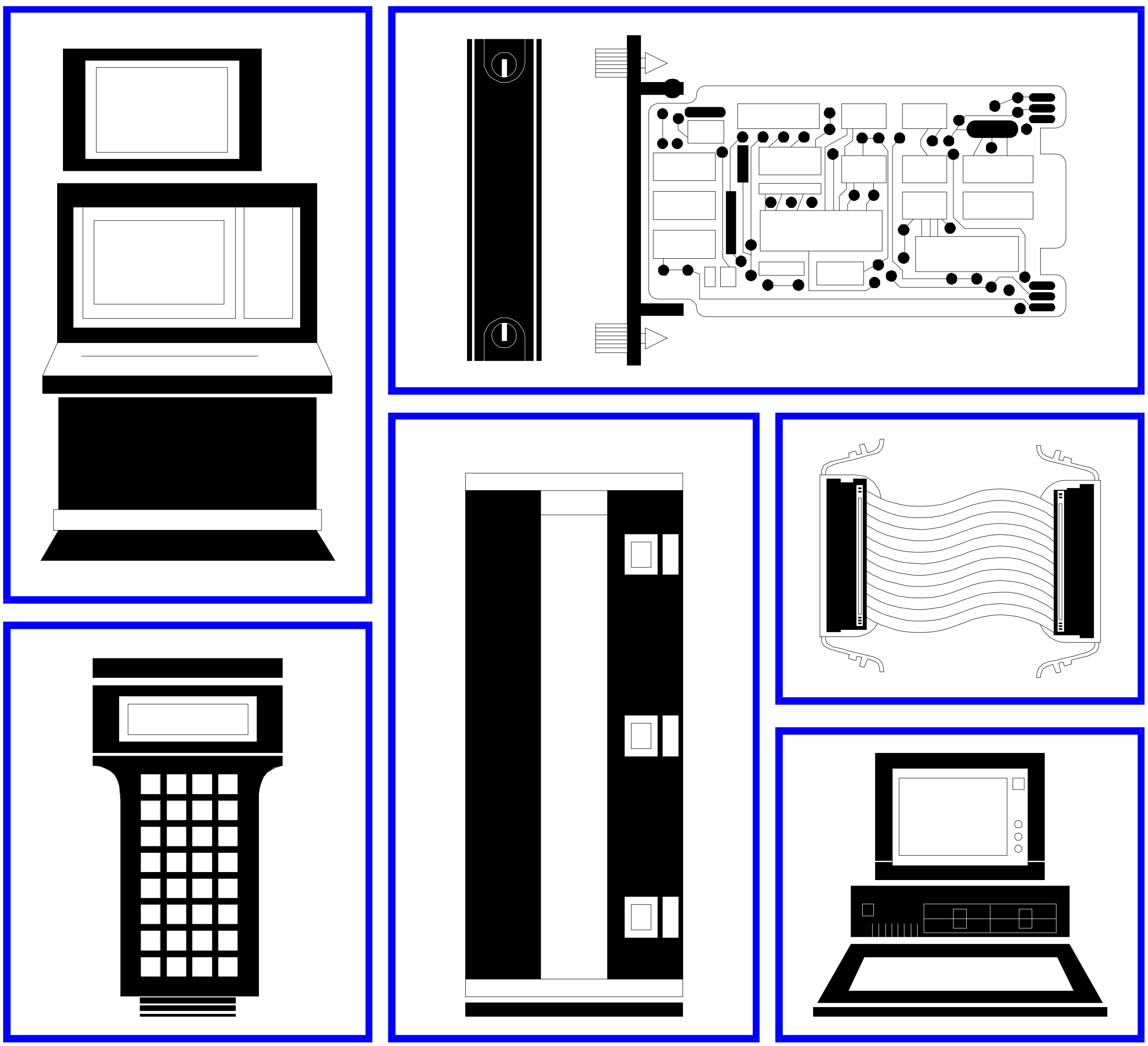
B90/UDF

Instruction

Batch 90 and User Defined Function (UDF) Programming Language Reference
(Software Release 4.0)



Bailey® Process Control and Automation Solutions from Elsag Bailey Group

WARNING notices as used in this instruction apply to hazards or unsafe practices that could result in personal injury or death.

CAUTION notices apply to hazards or unsafe practices that could result in property damage.

NOTES highlight procedures and contain information that assists the operator in understanding the information contained in this instruction.

WARNING

INSTRUCTION MANUALS

DO NOT INSTALL, MAINTAIN, OR OPERATE THIS EQUIPMENT WITHOUT READING, UNDERSTANDING, AND FOLLOWING THE PROPER **Elsag Bailey** INSTRUCTIONS AND MANUALS; OTHERWISE, INJURY OR DAMAGE MAY RESULT.

RADIO FREQUENCY INTERFERENCE

MOST ELECTRONIC EQUIPMENT IS INFLUENCED BY RADIO FREQUENCY INTERFERENCE (RFI). CAUTION SHOULD BE EXERCISED WITH REGARD TO THE USE OF PORTABLE COMMUNICATIONS EQUIPMENT IN THE AREA AROUND SUCH EQUIPMENT. PRUDENT PRACTICE DICTATES THAT SIGNS SHOULD BE POSTED IN THE VICINITY OF THE EQUIPMENT CAUTIONING AGAINST THE USE OF PORTABLE COMMUNICATIONS EQUIPMENT.

POSSIBLE PROCESS UPSETS

MAINTENANCE MUST BE PERFORMED ONLY BY QUALIFIED PERSONNEL AND ONLY AFTER SECURING EQUIPMENT CONTROLLED BY THIS PRODUCT. ADJUSTING OR REMOVING THIS PRODUCT WHILE IT IS IN THE SYSTEM MAY UPSET THE PROCESS BEING CONTROLLED. SOME PROCESS UPSETS MAY CAUSE INJURY OR DAMAGE.

AVERTISSEMENT

MANUELS D’OPÉRATION

NE PAS METTRE EN PLACE, RÉPARER OU FAIRE FONCTIONNER L’ÉQUIPEMENT SANS AVOIR LU, COMPRIS ET SUIVI LES INSTRUCTIONS RÉGLEMENTAIRES DE **Elsag Bailey**. TOUTE NÉGLIGENCE À CET ÉGARD POURRAIT ÊTRE UNE CAUSE D’ACCIDENT OU DE DÉFAILLANCE DU MATÉRIEL.

PERTURBATIONS PAR FRÉQUENCE RADIO

LA PLUPART DES ÉQUIPEMENTS ÉLECTRONIQUES SONT SENSIBLES AUX PERTURBATIONS PAR FRÉQUENCE RADIO. DES PRÉCAUTIONS DEVRONT ÊTRE PRISES LORS DE L’UTILISATION DU MATÉRIEL DE COMMUNICATION PORTATIF. LA PRUDENCE EXIGE QUE LES PRÉCAUTIONS À PRENDRE DANS CE CAS SOIENT SIGNALÉES AUX ENDROITS VOULUS DANS VOTRE USINE.

PERTURBATIONS DU PROCÉDÉ

L’ENTRETIEN DOIT ÊTRE ASSURÉ PAR UNE PERSONNE QUALIFIÉE EN CONSIDÉRANT L’ASPECT SÉCURITAIRE DES ÉQUIPEMENTS CONTRÔLÉS PAR CE PRODUIT. L’AJUSTEMENT ET/OU L’EXTRACTION DE CE PRODUIT PEUT OCCASIONNER DES À-COUPS AU PROCÉDÉ CONTRÔLE LORSQU’IL EST INSÉRÉ DANS UNE SYSTÈME ACTIF. CES À-COUPS PEUVENT ÉGALEMENT OCCASIONNER DES BLESSURES OU DES DOMMAGES MATÉREILS.

NOTICE

The information contained in this document is subject to change without notice.

Preface

The Batch 90™ programming language provides a method of configuring batch sequence strategies. The User Defined Function (UDF) programming language provides a method of creating a custom control function. This instruction explains the Batch 90 and UDF programming languages.

NOTE: The Batch 90 terms used in this instruction comply with the ISA-S88.01-1995 standard. Elsag Bailey specific terms used in previous versions are listed in Section 4 of this instruction for those who wish to continue to use these terms.

™ Trademark of Elsag Bailey Process Automation.

List of Effective Pages

Total number of pages in this instruction is 322, consisting of the following:

Page No.	**Change Date**
Preface	Original
List of Effective Pages	Original
iii through viii	Original
1-1 through 1-4	Original
2-1 through 2-20	Original
3-1 through 3-12	Original
4-1 through 4-251	Original
A-1 through A-15	Original
B-1 through B-7	Original
Index-1 through Index-5	Original

When an update is received, insert the latest changed pages and dispose of the superseded pages.

NOTE: On an update page, the changed text or table is indicated by a vertical bar in the outer margin of the page adjacent to the changed area. A changed figure is indicated by a vertical bar in the outer margin next to the figure caption. The date the update was prepared will appear beside the page number.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

List of Figures

List of Figures (continued)

List of Tables

SECTION 1 - INTRODUCTION

OVERVIEW

Batch 90 and User Defined Function (UDF) are programming languages whose purpose is to create sequence control logic using descriptive, readable, and understandable statements. The Batch 90 programming language is specifically designed to implement flexible sequences in which the order of operations and parameter data are specified by recipes. Similarly, the UDF programming language is designed to implement fixed sequences for which the order of operations is less flexible.

ENHANCEMENTS

Version 4.0 of Batch 90 and User Defined Function (UDF) offers the following enhancements over previous versions:

- The interfaces for all tools associated with the Batch 90 and UDF programming languages have been enhanced to follow the style of the Microsoft® Windows® 95 operating system. This provides a graphical user interface that is easier to understand and use.

- Both Batch 90 and UDF programming language compilers have been enhanced to include a text editor that is fully integrated with all other batch tools.

- Support for constant and variable string arrays that allows for more streamlined coding of complex string manipulations.

Additional enhancements to Batch 90 include:

- The ability to specify alphanumeric campaign, batch, lot, and recipe identifiers allows for more meaningful naming conventions.

- Support for command and status string variables in the common sequence data structures enabling the passing of alphanumeric data between common sequence applications.

- The ability of phases (formerly step subroutines) to extend beyond operation (formerly phase) boundaries.

Information regarding features added to this product as the document was being printed is available in a ***README.B90*** or ***README.UDF*** file located in the directory specified during the

® Registered trademark of Microsoft Corporation.

installation process. This is an ASCII document file and can be routed to a printer via Windows or DOS commands. Print this file and append it to this document.

INTENDED USER

This instruction provides the necessary information to utilize the Batch 90 and UDF programming languages. It is intended for programmers with batch process or sequential logic control experience and a working knowledge of function codes and the Elsag Bailey CAD/EWS software. The programmer should also be familiar with the configuration of operator interface stations.

INSTRUCTION CONTENT

This instruction contains four sections and two appendices. It also includes a Table of Contents, List of Figures, List of Tables, and Index giving several options to locate specific information quickly. Appendices supplement information presented in the individual sections. The sections that make up this instruction include:

Introduction Provides an overview of the Batch 90 and UDF programming languages and this instruction.

Batch 90 Description Describes the Batch 90 programming language.

User Defined Function (UDF) Description Describes the User Defined Function (UDF) programming language.

Function Library Provides detailed information about and an example of each command and statement. Also explained are the rules, reserved words, and logic operators used in commands and statements.

HOW TO USE THIS INSTRUCTION

Read this instruction through in sequence before attempting to program using the Batch 90 or UDF programming languages. It is important to become familiar with the entire contents of the instruction prior to creating any programs to insure maximum use of all programming language capabilities.

This instruction limits the information presented in each section to only specific items required to complete the desired task. The organization enables finding specific information quickly, and permits using this instruction as a reference after becoming fully familiar with the programming languages.

Be sure to read notes in text. Notes provide:

- Additional information.

- Information that should be considered before using a certain command or statement.

DOCUMENT CONVENTIONS

This document uses standard text conventions throughout to represent keys, user data inputs and display items:

KEY	Identifies a keyboard key.
Example:	Press ENTER.
Display item	Any item that displays on the screen appears as italic text in this document.
Examples:	*F7 - DELETE UDF* (menu selection)
	Invalid definition (message)
	Select function (prompt)
File name	Any file names and file extensions appear as bold-italic text.
Example:	***PROCESS.B90***

The document uses a specific set of text conventions for user inputs:

BOLD	Identifies any user input or part of a command line that is ***not*** optional or variable, and must be entered exactly as shown.
italic	Identifies a variable parameter in user input or a command line.
[ ]	The brackets indicate a parameter is optional. Text within the brackets still follows the previously described conventions. The brackets only designate optional information and are not actually entered.
Example:	**NEXT OPERATION** [*exp*]

GLOSSARY OF TERMS AND ABBREVIATIONS

Table 1-1 is a glossary of terms and abbreviations used in this instruction. It contains those terms and abbreviations that are unique to Elsag Bailey or have a definition that is different from standard industry usage.

Table 1-1. Glossary of Terms and Abbreviations

Term	Definition
CAD	Computer aided design. A Bailey computerized drawing package that allows an engineer/technician to create function block control logic.
INFI-NET®	Advanced data communication highway.
MFP	Multi-function processor module. A multiple-loop controller with data acquisition and information processing capabilities.
OIS	Operator interface station. Integrated operator console with data acquisition and reporting capabilities. It provides a digital access into the process for flexible control and monitoring.
Plant Loop	Network 90® data communication highway.

REFERENCE DOCUMENTS

This instruction provides information about the Batch 90 and UDF programming languages only. Table 1-2 lists additional documents that relate to associated hardware and software.

Table 1-2. Reference Documents

Number	Document
I-E96-200	Function Code Application Manual
I-E96-201	Multi-Function Processor Module (IMMFP01)
I-E96-202	Multi-Function Processor Module (IMMFP02)
I-E96-203	Multi-Function Processor Module (IMMFP03)
I-E96-770	Module Configuration Tools (Release 1.0)
WBPEEUI330251A0	Batch Data Manager (Release 4.0)

® Registered trademark of Elsag Bailey Process Automation.

SECTION 2 - BATCH 90 DESCRIPTION

INTRODUCTION

This section describes unit recipes, batch programs, operations, and subroutines. This section also defines what they do, and how they interact with each other.

UNIT PROCEDURES

A unit procedure is a plan for executing a batch sequence. A unit recipe is organized as a sequence of operations. An alphanumeric descriptor identifies each unit procedure. The unit procedure specifies the following information for each operation:

1. The names of the phase subroutines that the operation will execute. An operation can specify any phase subroutine contained in the batch program. For example, operation five of a particular unit procedure may call the phase subroutine named BRINE_ON_JACKET. Operation five of another unit procedure may call the phase subroutine named ADD_B_OR_C. The operation numbers specify the normal execution order of the operations. The compiler directive #MAX-PARALLEL limits the number of phase subroutines that can be assigned to any given operation.

