64
PID Loops	10
Internal Flags	80
Number of Timers	Unlimited
Timer Resolution	1/100second
Number of Characters per Write Term	512
Step Changes Listed in Trace Display	80
Histogram Size	255 Entries
Force Table Size	32
Monitor Table Size	6 Entries
Monitor Tables	10

Table A-2. Diagnostic Processor Specifications

Serial Ports	2RS–232 / 422 / 485
Memory Backup Battery	3 Volt Lithium
Battery Shelf Life	10 years
Battery Memory Retention with Power OFF	6 months nominal
Operating Temperature	0 to 60ºC
Storage Temperature	–40 to 85ºC
Humidity (non–condensing)	5–95%
Vibration	3.5 mm, 5–9 Hz:
	1.0 G 9–150 Hz
Shock	15 G's 11 msec

Table A-3. Meets or Exceeds these Standards

IEC	485, 380
JIS	C 0912, JIS C 0911
DIN	435, 380
UL	508, 1012
CSA	C22.2 No. 142,
C22.2	
NEMA / ICS	2–230.40
ANSI / IEE	C–37.90A–1978
VDE	805, 806, 871–877
FCC	15J Part A
VME	Supports VME Standard C.1

Table A-4. 90-30 Diagnostic Processor I/O and Register Specifications

Type	Capacity
%I	512
%Q	512
%AI	128
%AQ	64
%T	256
%M	1024
%G	1280
%S	32
%SA	32
%SB	32
%SC	32
%R	2048

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W