2. The formulation data for each operation. The selected phase subroutines specify the parameter lists. The formulation data is passed from the unit procedure to the associated phase subroutine in accordance with the respective parameter lists. For example, operation five of a unit procedure calls phase subroutine ADD_B_OR_C. The formulation data specifies the quantity of component B to be added to the reactor and a default time during which the addition must be completed. Therefore, operation five of this unit procedure has two pieces of formulation data associated with it. This data passes to the phase subroutine ADD_B_OR_C through the parameter list.

The formulation data associated with the operation does not have to be numerical data exclusively. For example, it is possible to pass device drivers, function block identities, etc. This is a very powerful feature. It allows generic phase subroutines to be written and the passing of actual devices and sensors to be used from the unit procedure to the batch program logic.

Example Unit Procedure

Figure 2-1 shows a diagram of a simple batch process. This process has several operations. Figure 2-2 is a listing of these operations. A phase subroutine performs each operation.

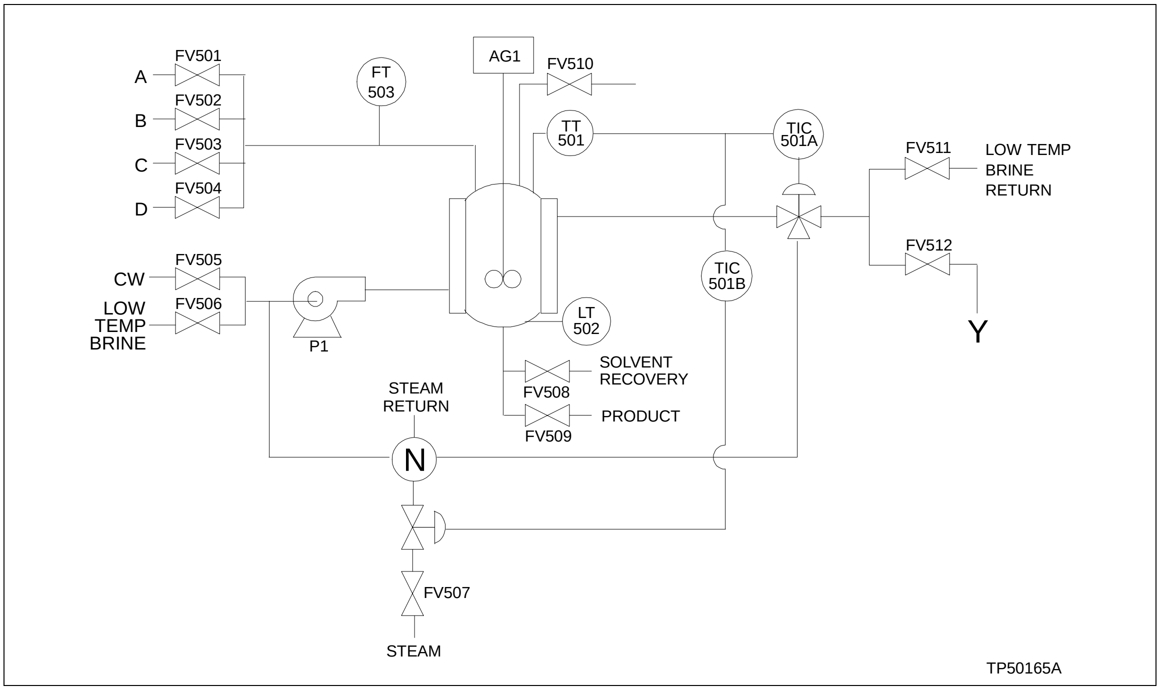


Figure 2-1. Example Batch Process

EXECUTED STOP
  CLEANING REACTOR
  ADD COMPONENT B TO REACTOR
  ADD COMPONENT C TO REACTOR
  ADD COMPONENT D TO REACTOR
  RAMP TEMPERATURE OF REACTOR
  PUT JACKET ON LOW TEMP BRINE
DUMP PRODUCT

Figure 2-2. Batch Sequence Listing of Operations

A unit procedure specifies the operation execution order which specifies the order that phase subroutines are executed and the data that is passed. Figure 2-3 is a listing of a typical unit procedure for this process.

Executed stop is always operation zero of every unit procedure. When the executed stop input of the batch sequence function block goes to a one, the batch sequence block executes operation zero. Therefore, any specific logic that is to be performed on an executed stop must be configured within a phase subroutine that is in operation zero of the unit procedure. Numbering of operations is from zero to 250. Operation zero must be present or the unit procedure editor will renumber the operations so that the first operation is operation zero.

Create unit procedures using the unit procedure editor. Establish LO and HI values in the unit procedure using the editor.

OPERATION	PHASE SUBR	LO		VALUES		HI
0	EXECUTED_STOP					
1	CLEANING_REACTOR					
2	ADD_B_OR_C					
	DEFAULT_TIME	0	<	900	<	1000
	TOTAL_FLOW_B	0	<	400	<	500
3	BRINE_ON_JACKET					
	REACTOR_TEMP	0	<	30	<	50
4	RAMP_TEMPERATURE					
	RATE1			0	<	1.5
	DEFAULT_TIME	0	<	900	<	2000
	INITIAL_TEMP	-50	<	-40	<	10
	FINAL_TEMP	0	<	30	<	60
5	ADD_B_OR_C					
	DEFAULT_TIME	0	<	1200	<	1500
	TOTAL_FLOW_C	0	<	600	<	900
6	ADD_D					
	COOK_TIME	0	<	1500	<	2000
	DEFAULT_TIME	0	<	2000	<	2200
	TOTAL_FLOW_D	0	<	500	<	600
	REACTOR_TEMP	0	<	50	<	100
7	DUMP_PRODUCT					

Figure 2-3. Example Unit Procedure

These values place input boundaries on formulation data. If the values in the formulation data are not within the LO/HI values, the editor generates a compilation error that must be corrected before the unit recipe can be downloaded to an MFP module for execution.

BATCH PROGRAMS

A batch program consists of data declaration sections (BATCH or UNIT DATA) and a set of subroutines. There are three types of subroutines: phase, function, and monitor. Each batch sequence function block will have one batch program file associated with it. Several different batch programs, each associated with a different batch sequence function block, can reside in a single MFP module.

Use the editor within the Batch Data Manager to write Batch 90 programs. If necessary, any word processor or text editor that outputs an ASCII formatted file can be used to create batch files. The text file is the source file. The compiler processes the program file and checks the syntax before generating the executable object file.

This is a general overview of these program elements (refer to Section 4 for detailed explanations).

Batch Data Types

Batch data defines data that is available to all subroutines of a given batch program. Examples of information of global importance are control loops, devices (pumps, valves, etc.), and function blocks used to solve calculation or interlock problems. The advantage of batch data declaration is that the data is permanent, defined only once, and available to all subroutines.

There are four types of batch data:

- Function blocks.
- Active data structures (integrators, timers, ramps, etc.).
- Variables and constants.
- Monitor subroutine declarations.

Phase Subroutine

The phase subroutine is the major part of a batch program. It is responsible for controlling the sequence of activities within an operation or across multiple operations. The number of operations a multi-operation phase (superphase) subroutine will operate through is specified in the unit procedure editor.

Within a unit procedure, each operation specifies the names of phase subroutines to be executed for that operation. Each phase subroutine contains a parameter list for which values are specified by the unit procedure. Phase subroutine execution may extend beyond operation boundaries. Such multi-operation phases (superphase) are specified within the unit recipe editor. When the single operation phase subroutine completes its activities, control passes to the next operation. The same phase subroutine may be called by different operations, or called multiple times by an operation. The multi-operation phase subroutine continues to execute across operation boundaries until the phase subroutine completes its activities. Figure 2-4 shows the structure of a phase subroutine.

Heading Section

This section defines the phase subroutine name, parameter list, and optional phase comment. The name allows the unit procedure to address a phase subroutine by a meaningful title such as ADD_A. The parameter list defines the parameter names, data types, (optional) default values, and allowable selections for the formulation data passed from the unit recipe.

Declarations Section

The declarations section defines the data structures that are local to the phase subroutine and calls made to local monitor subroutines. Local data typically is timers, integrators, ramps, variables, etc. Local data can only be used by the phase subroutine that declared it. Also, this data is temporary; it is only

```
PHASE SUBR name    (HEADING section)
    [Phase comment]    (HEADING section)
    Parameter list    (HEADING section)
DECLARATIONS
    Local data and monitor subroutine calls
CONTINUOUS
    Executable statement
NORMAL LOGIC
    Executable statements
FAULT LOGIC
    Executable statements
HOLD LOGIC
    Executable statements
RESTART LOGIC
    Executable statements
ENDSUBR  [name]
```

Figure 2-4. Phase Subroutine Structure

valid while the phase subroutine is active. Local data divides into three types:

- Variables.
- Constants.
- Active Data Structures.

Phase subroutines have access to all global data (including function blocks) declared in any batch or unit data section of the batch program that comes before the phase subroutine. No additional declaration is necessary or allowed inside the phase subroutine.

Continuous Section

The entire continuous section executes every function block cycle in parallel with the sequential logic (NORMAL, HOLD, FAULT, RESTART). It is ideal for defining interlock logic that must be performed continuously and is specific to the phase subroutine. The continuous section prohibits commands that can result in a wait condition. Examples of prohibited commands are REPEAT, WAIT, WAIT UNTIL, etc.

Normal Logic

Normal logic is an executable section that performs the normal, sequential logic of a recipe phase within an operation of a unit procedure. It begins when the phase subroutine starts. It remains in operation until:

1. The last statement of the normal logic executes. The phase subroutine has ended and control passes to the next phase subroutine called for by the unit procedure. This is true for all operations except for the special case of operation number one (executed stop).

2. The mode of the phase subroutine changes from NORMAL to FAULT or HOLD. The mode typically changes to FAULT by some interlock failure detected by the continuous section or a

monitor subroutine. Typically, an operator action is required to place a phase subroutine into HOLD.

3. The execution of a NEXT OPERATION statement causes the current phase subroutine to become inactive. Also, the next phase subroutine called for by the unit procedure becomes active.

4. The execution of a DONE statement forces the program execution to batch complete state.

Fault Logic

Fault logic is the section that executes user defined statements when a fault condition occurs. When a fault occurs in the normal logic, continuous logic or a monitor subroutine, execution of normal logic suspends and control passes to the fault logic. In the fault logic define the executable commands necessary to hold or stop the batch safely while solving the problem. Once the fault logic completes, the sequence goes into holding state.

Hold Logic

The purpose of hold logic is to allow execution of safety shutdowns when the batch sequence is placed into hold. Control transfers from normal logic to this section when the Run/Hold input to the batch sequence block toggles to hold.

Restart Logic

Restart logic performs a restart of a batch sequence placed into the holding state either through a fault or hold command. Restart logic executes before normal logic and its purpose is to return the batch unit to a state where normal logic can continue operation.

NOTE: Refer to Appendix B for phase subroutine execution details.

Function Subroutine

The function subroutine allows the definition of commonly used logic. This subroutine is then used as often as needed. This aids in reducing the program size. A function subroutine may be called from a phase, monitor, or other function subroutine. A function subroutine takes control from the caller, executes to completion and then returns control to the caller. Figure 2-5 shows the function subroutine structure.

```
FUNCTION name    (HEADING section)
      Parameter list    (HEADING section)
DECLARATIONS
      Local data
EXECUTABLE
      Executable statements
ENDSUBR  [name]
```

Figure 2-5. Function Subroutine Structure

Heading Section This section defines the function name and parameter list.

Declarations Section Local data refers to data used solely within the subroutine. This data is valid only while the subroutine is active. Only variables and constants can be declared local to a function subroutine.

Executable Section Execution starts with the first statement of the executable section and continues until a RETURN or ENDSUBR statement executes. Then, control transfers back to the caller.

Monitor Subroutine

Monitor subroutines execute continuous logic that the user can stop and start when desired. Once defined, monitor subroutines can be called by multiple phase subroutines. Executed continuously at the function block cycle, monitor subroutines operate in parallel with normal logic according to the state transition diagram in Appendix B. Refer to Figure 2-6 for the monitor subroutine structure.

```
MONITOR name    (HEADING section)
      Parameter List    (HEADING section)
DECLARATIONS
      Local Data
CONTINUOUS
      Executable Statements
ENDSUBR  [name]
```

Figure 2-6. Monitor Subroutine Structure

The monitor subroutine components are very similar to those of the phase subroutine. Monitor subroutines can be declared either locally (in phase subroutines) or in a batch data section. When declared locally they can be referenced only in the active phase subroutine that declared the monitor subroutine.

When declared in a batch data section, the monitor subroutine can be referenced by any phase subroutine that follows the declaration. Once started, a monitor subroutine will continue to execute until stopped or until the batch is complete. It will even run if the sequence is in holding.

NOTE: A monitor subroutine cannot start itself.

Heading Section This section defines the monitor subroutine name and parameter list. The list defines the local parameter aliases (names used solely in the monitor subroutine).

Declarations Section Local data refers to data used solely within the subroutine. This data is valid only while the subroutine is active. Only variables and constants can be declared local to a monitor subroutine.

Continuous Section

The monitor subroutine has only a continuous logic section. Continuous logic executes every function block cycle. A monitor subroutine may not contain any statements that can result in a wait condition (for example, REPEAT, WAIT UNTIL, etc.).

BATCH PROGRAM CONSTRUCTION

This section illustrates how to construct a batch program through an example. This example will show how to:

- Create batch data section.
- Create monitor subroutines.
- Create function subroutines.
- Create phase subroutines.

The batch process has several devices, control loops and alarms associated with it. These processes must be defined as function blocks (normally with the engineering work station). Then, add the function blocks to the program with batch declaration statements. The example declares two constants and 27 function blocks (with associated MFP addresses).

The block command identifies a configurable name that is assigned to a function block (for example, FV501) and function block type (for example, FC = DD meaning device driver).

Any function code can be declared and its value read by the batch program. For certain types of function codes (device drivers, multi-state device drivers, control stations, remote control memories and remote manual set constants, etc.) additional attributes are available. Reference extensions identify the attributes. Reference extensions give the batch language the ability to read or command attributes of these function blocks. For example, with a control station, the command CSNAME.SP allows the batch language to reference or change the set point of the declared control station called CSNAME. A detailed explanation of the reference extensions is in Section 4 of this document. Refer to Figure 2-7 for a batch data example.

It is possible to declare function blocks in the batch data section that do not exist in the module. Similarly, it is possible to declare function blocks incorrectly (for example, assign wrong function code numbers). The Batch 90 compiler does not check for these types of errors. Before the batch sequence function block goes into run mode, it will check to make sure that all function block declarations are correct. If not, the batch sequence block generates a fault (fault code -16).

Figure 2-8 is an example of how to configure device drivers and control loops using an engineering work station.

```
BATCH DATA
    CONST EXPIRED = 1
    CONST NOT_OPEN = 0
    BLOCK FV501, BLK = 100, FC = DD
    BLOCK FV502, BLK = 102, FC = DD
    BLOCK FV503, BLK = 104, FC = DD
    BLOCK FV504, BLK = 106, FC = DD
    BLOCK FV505, BLK = 108, FC = DD
    BLOCK FV506, BLK = 110, FC = DD
    BLOCK FV507, BLK = 112, FC = DD
    BLOCK FV508, BLK = 114, FC = DD
    BLOCK FV509, BLK = 116, FC = DD
    BLOCK FV510, BLK = 118, FC = DD
    BLOCK FV511, BLK = 120, FC = DD
    BLOCK FV512, BLK = 122, FC = DD
    BLOCK FC503, BLK = 130, FC = DD
    BLOCK TC501A, BLK = 142, FC = CS
    BLOCK TC501B, BLK = 150, FC = CS
    BLOCK CIS, BLK = 200, FC = 79, OUT (0) = LT502, OUT (1) = FT503
    BLOCK P1, BLK = 210, FC = DD
    BLOCK AG1, BLK = 212, FC = DD
    BLOCK START PB, BLK = 300, FC = RCM
    BLOCK PHASE NUMBER, BLK = 320, FC = REMSET
    BLOCK BUFFER, BLK = 330, FC = BBUF, OUT (0) = B00, OUT (1) = B01, OUT (2) = B02, \
    OUT (3) = B03
    BLOCK TEXT_LINE_1, BLK = 500, FC = TEXT
    BLOCK TEXT_LINE_2, BLK = 510, FC = TEXT
    BLOCK TEXT_LINE_3, BLK = 520, FC = TEXT
    BLOCK TEXT_LINE_4, BLK = 530, FC = TEXT
    BLOCK TEXT_LINE_5, BLK = 100, FC = TEXT
    BLOCK OPER_MSG, BLK = 600, FC = DATAEXPT
END DATA
```

Figure 2-7. Example Batch Data Section

Monitor Subroutine Definitions

The batch process example has several levels of device interlocking. They are:

- Basic device interlocking.
- Reactor protection interlocking.

BASIC DEVICE INTERLOCKING

The control system must monitor all the valves associated with inlet reactant feed valves (FV501, FV502, FV503, and FV504) and the discharge valves (FV508 and FV509). These devices must be interlocked constantly and, in case of failure, a safety shutdown must be executed. Figure 2-9 is the configuration for this interlock. The purpose of the FAULT statement is to suspend the normal logic of the phase subroutine and start the fault logic. The numerical value after the FAULT statement allows the passing of a variable to the fault logic. This allows the identification of the specific fault. Although the figure shows a different fault number for each fault statement, it is not required that each fault number be unique.

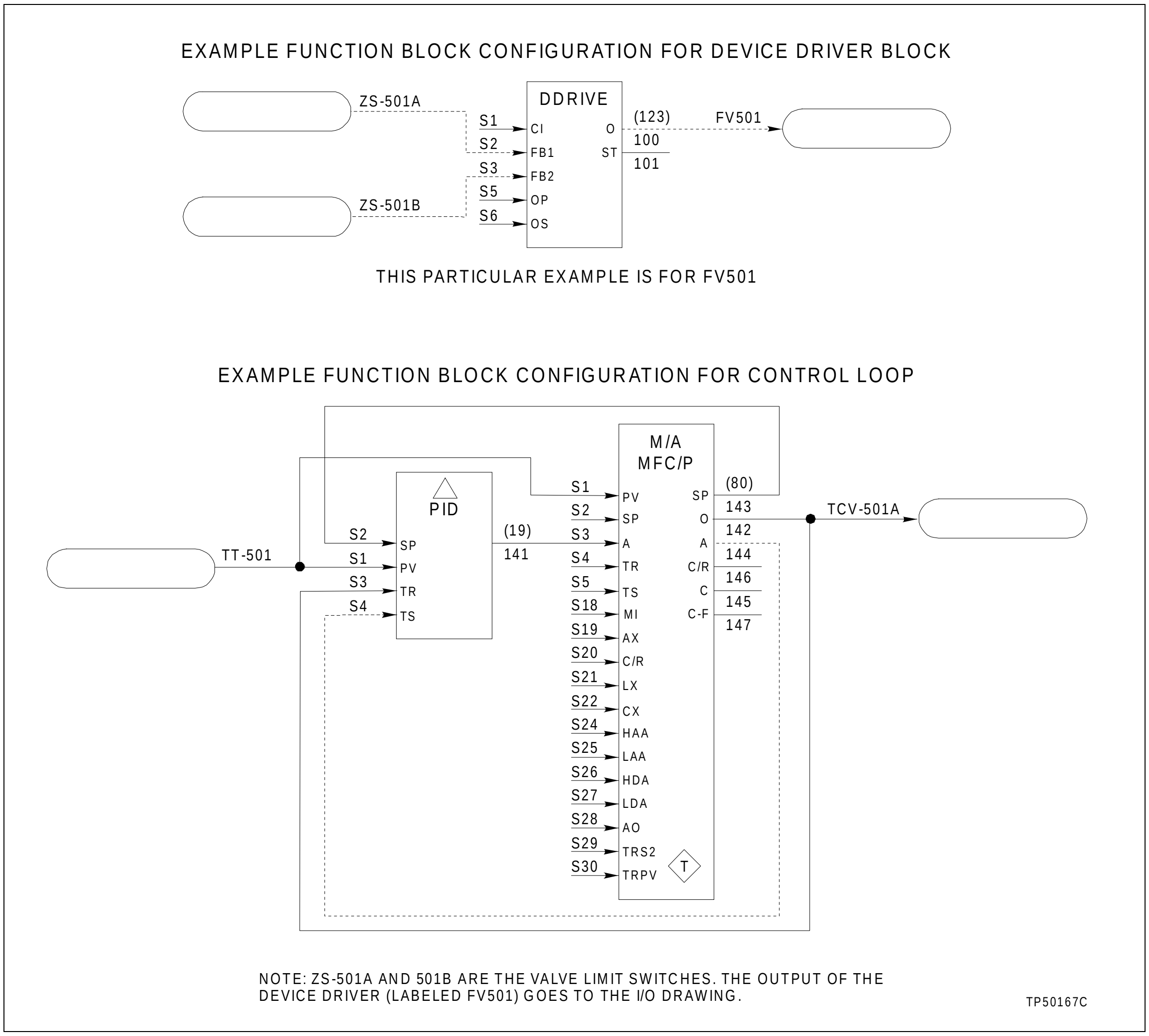


Figure 2-8. Example Function Block Configuration

REACTOR PROTECTION INTERLOCK

During the addition of component D, there is an exothermic reaction. Due to the possibility of a runaway reactor during the

```
MONITOR  BASIC_INTERLOCKING
     CONTINUOUS
          IF  (FV501.STS  =  BAD)  FAULT  1
          IF  (FV502.STS  =  BAD)  FAULT  2
          IF  (FV503.STS  =  BAD)  FAULT  3
          IF  (FV504.STS  =  BAD)  FAULT  4
          IF  (FV510.STS  =  BAD)  FAULT  5
          IF  (FV508.STS  =  BAD)  FAULT  6
          IF  (FV509.STS  =  BAD)  FAULT  7
ENDSUBR  BASIC_INTERLOCKING
```

Figure 2-9. Example Monitor Subroutine for Basic Device Interlocking addition of component D, the cooling system must be monitored to assure that all components are operating correctly. In addition, the temperature is also monitored. If either the cooling system malfunctions or the temperature rises above a certain level, then an emergency shutdown must be executed. The cooling system consists of transmitter TT501, pump P1, and valves FV505, FV506, FV511, and FV512. This monitor activity is only valid during those phase subroutines in which component D is added. Figure 2-10 shows the programming for the monitor subroutine providing reactor protection interlocking.

```
MONITOR  REACTOR_PROTECTION  VAR  TEMPERATURE
     DECLARATIONS
          VAR  X
     CONTINUOUS
          IF  (FV505.STS  =  BAD)  FAULT  8
          IF  (FV506.STS  =  BAD)  FAULT  9
          IF  (FV511.STS  =  BAD)  FAULT  10
          IF  (FV512.STS  =  BAD)  FAULT  11
          IF  (P1.STS  =  BAD)  FAULT  12
          X  =  TEMPERATURE  +  5
          IF  (X  >  TC501A.PV)  FAULT  13
ENDSUBR  REACTOR_PROTECTION
```

Figure 2-10. Example Monitor Subroutine for Reactor Protection Interlocking

Function Subroutine Definitions

The devices in the batch sequence example can be either in manual or automatic mode. In manual mode, the operator has taken control. In automatic mode, the batch logic can control the devices. It is often necessary to put the devices into automatic. It is best to create a function subroutine to do this. Refer to Figure 2-11 for an example.

Phase Subroutine Definitions

This process has several processes that need to be defined. They are:

- Cleaning the reactor.
- Adding component B.
- Adding component C.
- Adding component D.
- Ramping the reactor temperature.
- Filling reactor jacket with low temperature brine.
- Dumping the product.

```
FUNCTION  INITIALIZE_VALVES
    EXECUTABLE
        SET  FV501.MODE  =  AUTO
        SET  FV502.MODE  =  AUTO
        SET  FV503.MODE  =  AUTO
        SET  FV504.MODE  =  AUTO
        SET  FV505.MODE  =  AUTO
        SET  FV506.MODE  =  AUTO
        SET  FV507.MODE  =  AUTO
        SET  FV508.MODE  =  AUTO
        SET  FV509.MODE  =  AUTO
        SET  FV510.MODE  =  AUTO
        SET  FV511.MODE  =  AUTO
        SET  FV512.MODE  =  AUTO
        SET  FV501.MODE  =  AUTO
        SET  P1.MODE  =  AUTO
        SET  AG1.MODE  =  AUTO
ENDSUBR  INITIALIZE_VALVES
```

Figure 2-11. Example Device Mode Change Function Subroutine

CLEANING_REACTOR PHASE

This phase subroutine opens the feed valve to component A (cleaning solvent), starts the agitator and waits for five minutes. When the five minutes are over, the feed valve shuts and the dump valve to solvent recovery (valve FV508) opens. A ten minute allowance provides time to empty the reactor.

The only interlocking needed is the basic device interlocking performed by the monitor subroutine BASIC_INTERLOCKING. In case of failure, all valves shut and the program informs the operator. There is no need to have restart logic for this phase subroutine. Refer to Figure 2-9 for monitor subroutine details.

Any text between an open and closed bracket (characters { }) is strictly a comment. The batch compiler includes comments in its reference listing, but they are ignored for all other purposes. Refer to Figure 2-12 for an example of a phase subroutine for cleaning the reactor.

ADD_B_OR_C PHASE SUBROUTINE

For the example batch process shown in Figure 2-1, it can be concluded that the phase logic required to add components B and C is identical. The only change is the inlet valve to be operated. Only one phase subroutine needs to be written. Which inlet valve to be operated and thereby which component to be added (B or C), is specified by the INLET_VALVE parameter. The allowable selections for the INLET_VALVE parameter are

```
PHASE  SUBR  CLEANING_REACTOR
    DECLARATIONS
        TIMER  TIMER1  (SEC)
        MONITOR  BASIC_INTERLOCKING  STOPPED
    NORMAL  LOGIC
        START  BASIC_INTERLOCKING
        CALL  INITIALIZE_VALVES
        SET  AG1.CO  =  ON
        SET  FV501.CO  =  OPEN
        START  TIMER1  300  {SET FOR 300 SEC}
        WAIT  UNTIL  (TIMER1.ALM  =  EXPIRED)
        SET  FV501.CO  =  CLOSED
        SET  FV508.CO  =  OPEN
        RESET  TIMER1  (0)
        START  TIMER1  600
        WAIT  UNTIL  (TIMER1.ALM  =  EXPIRED)
        SET FV508.CO  =  CLOSED
        SET AG1.CO  =  OFF
    FAULT  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV501.CO  =  CLOSED
        SET  FV508.CO  =  CLOSED
        {THIS SHUTS FEED TO THE REACTOR AND CLOSES VALVE TO
        SOLVENT RECOVERY}
ENDSUBR  CLEANING_REACTOR
```

Figure 2-12. Example Cleaning Reactor Phase Subroutine

specified by the selection list (“= (FV502, FV503)”). Thus the unit recipe will specify the following:

INLET_VALVE	Determine the component to add.
CHARGE_AMOUNT	Determines the amount of component to add.
DEFAULT_TIME	Determine the maximum time allowed to add the component.

The selected component is added to the reactor by opening the valve specified by the recipe and integrating the flow. When the integrated value (FIQ.VAL) is equal to or greater than the CHARGE_AMOUNT value, the inlet valve is closed.

This phase subroutine adds component B or C to the reactor by opening valve FV502 or FV503, integrating the flow and comparing it to the amount called for by the unit procedure. When the integrated value is equal to or greater than the amount called for, valve FV502 or FV503 closes.

The basic interlocking monitor subroutine performs the interlocking on this phase subroutine. In declaring the monitor subroutine, it is running unless the declaration specifies it stopped. Therefore, it is unnecessary to start the monitor in either the normal or continuous logic. Additionally, there is a test quality check on the flow signal to ensure a good 4 to

20-mA signal. There is also a default time. This means the phase subroutine can be active only so much time before generating a fault. Refer to Figure 2-13 for an example of the phase subroutine for adding component B or C.

```
PHASE  SUBR  ADD_B_OR_C
    {THIS PHASE WILL CHARGE THE REACTOR WITH EITHER COMPONENT B OR C. THE COMPONENT
    SELECTION IS MADE BY THE INLET_VALVE PARAMETER (FV502 = B, FV503 = C).
    "CHARGE_AMOUNT" SPECIFIES THE AMOUNT TO BE CHARGED.}
        DD  INLET_VALVE  =  (FV502,  FV503)
        ANY  CHARGE_AMOUNT
        ANY  DEFAULT_TIME
    DECLARATIONS
        MONITOR  BASIC_INTERLOCKING
        TIMER  TIMER1  (SEC)
        INTEGRATOR  FIQ
    CONTINUOUS
        IF  (FT503.Q  =  BAD)  FAULT  1
        START  TIMER1  DEFAULT_TIME
        IF  (TIMER1.ALM  =  EXPIRED)  FAULT  2
        IF  (LT502.Q  =  BAD  OR LT502.VAL  >  100)  SET  AG1.CO  =  ON
    NORMAL  LOGIC
        CALL  INITIALIZE_VALVES
        SET  INLET_VALVE.CO  =  OPEN
        START  FIQB
        WAIT  UNTIL  (FIQ.VAL  >=  CHARGE_AMOUNT)
        SET  INLET_VALVE.CO  =  CLOSED
    FAULT  LOGIC
        CALL  INITIALIZE_VALVES
        SET  INLET_VALVE.CO  =  CLOSED
        IF  (FAULT  CODE  =  1)  DISPLAY  MESSAGE  1  COLOR  RED  USING  TEXT_LINE_1
        IF  (FAULT  CODE  =  2)  SET  OPER_MSG.OSTR  =  "DEFAULT  TIME  ELAPSED"
    HOLD  LOGIC
        CALL  INITIALIZE_VALVES
        SET  INLET_VALVE.CO  =  CLOSED
ENDSUBR  ADD_B_OR_C
```

Figure 2-13. Example Add Component B or C Phase Subroutine

One other operation that needs to be controlled is the agitator. If the level in the tank is greater than 100 gallons or the transmitter tests bad, the agitator starts.

Two items pass from the unit procedure. These are given the type ANY, indicating numeric formulation values. Other types (such as DD for device drivers) can be passed from the unit procedure to the phase subroutine.

ADD_D PHASE SUBROUTINE

This phase subroutine is also similar to phase subroutine ADD_B_OR_C with the following exceptions: addition of component D is very exothermic. Therefore, the reactor protection monitor subroutine must be running. If the operator puts the reactor into hold, there must be hold logic to freeze the reactor (stop the reaction and put it into a safe state). There must be sufficient restart logic to allow the reactor's cooling jacket to switch to water and for the temperature to recover near set point before progressing.

The temperature controller set point is set to the value passed from the unit recipe. When the temperature set point is within one degree of the process variable, the phase subroutine ends.

Finally, there is a cook time specified in the unit recipe. Control must remain in this phase subroutine for a long enough period to complete the reaction. Figure 2-14 shows the logic for phase subroutine ADD_D.

RAMP TEMPERATURE OF REACTOR PHASE SUBROUTINE

This phase subroutine ramps the reactor temperature from one unit recipe parameter value to another. This phase subroutine runs during the exothermic part of the reaction (the reactor protection monitor subroutine must be used). Refer to Figure 2-15 for an example of the phase subroutine that ramps the reactor temperature.

BRINE_ON_JACKET PHASE SUBROUTINE

The jacket coolant switches from cooling water to low temperature brine. The jacket flushes with brine and then the brine return opens. The purpose of this is to prevent dilution of the brine system with water. Should there be a failure of the valves during this phase subroutine, the reactor freezes; the low temperature brine goes on and stays on. Refer to Figure 2-16 for the phase subroutine BRINE_ON_JACKET logic.

DUMP_PRODUCT PHASE SUBROUTINE

This phase subroutine opens the product discharge valve and allows material to go to the product header. After the level in the tank drops below 100 gallons, the tank flushes for five minutes with solvent. This material goes to the product header as well. If there is a fault during this phase subroutine, all valves shut and the operator receives alarms. Refer to Figure 2-17 for an example of a phase subroutine that dumps the product.

BATCH SEQUENCE FUNCTION CODE

Batch 90 language programs need to be linked to operator consoles using standard INFI 90® OPEN exception reporting function blocks. The batch sequence function code (function code 148) has inputs and outputs designated to interface with existing function blocks that generate exception reports. Refer to the ***Function Code Application Manual*** for detailed

® Registered trademark of Elsag Bailey Process Automation.

```
PHASE  SUBR  ADD_D
        ANY  COOK_TIME
        ANY  DEFAULT_TIME
        ANY  REACTOR_TEMP
    DECLARATIONS
        VAR  X
        MONITOR  BASIC_INTERLOCKING
        MONITOR  REACTOR_PROTECTION  (REACTOR_TEMP)
        TIMER  TIMER1  (SEC)
        TIMER  TIMER2  (SEC)
        TIMER  TIMER3  (SEC)
        INTEGRATOR  FIQD  (FT503.VAL,  MIN)
    CONTINUOUS
        IF  (FT503.Q  =  BAD)  FAULT  1
        START  TIMER1  DEFAULT_TIME
        IF  (TIMER1.ALM  =  EXPIRED)  FAULT  2
        IF  (LT502.Q  =  BAD  OR  LT502.VAL  =>  100)  SET AG1.CO  =  ON
    NORMAL  LOGIC
        CALL  INITIALIZE_VALVES
        START  TIMER3  COOK_TIME
        SET  FV506.CO  =  CLOSED
        SET  FV511.CO  =  CLOSED
        SET  FV505.CO  =  OPEN
        SET  FV512.CO  =  OPEN
        SET  P1.MODE  =  AUTO
        SET  P1.CO  =  ON
        SET  TC501B.MODE  =  MANUAL
        SET  TC501A.SP  =  REACTOR  TEMP
        SET  TC501A.MODE  =  AUTO
        REPEAT
            X  =  ABS  (TC501A.PV  -  TC501A.SP)
        UNTIL  X  >  1
        SET  FV504.CO  =  OPEN
        START  FIQD
        WAIT  UNITL  (FIQD.VAL  =>  TOTAL_FLOW_D)
        SET  FV504.CO  =  CLOSED
        WAIT  UNTIL  (TIMER3.ALM  =  EXPIRED)
    FAULT  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV504.CO  =  CLOSED
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        SET TC501A.SP  =  -20
        SET  TC501A.MODE  =  AUTO
        START  TIMER2  300
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
    HOLD  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV504.CO  =  CLOSED
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        SET  TC501B.SP  =  -20
        SET  TC501B.MODE  =  AUTO
        START  TIMER2  100
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
    RESTART  LOGIC
        SET  FV506.CO  =  CLOSED
        SET  FV505.CO  =  CLOSED
        SET  FV511.CO  =  CLOSED
        SET  FV512.CO  =  CLOSED
ENDSUBR  ADD_D
```

Figure 2-14. Example Add Component D Phase Subroutine

```
PHASE  SUBR  RAMP_TEMPERATURE
        ANY  RATE1
        ANY  DEFAULT_TIME
        ANY  INITIAL_TEMP
        ANY  FINAL_TEMP
    DECLARATIONS
        MONITOR  BASIC_INTERLOCKING
        MONITOR  REACTOR_PROTECTION  (FINAL  TEMP)
        TIMER  TIMER1  (SEC)
        TIMER  TIMER2  (SEC)
        RAMP  RAMP1  (TC501.SP,  SEC)
    CONTINUOUS
        START  BASIC_INTERLOCKING
        START  REACTOR_PROTECTION
        START  TIMER1  DEFAULT_TIME
        IF  (TIMER1.ALM  =  EXPIRED)  FAULT  1
        IF  (LT502.Q  =  BAD  OR  LT502.VAL  =>  100)  SET AG1.CO  =  ON
    NORMAL  LOGIC
        START  RAMP1  FROM  INITIAL_TEMP  TO  FINAL_TEMP  AT  RATE1
        START  TIMER2  300
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        FAULT  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        SET TC501B.MODE  =  MANUAL
        SET  TC501B.CO  =  0
        SET  TC501A.SP  =  -20
        SET  TC501A.MODE  =  AUTO
        START  TIMER2  300
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
    HOLD  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV504.CO  =  CLOSED
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        SET  TC501B.MODE  =  MANUAL
        SET  TC501B.CO  =  0
        SET  TC501A.SP  =  -20
        SET  TC501A.MODE  =  AUTO
        START  TIMER2  100
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
ENDSUBR  RAMP_TEMPERATURE
```

Figure 2-15. Example Ramp Reactor Temperature Phase Subroutine

information about the batch sequence function code. A properly configured batch sequence function block allows:

- The running or holding of the batch program execution.
- Executed stopping of the batch program.

```
PHASE  SUBR  BRINE_ON_JACKET
        ANY  REACTOR_TEMP
    DECLARATIONS
        MONITOR  BASIC_INTERLOCKING  RUNNING
        TIMER  TIMER2  (SEC)
    NORMAL  LOGIC
        CALL  INITIALIZE_VALVES
        SET  TC501B.MODE  =  MANUAL
        SET  TC501B.CO  =  0
        SET  TC501A.SP  =  REACTOR_TEMP
        SET  TC501A.MODE  =  AUTO
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        WAIT  FOR  100  SEC
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
    FAULT  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        SET TC501B.MODE  =  MANUAL
        SET  TC501B.CO  =  0
        SET  TC501A.SP  =  -20
        SET  TC501A.MODE  =  AUTO
        START  TIMER2  300
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
    HOLD  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV504.CO  =  CLOSED
        SET  FV506.CO  =  OPEN
        SET  FV505.CO  =  CLOSED
        SET  TC501A.SP  =  -20
        SET  TC501A.MODE  =  AUTO
        START  TIMER2  100
        WAIT  UNTIL  (TIMER2.ALM  =  EXPIRED)
        SET  FV511.CO  =  OPEN
        SET  FV512.CO  =  CLOSED
ENDSUBR  BRINE_ON_JACKET
```

Figure 2-16. Example Put Jacket on Low Temperature Brine Phase Subroutine

- Resetting the batch program from a previous unit recipe to a new unit recipe when a previous batch sequence aborts abnormally.
- Acknowledgment of messages.
- Changing an operation number within a unit recipe.
- Changing a unit recipe number when a batch program completes or is in holding.

Figure 2-18 shows the simplest configuration of a batch sequence function block interfaced with a remote manual set constant (RMSC) and remote control memory (RCM) function blocks. The remote manual set constants hold the values of the

```
PHASE  SUBR  DUMP_PRODUCT
    DECLARATIONS
        MONITOR  BASIC_INTERLOCKING  RUNNING
        TIMER  TIMER1  (SEC)
    NORMAL  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV509.CO  =  OPEN
        WAIT  UNTIL  (LT502.VAL  =  100)
        START  TIMER1  300
        SET  FV501.CO  =  OPEN
        WAIT  UNTIL  (TIMER1.ALM  =  EXPIRED)
        SET  FV501.CO  =  CLOSED
        WAIT  FOR  100  SEC
        SET  FV509.CO  =  CLOSED
        DONE
    FAULT  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV501.CO  =  CLOSED
        SET  FV503.CO  =  OPEN
        DISPLAY  MESSAGE  25  COLOR  RED  BLINKING  ON  USING  TEXT_LINE_4
        CALL  INITIALIZE_VALVES
    HOLD  LOGIC
        CALL  INITIALIZE_VALVES
        SET  FV501.CO  =  CLOSED
        SET  FV509.CO  =  OPEN
        DISPLAY  MESSAGE  25  COLOR  RED  BLINKING  ON  USING  TEXT_LINE_4
ENDSUBR  DUMP_PRODUCT
```

Figure 2-17. Example Dump Product Phase Subroutine

unit recipe and operation number. When the batch program is running, the remote manual set constant function blocks track the current unit recipe and operation number being executed. This allows the operator to determine what the current unit recipe and operation number are. The unit recipe and operation number can not be changed while the program is executing.

NOTE: Using the function code logic in Figure 2-18, the operation number will increment to (last operation + 1) as defined by the unit recipe. To restart the batch, select the starting operation number and cycle the run/hold push button to run. Failure to select the starting operation causes a -15 runtime fault code or the unintentional starting of a new recipe.

If the sequence stops executing, the track signals stop. At this point the operator can change the unit recipe and operation number using the remote manual set constant function blocks.

The program can stop executing in two ways. The first way is for the operator to hold the program execution. The second way is for fault logic within the program to cause the stop.

Refer to Appendix A for a more detailed CAD configuration.

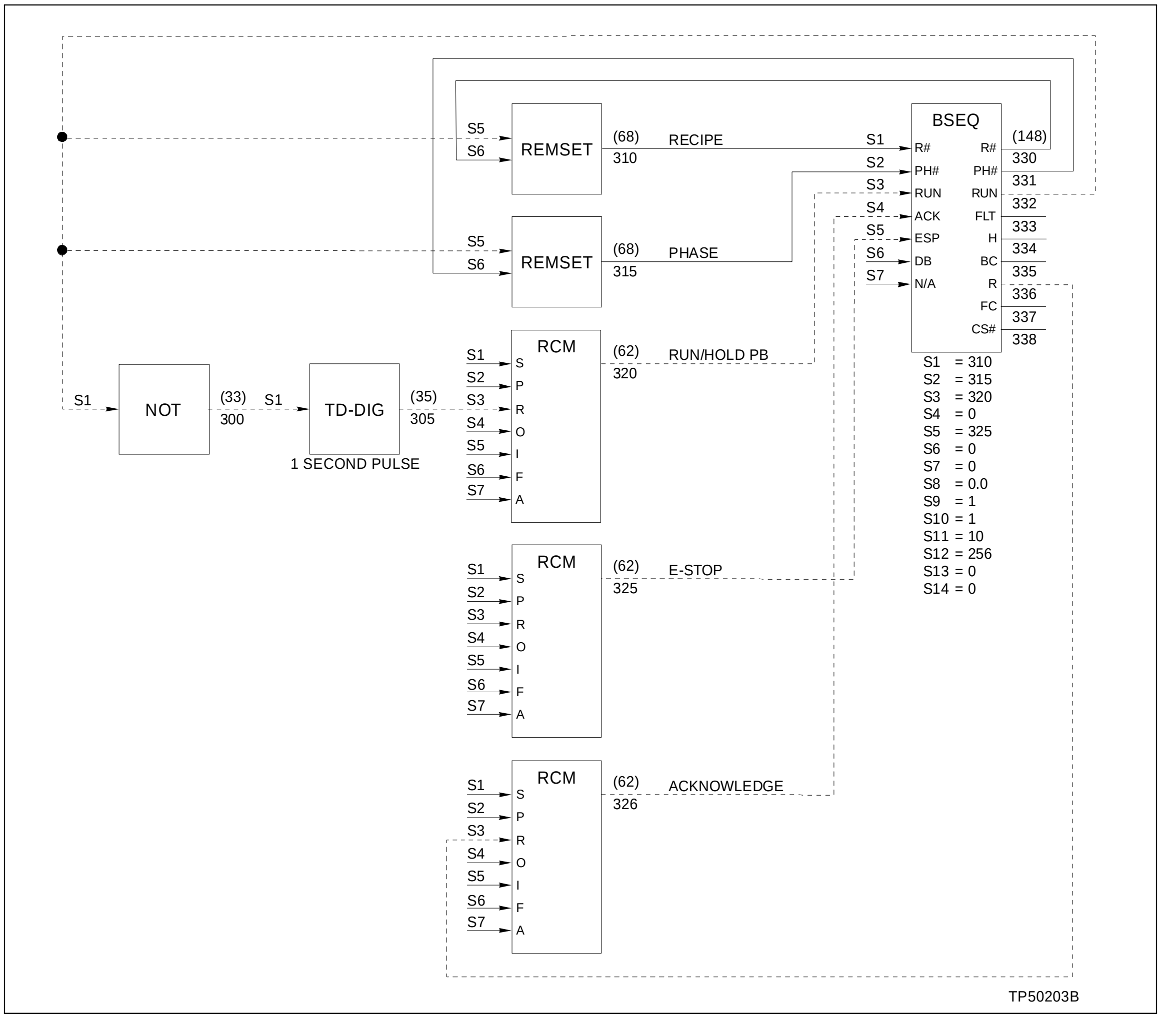


Figure 2-18. Example Batch Sequence Interface Logic

SECTION 3 - USER DEFINED FUNCTION (UDF) DESCRIPTION

INTRODUCTION

This section describes what a UDF program is, what it does, and how it functions like a function code.

UDF PROGRAMS

A UDF program is a plan for executing a sequence of activities. The program consists of a data section and a set of subroutines. There are three types of subroutines: state subroutine, function subroutine, and monitor subroutine. Each UDF declaration function block will have one UDF program associated with it. Several UDF Type 1 (UDF1) and UDF Type 2 (UDF2) function blocks can reference the same UDF declaration function block.

Use the editor within the Batch Data Manager to write UDF programs. If necessary, any word processor that outputs an ASCII formatted file can be used to create batch files. The text file is the source file. The compiler processes the ASCII file and checks the syntax before generating the executable object file.

This is a general overview of these program elements (refer to Section 4 for detailed explanations).

Data

The data section defines the data (called global data) that is available to all subroutines in a given UDF program. Examples of information of global importance are control loops, devices (pumps, valves, etc.) and function blocks used to solve calculation or interlock problems. The advantages of global data declaration is that the data is permanent, defined only once, and available to all subroutines.

There are five types of global data:

- Inputs to UDF1, UDF2, and Auxiliary UDF function blocks.
- Outputs from UDF1, UDF2, and Auxiliary UDF function blocks.
- Specifications of UDF1, UDF2, and Auxiliary UDF function blocks.
- Function blocks.
- Variables and constants.

State Subroutine

The state subroutine is a major part of a UDF program. Every UDF program must have one or more state subroutines. They are responsible for controlling the flow of execution. Figure 3-1 shows the structure of a state subroutine.

```
STATE SUBR name    (HEADING section)
DECLARATIONS
      Local data and monitor subroutine calls
CONTINUOUS
      Executable statement
NORMAL LOGIC
      Executable statements
FAULT LOGIC
      Executable statements
ENDSUBR  [name]
```

Figure 3-1. State Subroutine Structure

Heading Section

This section defines the state subroutine name. The name allows the program to address a state subroutine by a meaningful title.

Declarations Section

The declarations section defines the state subroutine local data and specifies calls to local monitor subroutines. Local data typically has timers, integrators, ramps, variables, etc. Local data can only be used by the state subroutine that declared it. Also, this data is temporary; it is only valid while the state subroutine is active. Local data divides into three types:

- Variables.
- Constants.
- Active Data Structures.

State subroutines have access to all data (including function blocks) declared in any data section of the UDF program that comes before the state subroutine. No additional declaration is necessary or allowed inside the state subroutine.

Continuous Section

The entire continuous section executes every function block cycle in parallel with the sequential logic (NORMAL and FAULT). It is ideal for defining interlock logic that must be performed continuously and is specific to the state subroutine. The continuous section prohibits commands that can result in a wait condition. Examples of prohibited commands are REPEAT, WAIT, WAIT UNTIL, etc.

Normal Logic

Normal logic is an executable section that performs the normal sequential logic of a state subroutine. It begins when the state subroutine starts. It remains in operation until:

1. The mode of the state subroutine changes from NORMAL to FAULT. The mode typically changes to FAULT by some interlock failure detected by the continuous section or a monitor subroutine.

2. The execution of a NEXT STATE statement causes the current state subroutine to become inactive. The next subroutine specified by the NEXT STATE statement becomes active.

Fault Logic

Fault logic is the section that executes user defined statements when a fault condition occurs. When a fault occurs in the normal logic, continuous logic, or a monitor subroutine, execution of normal logic suspends and control passes to the fault logic. In the fault logic, define the executable commands necessary to hold or stop the sequence safely while solving the problem.

Function Subroutine

The function subroutine allows the one time definition of commonly used logic and the reuse of this subroutine as often as needed. This aids in reducing the program size. A function subroutine may be called from a state subroutine, a monitor subroutine or another function subroutine. A function subroutine takes control from the caller, executes to completion, and then returns control to the caller. Figure 3-2 shows the function subroutine structure.

```
FUNCTION name    (HEADING section)
      Parameter list    (HEADING section)
DECLARATIONS
      Local data
EXECUTABLE
      Executable statements
ENDSUBR  [name]
```

Figure 3-2. Function Subroutine Structure

Heading Section

This section defines the function name and parameter list.

Declarations Section

Local data refers to data used solely within the subroutine. This data is valid only while the subroutine is active. Only constants and variables can be declared local to a function subroutine.

Executable Section

Execution starts with the first statement of the executable section and continues until a RETURN or ENDSUBR statement executes. Then, control transfers back to the caller. If a function subroutine is called from normal logic, it can contain wait states. Wait states will be ignored if a function subroutine is called from a continuous section or a monitor subroutine.

Monitor Subroutines

Monitor subroutines execute interlock logic common to several state subroutines. Once defined, monitor subroutines can be called by multiple state subroutines. Executed continuously at the function block cycle, monitor subroutines operate in parallel with normal logic and effectively watch process interlock conditions. Fault mode operates when the monitor subroutine sets a fault flag. Normal logic then stops and control passes to the fault logic. Refer to Figure 3-3 for the monitor subroutine structure.

```
MONITOR name    (HEADING section)
    Parameter list    (HEADING section)
DECLARATIONS
    Local data
CONTINUOUS
    Executable statements
ENDSUBR  [name]
```

Figure 3-3. Monitor Subroutine Structure

Heading Section

This section defines the monitor subroutine name and parameter list. The list defines the local parameter aliases (names used solely in the monitor subroutine).

Declarations Section

Only variables, variable arrays, constants and constant arrays can be declared in a monitor subroutine.

Continuous Section

The monitor subroutine has only a continuous logic section. Continuous logic executes every function block cycle. A monitor subroutine may not contain any statements that can result in a wait condition (for example, REPEAT, WAIT UNTIL, etc.).

UDF PROGRAM EXECUTION

Start-up of the Multi-Function Processor (MFP) modules by either powering up or changing from configure to execute mode causes the MFP modules to scan for all UDF configuration blocks before normal function block execution. The MFP modules will then load object files from NVRAM memory of the module into the dynamic RAM memory defined by the configuration block. If there is not enough memory allocated for the program to reside in or if there is no such object file, the module will go into error mode. After successful loading of the UDF files, the MFP module checks the UDF object file to verify all declarations to function blocks are correct. If an error occurs, the module goes into error mode.

Assuming all diagnostic tests pass, normal function block execution will begin. Each UDF type one or type two function block uses a declaration block and will contain the data values specific to the particular function block. All UDF function blocks can access the same UDF program but contain within themselves unique variables, timers, etc. pertaining to its own function block.

UDF PROGRAM CONSTRUCTION

This section illustrates how to construct a UDF program through an example. This example will show how to:

- Create the data section.
- Create monitor subroutines.
- Create function subroutines.
- Create state subroutines.

Figure 3-4 shows a diagram of a simple UDF process. The UDF process has several devices, control loops associated with it. These processes must be defined as function blocks (normally with the engineering work station). Then, add the function blocks to the program with data declaration statements. The example declares five variables, five function blocks (with associated MFP module addresses), and thirteen UDF block inputs.

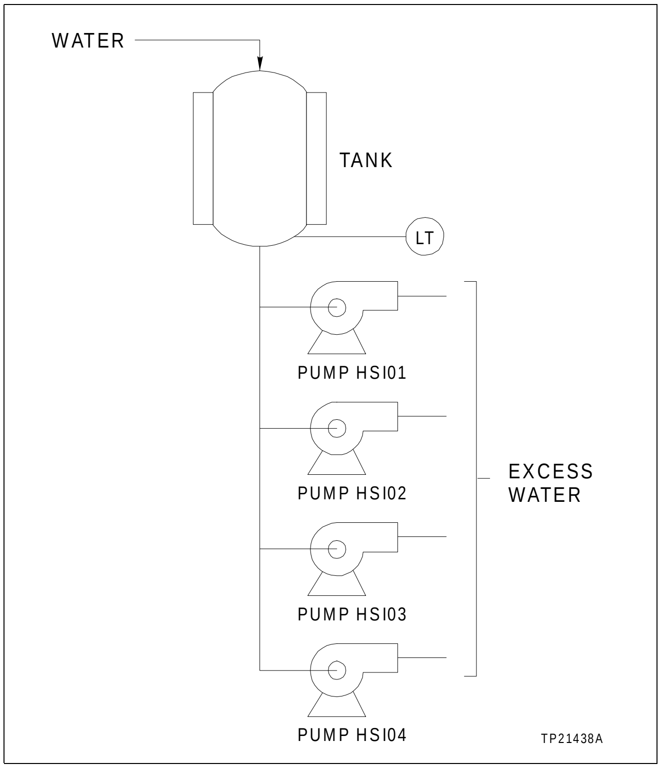


Figure 3-4. Example UDF Process

The block command identifies a configurable name that is assigned to a function block (for example, HS101) and function block type (for example, FC = DD meaning device driver). Device driver has an additional descriptor for whether it is normal (zero output closes the valves) or inverted (zero output opens the valve). If not specified, the system will default to a normal status on a device driver.

Any function code can be declared and its value read by the UDF program. For certain types of function codes (device drivers, multi-state device drivers, control stations, remote control memories, remote manual set constants, etc.) additional attributes are available. Reference extensions identify the attributes. Reference extensions give the UDF language the ability to read or command attributes of these functions. For example, with a control station, the command CSNAME.SP allows the UDF language to reference or change the set point of the declared control station called CSNAME. A detailed explanation of the reference extensions is in Section 4 of this document. Refer to Figure 3-5 for a data example.

It is possible to declare function blocks in the data section that do not exist in the module. Similarly, it is possible to declare function blocks incorrectly (for example, assign wrong function code numbers). The UDF compiler does not check for these types of errors. Before the UDF declaration function block goes into run mode, it will check to make sure that all function block declarations are correct. If not, the MFP module will go into error mode and the block address of the incorrect declaration is indicated by status bytes of the MFP module.

Figure 3-6 is an example configuration for the level controller and each pump. Figure 3-7 is an example UDF function block configuration.

Monitor Subroutine Definition

It is necessary to monitor the status of the pumps in the system. The status is used to determine which pump is to be started next and which pump is to be stopped next based on run time. Figure 3-8 shows the configuration of this subroutine.

Function Subroutine Definitions

Pumps must be started and stopped in this example program. Function subroutines START_PUMP and STOP_PUMP perform these pump operations. Figure 3-9 shows the function subroutine that starts up a pump. Figure 3-10 shows the function subroutine that stops a pump.

```
DATA
    UDF TYPE2

    INPUT 1 HS101_IN_SERVICE
    INPUT 2 HS102_IN_SERVICE
    INPUT 3 HS103_IN_SERVICE
    INPUT 4 HS104_IN_SERVICE
    INPUT 5 HS101_RUN_TIME
    INPUT 6 HS102_RUN_TIME
    INPUT 7 HS103_RUN_TIME
    INPUT 8 HS104_RUN_TIME
    INPUT 9 HS101_FEEDBACK
    INPUT 10 HS102_FEEDBACK
    INPUT 11 HS103_FEEDBACK
    INPUT 12 HS104_FEEDBACK
    INPUT 13 PUMPS_RUNNING

    BLOCK HS101, BLK = 110, FC = DD
    BLOCK HS102, BLK = 120, FC = DD
    BLOCK HS103, BLK = 130, FC = DD
    BLOCK HS104, BLK = 140, FC = DD
    DD ARRAY PUMP (1:4) (HS101, HS102, HS103, HS104)
    BLOCK TANK_LEVEL, BLK = 100, FC = CS

    VAR I
    VAR HIGHEST_RUN_TIME
    VAR LOWEST_RUN_TIME
    VAR NEXT_PUMP_TO_START
    VAR NEXT_PUMP_TO_STOP
END DATA
```

Figure 3-5. Example Data Section

State Subroutine Definitions

This process has two activities that need to be defined. They are:

- Initializing the pumps.
- Running the pumps

The INITIALIZE START state subroutine puts the device drivers into automatic mode and turns tracking on. This enables the bumpless transfer between automatic and manual mode. This subroutine also checks which pumps are running and sets their device driver so that the control stations will follow this field when the MFP module powers up. Refer to Figure 3-11.

The RUN_PUMPS state subroutine has two modes. In manual mode, the device drivers are put into manual mode and the pumps must be operated by the operators. In automatic mode, the device drivers are put into automatic mode and are started and stopped based on tank level. If one of the pumps fails, the next pump is started to insure the appropriate number of pumps are running. Refer to Figure 3-12.

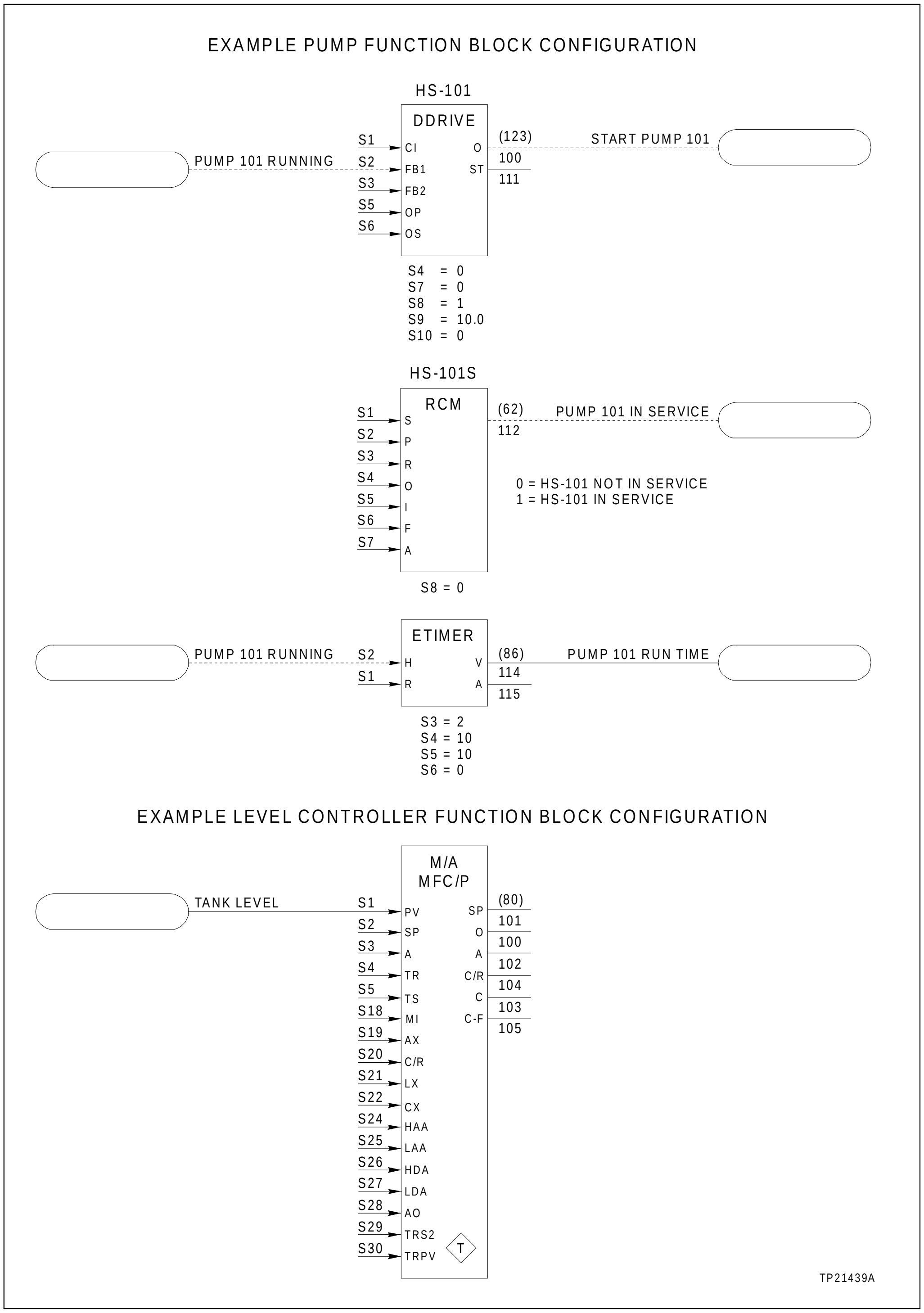


Figure 3-6. Example Level Controller and Pump Function Block Configurations

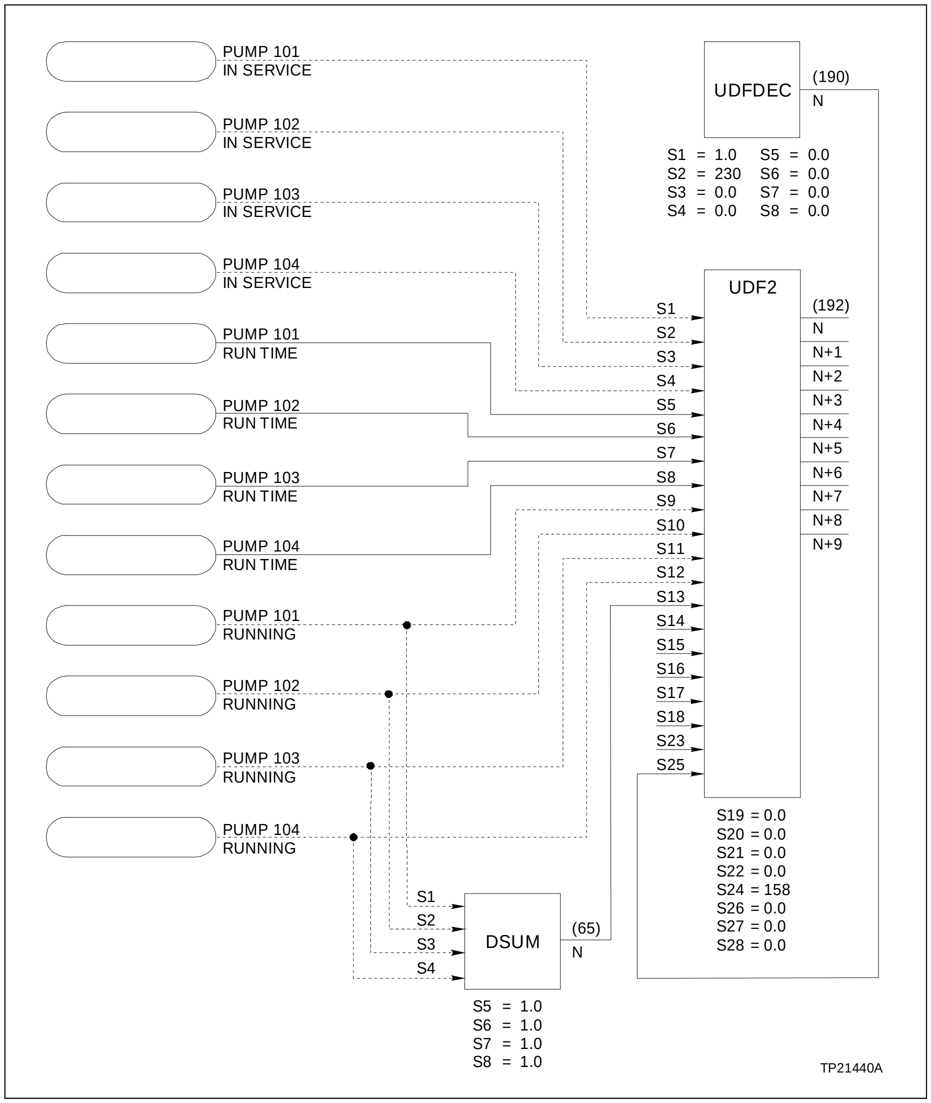


Figure 3-7. Example UDF Function Block Configuration

```
MONITOR  PUMP_STATUS
    CONTINUOUS
        IF  (HS101_IN_SERVICE.VAL  =  ON)  AND (HS101.CO  =  OFF)  THEN  LOWEST_RUN_TIME  =  \
            HS101_RUN_TIME.VAL
            NEXT_PUMP_TO_START  =  1
        ENDIF
        IF  (HS102_RUN_TIME.VAL  <  LOWEST_RUN_TIME)  AND  (HS102_IN_SERVICE.VAL  =  ON)  \
            AND  (HS101.CO  =  OFF)  THEN  LOWEST_RUN_TIME  =  HS102_RUN_TIME.VAL
            NEXT_PUMP_TO_START  =  2
        ENDIF
        IF  (HS103_RUN_TIME.VAL  <  LOWEST_RUN_TIME)  AND  (HS103_IN_SERVICE.VAL  =  ON)  \
            AND  (HS101.CO  =  OFF)  THEN  LOWEST_RUN_TIME  =  HS103_RUN_TIME.VAL
            NEXT_PUMP_TO_START  =  3
        ENDIF
        IF  (HS104_RUN_TIME.VAL  <  LOWEST_RUN_TIME)  AND  (HS104_IN_SERVICE.VAL  =  ON)  \
            AND  (HS101.CO  =  OFF)  THEN  LOWEST_RUN_TIME  =  HS104_RUN_TIME.VAL
            NEXT_PUMP_TO_START  =  4
        ENDIF
        IF  (HS101.CO  =  ON)  THEN  HIGHEST_RUN_TIME  =  HS101_RUN_TIME.VAL
            NEXT_PUMP_TO_STOP  =  1
        ENDIF
        IF  (HS102_RUN_TIME.VAL  >  HIGHEST_RUN_TIME)  AND  (HS102.CO  =  ON)  THEN  \
            HIGHEST_RUN_TIME  =  HS102_RUN_TIME.VAL
            NEXT_PUMP_TO_STOP  =  2
        ENDIF
        IF  (HS103_RUN_TIME.VAL  >  HIGHEST_RUN_TIME)  AND  (HS103.CO  =  ON)  THEN  \
            HIGHEST_RUN_TIME  =  HS103_RUN_TIME.VAL
            NEXT_PUMP_TO_STOP  =  3
        ENDIF
        IF  (HS104_RUN_TIME.VAL  >  HIGHEST_RUN_TIME)  AND  (HS104.CO  =  ON)  THEN  \
            HIGHEST_RUN_TIME  =  HS104_RUN_TIME.VAL
            NEXT_PUMP_TO_STOP  =  4
        ENDIF
ENDSUBR  PUMP_STATUS
```

Figure 3-8. Example Monitor Subroutine for Determining Pump Status

```
FUNCTION  START_PUMP
    DD  PUMP
    EXECUTABLE
        IF  (PUMP.MODE  ISNOT  AUTO)  THEN  SET  PUMP.MODE  =  \
        AUTO
            SKIP  CYCLE
        ENDIF
        SET  PUMP.CO  =  ON
        WAIT  UNTIL  PUMP.STS  ISNOT  WAITING
ENDSUBR  START_PUMP
```

Figure 3-9. Example Function Subroutine to Start a Pump

```
FUNCTION  STOP_PUMP
    DD  PUMP
    EXECUTABLE
        IF  (PUMP.MODE  ISNOT  AUTO)  THEN  SET  PUMP.MODE  =  \
        AUTO
            SKIP  CYCLE
        ENDIF
        SET  PUMP.CO  =  OFF
        WAIT  UNTIL  PUMP.STS  ISNOT  WAITING
ENDSUBR  STOP_PUMP
```

Figure 3-10. Example Function Subroutine to Stop a Pump

```
STATE  SUBR  INITIALIZE  START
    N ORMAL  LOGIC
        SET  HS101.TRK  =  ON
        SET  HS102.TRK  =  ON
        SET  HS103.TRK  =  ON
        SET  HS104.TRK  =  ON
        SET  HS101.MODE  =  AUTO
        SET  HS102.MODE  =  AUTO
        SET  HS103.MODE  =  AUTO
        SET  HS104.MODE  =  AUTO
        IF  (HS101_FEEDBACK.VAL  =  ON)  THEN  SET HS101.CO  =  ON
        ENDIF
        IF  (HS102_FEEDBACK.VAL  =  ON)  THEN  SET HS102.CO  =  ON
        ENDIF
        IF  (HS103_FEEDBACK.VAL  =  ON)  THEN  SET HS103.CO  =  ON
        ENDIF
        IF  (HS104_FEEDBACK.VAL  =  ON)  THEN  SET HS104.CO  =  ON
        ENDIF
        NEXT  STATE  RUN_PUMPS
ENDSUBR  INITIALIZE
```

Figure 3-11. Example State Subroutine for Initializing the Pumps

```
STATE SUBR RUN_PUMPS
    DECLARATIONS
        MONITOR PUMP_STATUS RUNNING
    NORMAL LOGIC
        IF (TANK_LEVEL.MODE = MANUAL) THEN
            FOR I = 1 TO 4 DO
                SET PUMPS(I).MODE = MANUAL
            ENDFOR
        ELSE
            FOR I = 1 TO 4 DO
                SET PUMPS(I).MODE = AUTO
            ENDFOR
        IF ((PUMPS_RUNNING.VAL <= 1) AND (TANK_LEVEL.PV - TANK_LEVEL.SP > 1)) OR \
            ((PUMPS_RUNNING.VAL <= 2) AND (TANK_LEVEL.PV - TANK_LEVEL.SP > 2)) OR \
            ((PUMPS_RUNNING.VAL <= 3) AND (TANK_LEVEL.PV - TANK_LEVEL.SP > 3)) OR \
            ((PUMPS_RUNNING.VAL <= 4) AND (TANK_LEVEL.PV - TANK_LEVEL.SP > 4)) \
            THEN CALL START_PUMP (PUMPS (NEXT_PUMP_TO_START))
        ENDIF
        IF ((PUMPS_RUNNING.VAL => 1) AND (TANK_LEVEL.PV - TANK_LEVEL.SP < 0.5)) OR \
            ((PUMPS_RUNNING.VAL => 2) AND (TANK_LEVEL.PV - TANK_LEVEL.SP < 1.5)) OR \
            ((PUMPS_RUNNING.VAL => 3) AND (TANK_LEVEL.PV - TANK_LEVEL.SP < 2.5)) OR \
            ((PUMPS_RUNNING.VAL => 4) AND (TANK_LEVEL.PV - TANK_LEVEL.SP < 3.5)) \
            THEN CALL STOP_PUMP (PUMPS (NEXT_PUMP_TO_STOP))
        ENDIF
        WAIT FOR 1 MIN
    ENDIF
    NEXT STATE RUN_PUMPS
ENDSUBR RUN
```

Figure 3-12. Example State Subroutine for Running the Pumps

SECTION 4 - FUNCTION LIBRARY

INTRODUCTION

Batch 90 and User Defined Function Code (UDF) programming languages, like other programming language, have certain usage rules. These rules include the use of upper and lower case letters, punctuation, etc. This section presents general usage information and then explains syntax rules, commands, and statements. This section should be used to answer any questions about proper syntax.

SYNTAX

Syntax rules for the commands and statements consist of:

- How to use comments.
- Legal names.
- Command and statement length.
- Reserved words/constants/directives.
- Types of expressions.
- Relational, boolean, and arithmetic operators and expressions.
- Local and global conventions.

Comments

All comments inserted in Batch 90 or UDF programs begin with either { or/* and end with } or */. A comment can cross a line boundary. Comments can ***not*** be nested within other comments.

Example: {This is a comment} or /*This is a comment*/

Names

Legal characters are:

- a through z
- A through Z
- 0 through 9
- _ (underscore)

Example: NAME1

In source code, case (upper/lower) is not significant. In an application, case sensitivity can be enabled.

Batch 90 and UDF languages consider a hyphen a minus sign. Therefore, a hyphen should never be used as a separator between text.

First character must be alphabetic.

Maximum of 32 characters for a name except phase subroutine names which are limited to 16 characters.

Lines

One statement per line is permissible. A backslash (\) anywhere in a line causes the remainder of the display line to be ignored and the statement is continued on the next display line.

Reserved Words/Constants/Directives

Tables 4-1, 4-2, 4-3, and 4-4 provides listing of reserved words, predefined constants, reference extensions and directives respectively.

NOTE: Table entries marked (SP88) apply only when programming in ISA-S88.01-1995 standard terms. Table entries marked (Bailey) apply only when programming in Elsag Bailey traditional terms.

Table 4-1. Reserved Words

Reserved Word	B90	UDF	Reserved Word	B90	UDF
ABS	X	X	BBUF	X	X
ACK	X		BBUF ARRAY	X	X
ACQUIRE	X		BLINKING	X	X
ALIAS	X	X	BLK	X	X
ALPHA	X		BLOCK	X	X
AND	X	X	BREAK	X	X
AND WAIT		X	BSEQ	X	X
ANY	X	X	CALL	X	X
AOL	X	X	CAMPAIGN	X	
AOL ARRAY	X	X	CASE	X	X
APID	X	X	CASESENSE	X	X
APID ARRAY	X	X	CLASS	X	
ARRAY	X	X	COLOR	X	X
AT	X	X	COMMON		X
AUX		X	CONNECT	X	
BATCH	X		CONST	X	X
BATCH DATA	X		CONST ARRAY	X	X

Table 4-1. Reserved Words (continued)

Reserved Word	B90	UDF	Reserved Word	B90	UDF
CONST STRING	X	X	ENDDATA	X	X
CONST STRING ARRAY	X	X	ENDFOR	X	X
CONTINUOUS	X	X	ENDIF	X	X
CONTINUOUS LOGIC	X	X	ENDSUBR	X	X
COS	X	X	ENDWHILE	X	X
CS	X	X	ESTOP INPUT	X	
CS ARRAY	X	X	EXECUTABLE	X	X
CSEQ	X	X	EXP	X	X
CYCLE TIME	X	X	EXTERNAL	X	
CYCLETIME	X	X	EXTERNAL HOLD	X	
DATA		X	FALSE	X	X
DATAEXPT	X	X	FAULT	X	X
DATAEXPT ARRAY	X	X	FAULT CODE	X	X
DD	X	X	FAULT LOGIC	X	X
DD ARRAY	X	X	FC	X	X
DD INVERT	X	X	FGEN	X	X
DD NORMAL	X	X	FIRST CYCLE	X	X
DEBUG	X	X	FOR	X	X
DECLARATIONS	X	X	FROM	X	X
DESCRIPTOR	X	X	FUNCTION	X	X
DIGIT	X	X	HIHI	X	
DISABLE	X	X	HISTORIAN	X	
DISPLAY MESSAGE	X	X	HISTORIAN BUFFER	X	
DO	X	X	HISTORIAN STATUS	X	
DO CASE	X	X	HOLD	X	X
DOL	X	X	HOLD LOGIC	X	
DOL ARRAY	X	X	HOUR	X	X
DONE	X		IF	X	X
EJECT	X	X	INCLUDE	X	X
ELSE	X	X	INPUT		X
ENABLE	X	X	INTEGRATOR	X	X
END	X	X	INVERT	X	X
END CASE	X	X	IS	X	X
END DATA	X	X	IS NOT	X	X
END FOR	X	X	ISNOT	X	X
END IF	X	X	LAST OPERATION (SP88)	X	X
END SUBR	X	X	LAST PHASE (Bailey	X	X
END WHILE	X	X	LAST SUBR	X	X
ENDCASE	X	X	LIST	X	X

Table 4-1. Reserved Words (continued)

Reserved Word	B90	UDF	Reserved Word	B90	UDF
LOCATION	X		PHASE DESCRIPTOR (Bailey)	X	X
LOG	X	X	PHASE NUMBER (SP88)	X	
LOGIC		X	PHASE SUBR (SP88)	X	
LOOP	X	X	PRIORITY	X	
LOT	X		PROGRAM	X	
MAINTAIN	X		PROGRAM DESCRIPTOR	X	
MAJOR	X	X	PROGRAMID	X	
MAJOR ID	X	X	RAMP	X	X
MAJOR POINT	X	X	RBUF	X	X
MAXPARALLEL	X		RBUF ARRAY	X	X
MESSAGE	X	X	RCM	X	X
MIN	X	X	RCM ARRAY	X	X
MINE	X		RECIPE	X	
MINOR	X	X	RECIPE DESCRIPTOR	X	X
MINOR ID	X	X	RECIPE ENTRY	X	X
MINOR POINT	X	X	RECIPEID	X	
MODULE	X	X	REMSET	X	X
MONITOR	X	X	REMSET ARRAY	X	X
MSDD	X	X	REPEAT	X	X
MSDD ARRAY	X	X	RESERVE	X	
NEXT OPERATION (SP88)	X		RESET	X	X
NEXT PHASE (Bailey)	X		RESTART	X	X
NEXT STATE		X	RESTART LOGIC	X	
NOLIST	X	X	RESTART POINT	X	X
NORMAL LOGIC	X	X	RESUME	X	X
NORMALIZE	X	X	RESUME AT	X	X
NOT	X	X	RETURN	X	X
OPERATION (SP88)	X		RING	X	
OPERATION DESCRIPTOR (SP88)	X		RMC	X	X
OPTIONS	X		RMC ARRAY	X	X
OR	X	X	RND	X	X
OTHER	X	X	RUN INPUT	X	
OUT	X	X	SEC	X	X
OUTPUT		X	SEND	X	
PCU	X	X	SET	X	X
PHASE (Bailey)	X		SIN	X	X

Table 4-1. Reserved Words (continued)

Reserved Word	B90	UDF	Reserved Word	B90	UDF
SIZEOF	X	X	TOLOWER	X	X
SKIP CYCLE	X	X	TOUPPER	X	X
SMITH	X	X	TPR	X	
SMITH ARRAY	X	X	TRUE	X	X
SPEC		X	TRUNC	X	X
START	X	X	TUNED STATUS		X
STARTUP STATUS		X	UDF TYPE		X
STATE SUBR		X	UNACQUIRE	X	
STEP	X	X	UNCONNECT	X	
STEP SUBR (Bailey)	X		UNIT DATA	X	
STRING	X	X	UNIT DESCRIPTOR	X	X
STRING ARRAY	X	X	UNIT PARAMETERS	X	
SUBPHASE (Bailey)	X		UNRESERVE	X	
SUBR	X	X	UNTIL	X	X
SUBR DESCRIPTOR	X	X	USING	X	X
SUBR STATUS	X	X	VAR	X	X
SUPPRESS	X	X	VAR ARRAY	X	X
TAN	X	X	VAR STRING	X	X
TEXT	X	X	VERSION	X	
TEXT ARRAY	X	X	WAIT	X	X
THEN	X	X	WAIT FOR	X	X
TIMER	X	X	WAIT UNTIL	X	X
TIMER ARRAY	X	X	WAIT WHILE	X	X
TITLE	X		WATCH	X	
TO	X	X	WHILE	X	X
TO LOWER	X	X	XOR	X	X
TO UPPER	X	X			

Table 4-2. Reserved Constants

Constant	Value	B90	UDF	Constant	Value	B90	UDF
ALLOWED	1	X	X	GREEN	3	X	X
AUTO	1	X	X	FALSE	0	X	X
BAD	1	X	X	FORCED_OFF	0	X	X
BLACK	0	X	X	FORCED_ON	1	X	X
BLUE	4	X	X	HOLDING	0	X	X
CASCADE	2	X	X	INVERT	1	X	X
CLOSED	0	X	X	LOCAL	-3	X	
COMPLETE	0	X	X	MAGENTA	6	X	X
CYAN	5	X	X	MANUAL	0	X	X
GOOD	0	X	X	MINE	-2	X	

Table 4-2. Reserved Constants (continued)

Constant	Value	B90	UDF	Constant	Value	B90	UDF
NO	0	X	X	STATE_ THREE	3	X	X
NORMAL	0	X	X	STATE_TWO	2	X	X
NOT_FORCED	2	X	X	STATE_ZERO	0	X	X
OFF	0	X	X	STOPPED	0	X	X
ON	1	X	X	TRUE	1	X	X
OPEN	1	X	X	UNDEFINED	$-\infty$	X	X
PI	3.141 5	X	X	WAITING	2	X	X
RED	2	X	X	WHITE	1	X	X
RATIO	2	X	X	YELLOW	7	X	X
RUNNING	1	X	X	YES	1	X	X
STATE_ONE	1	X	X	YOURS	-1	X	

Table 4-3. Reserved Reference Extensions

Extension	Description	B90	UDF	Extension	Description	B90	UDF
.ALM	alarm status	X	X	.CV7	CSEQ command variable 7	X	
.AMSK	alarm mask	X	X	.CV8	CSEQ command variable 8	X	
.BATCH	batch identifier	X	X	.DAL	deviation alarm	X	X
.BLNK	blink status	X	X	.DILK	data export block data interlock status	X	X
.CAM-PAIGN	campaign identifier	X	X	.DIR	controller direction	X	X
.CLEN	current string length	X	X	.DTIM	dead time value	X	X
.CLR	message color	X	X	.FAULT	fault code	X	X
.CMDSTRG	CSEQ command string	X		.FB1	feedback value 1	X	X
.CO	control output	X	X	.FB2	feedback value 2	X	X
.CPEND	CSEQ command pending status	X		.FBT	feedback target	X	X
.CSTS	CSEQ connection status	X		.FBV	feedback value	X	X
.CV1	CSEQ command variable 1	X		.FF	feedforward value	X	X
.CV2	CSEQ command variable 2	X		.GAIN	gain value	X	X
.CV3	CSEQ command variable 3	X		.HAL	high alarm status	X	X
.CV4	CSEQ command variable 4	X		.HALIM	high alarm limit	X	X
.CV5	CSEQ command variable 5	X		.HAMSK	high alarm mask	X	X
.CV6	CSEQ command variable 6	X		.HLIM	high limit	X	

Table 4-3. Reserved Reference Extensions (continued)

Extension	Description	B90	UDF	Extension	Description	B90	UDF
.ILK1	interlock 1 value	X	X	.Q	quality	X	X
.ILK2	interlock 2 value	X	X	.QMSK	quality mask	X	X
.ILK3	interlock 3 value	X	X	.QSTS	CSEQ ownership queue status	X	
.ILK4	interlock 4 value	X	X	.RATE	rate	X	X
.ISTR	data export block input string value	X	X	.REALM	data export block re-alarm status	X	X
.K	gain multiplier	X	X	.RECIPE	recipe identifier	X	X
.KA	derivative lag constant	X	X	.RSTS	reservation status	X	
.KD	derivative rate action	X	X	.RX	ratio index	X	X
.KI	integral reset	X	X	.SP	set point	X	X
.KP	proportional gain	X	X	.SPM	set point modifier value	X	X
.LAG	lag value	X	X	.STAT-STRG	CSEQ status string	X	
.LAL	low alarm status	X	X	.STS	status	X	X
.LALIM	low alarm limit	X	X	.SV1	CSEQ status variable 1	X	
.LAMSK	low alarm mask limit	X	X	.SV2	CSEQ status variable 2	X	
.LIM	alarm limit	X	X	.SV3	CSEQ status variable 3	X	
.LLIM	low limit	X		.SV4	CSEQ status variable 4	X	
.LOT	lot identifier	X	X	.SV5	CSEQ status variable 5	X	
.MLEN	maximum string length	X	X	.SV6	CSEQ status variable 6	X	
.MLOCK	data export block mode lock status	X	X	.SV7	CSEQ status variable 7	X	
.MODE	mode	X	X	.SV8	CSEQ status variable 8	X	
.MSG	message number	X	X	.TIM	time	X	X
.NEW	data export block new input string status	X	X	.TPID	CSEQ target program ID number	X	
.OPERA-TION	operation number (SP88)	X	X	.TPR	CSEQ target partial recipe ID number	X	
.OSTR	data export block output string	X	X	.TRK	track	X	X
.OSTS	CSEQ ownership status	X		.TRUNC	data export block truncated value	X	X
.OVR	status override	X	X	.TUN	tuning time constant block	X	X
.PERM1	permissive 1 value	X	X	.TYP	algorithm type	X	X
.PERM2	permissive 2 value	X	X	.VAL	value	X	X
.PHASE	phase number (Bailey)	X	X	.WATCH	watch status	X	
.PRO-GRAM	CSEQ program ID number	X		.XSTS	execution status	X	X
.PV	process value	X	X				

Table 4-4. Reserved Directives

Directive	B90	UDF	Directive	B90	UDF	Directive	B90	UDF
ALPHA	X		HIHI	X		PROGRAMID	X	
BRM	X		INCLUDE	X	X	RECIPE MANAGEMENT	X	
CASESENSE	X		LIST	X	X	SUPPRESS	X	X
DEBUG	X	X	LOCATION	X		TITLE	X	X
DESCRIPTOR	X	X	MAXPARALLEL	X		VERSION	X	X
EJECT	X	X	NOLIST	X	X			

Expressions

An expression consists of one or more operands with an operator. Refer to Table 4-5 for a list of operators that apply to both Batch 90 and UDF programming languages.

Table 4-5. Operators and Expressions

Relational	Boolean	Arithmetic
<>	AND	+
>=	NOT	-
<=	OR	*
=<	XOR	**
=>		/
=		ABS
<		EXP
>		LOG
IS		RND
IS NOT		TRUNC
ISNOT		

Notation

Operators and comments require certain kinds of operands. The following notation is used to indicate operand type:

exp Any expression.

const Constant.

var Item of data that can be ***written***.

If several operands appear in an expression, they must be distinguished by appending a number (for example, exp1 + exp2).

Relational Operators and Expressions

The logical value ***false*** is represented by a value greater than or equal to -0.5 and less than 0.5. Otherwise, the value is ***true***. The values of relational and boolean expressions are 0.0 for false and 1.0 for true.

<> Usage: *exp1* <> *exp2*

True if exp1 is not equal to exp2; otherwise false.

Example: IF (FC501.VAL <> 25.0)

= Usage: *exp1* = *exp2*

True if exp1 equals exp2; otherwise false.

Example: WAIT UNTIL (LC501.VAL = 12.5)

< Usage: *exp1* < *exp2*

True if exp1 is less than or equal to exp2; otherwise false.

Example: WAIT UNTIL (LC501.VAL < LC502.VAL)

<=
=< Usage: *exp1* <= *exp2*
exp1 =< *exp2*

True if exp1 is less than or equal to exp2; otherwise false.

Examples: WAIT UNTIL (LC501.VAL <= 12.5)
WAIT UNTIL (LC501.VAL =< 12.5)

> Usage: *exp1* > *exp2*

True if exp1 is greater than exp2; otherwise false.

Example: WAIT UNTIL (LC501.VAL > 12.0)

>=
=> Usage: *exp1* >= *exp2*
exp1 => *exp2*

True if exp1 is greater than or equal to exp2; otherwise false.

Examples: WAIT UNTIL (LC501.VAL >= LC502.VAL)
WAIT UNTIL (LC501.VAL => LC502.VAL)

IS Usage: *exp1* IS *exp2*

True if exp1 (rounded to the nearest integer) is equal to exp2 (rounded to the nearest integer); otherwise false.

Example: IF (LC501.MODE IS AUTO)

IS NOT
ISNOT

Usage: *exp1* IS NOT *exp2*
exp1 ISNOT *exp2*

True if exp1 (rounded to nearest integer) is not equal to exp2 (rounded to nearest integer); otherwise false.

Examples:

```
IF (LC501.MODE IS NOT AUTO)
IF (LC501.MODE ISNOT AUTO)
```

VARIABLE TO LIST OR ARRAY COMPARISONS

The following expressions compare the value of a variable with a set of parameters to determine if the variable is included in the set. If the value of the variable is in the set, the result of the comparison is true. Otherwise, the result is false. Constants must be numeric or symbolic and can be mixed within a list. Array elements must also be numeric or symbolic.

(*var* **=** **(***const1*,*const2*,...,*constn***))**

(*var* **=** *array***(*))**

where:

var	Name of the variable to be compared to the list or array.
constn	Name of the constants to be compared to the variable.
array	Name of the array to be compared to the variable.

Examples:

```
IF (XMODE = (-i,OPEN,7)) THEN
    SET FC101.MODE = MAN
    SET FC101.CO = CLOSED
ENDIF

IF (X = ARYLIST(*)) THEN
    SET FC101.MODE = AUTO
ENDIF
```

VARIABLE TO LIMITS COMPARISONS

The following expressions compare the value of a variable with high and low limits to determine if the expression is true. The limits can be constants, variables, or expressions.

(*high***>***var***>***low***)**
(*high***>=***var***>***low***)**
(*high***>=***var***>=***low***)**
(*high***>***var***>=***low***)**
(*low***<***var***<***high***)**
(*low***<=***var***<***high***)**

(*low***<=***var***<=***high***)**
(*low***<***var***<=***high***)**

where:

var	Name of the variable to be compared to the limits.
high	High limit of the comparison.
low	Low limit of the comparison.

Examples:

```
IF  (95>XX>5)  THEN
     SET FC101.MODE  =  MANUAL
ENDIF

IF  (FC101.HLIM>ALM_VAR>FC101.LLIM)  THEN
     SET  FC101.MODE  =  MANUAL
     SET  SV201.CO  =  CLOSED
ENDIF
```

ARRAY COMPARISONS

The following expression compares the elements in one constant or variable array to the elements of a second constant or variable array. Only if there is an exact match is the expression considered to be true.

(*array1***(*) = ***array2***(*))**

where:

array1, *array2*	Name of the declared constant or variable arrays of the same dimensions.

Example:

```
IF  (OLD_ARRAY(*)  =  NEW_ARRAY(*))  THEN
          •
          •
          •
ENDIF
```

MINIMUM/MAXIMUM ARRAY VALUES

The following expressions select the minimum or maximum value from the elements of a constant or variable array.

MAX *array_name***(*)**
MIN *array_name***(*)**

where:

array_name	Name of the declared constant or variable array.

Examples: MAX_VALUE = MAX ALM_LIMIT(*)
MIN_VALUE = MIN SP_MIN_VAL(*)

Boolean Operators and Expressions

NOT Usage: NOT *exp*

True if exp false; otherwise false.

Example: WAIT UNTIL (NOT FC501.Q)

AND Usage: *exp1* AND *exp2*

Logical AND of exp1 and exp2. Value is true only if both exp1 and exp2 are true.

Example: WAIT UNTIL ((LC501.VAL >= 12.0) AND (LC501.MODE IS AUTO))

OR Usage: *exp1* OR *exp2*

Logical OR of exp1 and exp2. Value is true if either exp1 or exp2 is true.

Example: WAIT UNTIL ((LC501.VAL >= 12.0) OR (LC501.MODE IS AUTO))

XOR Usage: *exp1* XOR *exp2*

Logical EXCLUSIVE OR of exp1 and exp2. Value is true if either exp1 or exp2 is true, but not both.

Example: WAIT UNTIL ((LC501.VAL >= 12.0) XOR (LC501.MODE IS AUTO))

The boolean functions AND, OR, and XOR are also supported by arrays. For an AND expression to be true, both arrays must be identical. For an OR expression to be true, at least one element must be identical in both arrays. For an XOR expression to be true, every element is different than that of the other array.

(*array1***(*) AND** *array2***(*))**
(*array1***(*) OR** *array2***(*))**
(*array1***(*) XOR** *array2***(*))**

where:

array1, *array2* Name of the declared constant or variable arrays with the same dimensions.

Example:

```
IF (OLD_ARRAY(*) AND NEW_ARRAY(*)) THEN
        •
        •
        •
ENDIF
```

Arithmetic Operators and Expressions

+ Usage: *exp1* + *exp2*

Sum of exp1 and exp2.

- Usage: *exp1* - *exp2*

Difference of exp1 and exp2.

- Usage: -*exp*

Minus exp.

***** Usage: *exp1* * *exp2*

Product of exp1 and exp2.

****** Usage: *exp1* ** *exp2*

exp1 to exp2 power.

/ Usage: *exp1* / *exp2*

Quotient of exp1 divided by exp2.

ABS Usage: ABS (*exp*)

Absolute value of exp.

LOG Usage: LOG (*exp*)

Natural log of exp.

EXP Usage: EXP (*exp1*)

e to exp1 power.

RND Usage: RND (*exp*)

Round exp to nearest whole number.

TRUNC Usage: TRUNC (*exp*)

Truncate exp to whole number.

Trigonometric functions include:

SIN Usage: SIN (*exp*)

COS Usage: COS (*exp*)

TAN Usage: TAN (*exp*)

where *exp* = value in radians.

Local/Global Conventions

Data structures (timers, integrators, ramps, etc.) can be declared either in the phase subroutine or in a data section. When declared in a phase (Batch 90) or state (UDF) subroutine, the data structure is said to be local. When declared in a data section, the data structure is said to be global.

Local data structures only exist when the phase or state subroutine they are declared within is active. All local data structures are initialized when the phase or state subroutine becomes active. Most data structures (for example, timers, integrators, etc.) have initial values of zero.

Global data structures are initialized on a complete to run transition. Global data is available to all subroutines that follow for reference or command.

Any element (function subroutine, monitor subroutine, variables, function blocks, etc.) must be defined before it can be used in a program statement.

COMMANDS AND STATEMENTS

The purpose and format of each command and statement is explained in the following text. Also provided are examples and any other pertinent information. Unshaded areas under the language heading indicate languages that are supported. Shaded areas indicate languages that are not supported.

ACK

variable

PURPOSE: ACK is a built-in, read-only status variable. It indicates the status of the operator acknowledge input to the BSEQ function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: ACK value is read by using ACK in an expression.

ACK — Status of operator acknowledge input:
0 = false (no acknowledge).
1 = true (acknowledge has occurred).

Examples:

```
WAIT UNTIL (ACK)
WAIT UNTIL (ACK IS TRUE)
```

REMARKS: ACK is latched in the true state by a 0 to 1 transition of the operator acknowledge input. It remains true until reset by a read operation. It is usually desirable to do a dummy read to clear the ACK prior to the actual test for operator acknowledge.

Examples:

```
X = ACK             {Clear ACK}
WAIT UNTIL ACK      {Wait for operator input}
```

The first statement clears any operator acknowledge that may be left from an earlier time.

> **NOTE:** The seventh (n+6) block output of the BSEQ function block is designed to give the programmer a convenient means to reset the ACK pushbutton (typically configured as a remote control memory function block (function code 62)).

ACQUIRE

*Refer to **COMMON SEQUENCE**.*

ADVANCED PID

function block

PURPOSE: Advanced PID function blocks (APID, function code 156) provide a mechanism for the analog control of single input single output processes. The program interface to an APID function block provides information about a process and its controller and allows changes of the controller tuning parameters.

LANGUAGE:

BATCH 90	UDF

DECLARATION: **BLOCK** *name*, **BLK** = *const*, **FC** = **APID**

Parameter	Description
name	Name of the APID function block.
const	Function block address.

Example: BLOCK A501, BLK = 250, FC = APID

REFERENCES: APID data may be read by using the appropriate name in an expression.

Get process variable *name*.**PV**

Parameter	Description
name	Name of the APID function block.
name.**PV**	Process variable (specification S1).

Example: X = A501.PV

Get setpoint *name*.**SP**

Parameter	Description
name	Name of the APID function block.
name.**SP**	Set point (specification S2).

Example: X = A501.SP

Get control output *name*.**CO**

Parameter	Description
name	Name of the APID function block.
name.**CO**	Control output.

Example: X = A501.CO

ADVANCED PID *(continued)*

function block

Get track flag *name*.**TRK**

Parameter	Description
name	Name of the APID function block.
name.**TRK**	Track flag input (specification S4): 0 = tracking. 1 = released.

Example: X = A501.TRK

Get K tuning parameter (gain multiplier) *name*.**K**

Parameter	Description
name	Name of the APID function block.
name.**K**	K value (specification S11).

Example: X = A501.K

Get KP tuning parameter (proportional gain) *name*.**KP**

Parameter	Description
name	Name of the APID function block.
name.**KP**	KP value (specification S12).

Example: X = A501.KP

Get KI tuning parameter (integral reset) *name*.**KI**

Parameter	Description
name	Name of the APID function block.
name.**KI**	KI value (specification S13).

Example: X = A501.KI

Get KD tuning parameter (derivative rate) *name*.**KD**

Parameter	Description
name	Name of the APID function block.
name.**KD**	KD value (specification S14).

Example: X = A501.KD

Get KA tuning parameter (derivative log constant)

name.**KA**

Parameter	Description
name	Name of the APID function block.
name.**KA**	KA value (specification S15).

Example: X = A501.KA

Get algorithm type parameter

name.**TYP**

Parameter	Description
name	Name of the APID function block.
name.**TYP**	Algorithm type (specification S18): 0 = classical. 1 = non interacting. 2 = classical with external reset. 3 = manual reset non interacting.

Example: X = A501.TYP

Get setpoint modifier parameter

name.**SPM**

Parameter	Description
name	Name of the APID function block.
name.**SPM**	Set point modifier (specification S20): 0 = normal. 1 = integral only on setpoint change.

Example: X = A501.SPM

Get directional parameter

name.**DIR**

Parameter	Description
name	Name of the APID function block.
name.**DIR**	Direction (specification S21): 0 = reverse mode error (SP minus PV). 1 = direct mode error (PV minus SP).

Example: X = A501.DIR

ADVANCED PID *(continued)*

function block

Get high limit parameter *name*.**HLIM**

Parameter	Description
name	Name of the APID function block.
name.**HLIM**	High output limit (specification S16).

Example: X = A501.HLIM

Get low limit parameter *name*.**LLIM**

Parameter	Description
name	Name of the APID function block.
name.**LLIM**	Low output limit (specification S17).

Example: X = A501.LLIM

COMMANDS: The following statements are used to tune the function blocks.

Set K tuning parameter (gain multiplier) **SET** *name*.**K** **=** *exp*

Parameter	Description
name	Name of the APID block.
exp	**K** value.

Example: SET A501.K = 2.0

Set KP tuning parameter (proportional gain) **SET** *name*.**KP** **=** *exp*

Parameter	Description
name	Name of the APID function block.
exp	**KP** value.

Example: SET A501.KP = 1.0

Set KI tuning parameter (integral reset) **SET** *name*.**KI** **=** *exp*

Parameter	Description
name	Name of the APID function block.
exp	**KI** value.

Example: SET A501.KI = 3.0

ADVANCED PID *(continued)*

function block

Set KD tuning parameter (derivative rate)

SET *name***.KD = ** *exp*

Parameter	Description
name	Name of the APID function block.
exp	**KD** value.

Example: SET A501.KD = 0.3

#ALPHA

directive

PURPOSE:

The #ALPHA directive signals the compiler that the Batch 90 application contains alphanumeric identifiers for BATCH, CAMPAIGN, LOT, and RECIPE variables. These identifiers can be up to 16 characters in length. Valid alphanumeric characters are zero through nine, A through Z, and underscores.

LANGUAGE:

BATCH 90	UDF

FORMAT:

#ALPHA

Example:

```
#TITLE "EXAMPLE USING ALPHANUMERIC IDENTIFIERS"
#DEBUG LEVEL = 4
#DESCRIPTOR "ALPLA ID PROGR"
#ALPHA
```

REMARKS:

Usage of alphanumeric identifiers requires a change in function block configurations. REMSET function blocks, typically used as input blocks to BSEQ and BHIST function blocks, are replaced with DATAEXPT function blocks. Figure A-3 shows an example function block configuration.

Usage of at least one alphanumeric BATCH, CAMPAIGN, LOT, and RECIPE identifier within a project dictates that all BATCH, CAMPAIGN, LOT, and RECIPE identifiers within the project be alphanumeric. Numeric and alphanumeric identifiers can not be used in the same project.

Alphanumeric identifiers for BATCH, CAMPAIGN, LOT, and RECIPE require MFP module firmware revision F.2 or later.

ANALOG EXCEPTION REPORT

function block

PURPOSE: The analog exception report (AOL, function code 30) transmits an analog value and its attributes (quality and alarm) via the INFI-NET to operator consoles and other nodes.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Each AOL function block to be used by a batch program must be declared as follows:

NOTE: Input S1 of the AOL function block must be set to 2. Otherwise, batch program commands will be ignored.

BLOCK *name*, **BLK =** *const*, **FC = AOL** [**, WATCH**]

Parameter	Description
name	Name of the AOL function block.
const	Function block address.
WATCH (Batch 90 only)	If WATCH is specified, the initial AOL function block watch status is set to on by the Batch Historian on program start-up. If WATCH is not specified, the initial watch status is off.

Example: BLOCK AOL501, BLK = 240, FC = AOL

REFERENCES: AOL data may be read by using the appropriate name in an expression.

Get value *name*.**VAL**

Parameter	Description
name	Name of the AOL function block.
name.**VAL**	Current value.

Example: X AOL501.VAL

Get quality *name*.**Q**

Parameter	Description
name	Name of the AOL function block.
name.**Q**	Current quality: 0 = good. 1 = bad.

Example: X AOL501.Q

ANALOG EXCEPTION REPORT (continued)

function block

Get high alarm status *name*.**HAL**

Parameter	Description
name	Name of the AOL function block.
name.**HAL**	Current high alarm status: 0 = not high alarm. 1 = high alarm.

Example: X AOL501.HAL

Get low alarm status *name*.**LAL**

Parameter	Description
name	Name of the AOL function block.
name.**LAL**	Current low alarm status: 0 = not low alarm. 1 = low alarm.

Example: X AOL501.LAL

Get quality mask *name*.**QMSK**

Parameter	Description
name	Name of the AOL function block.
name.**QMSK**	Current quality mask: 0 = forced 0 (good). 1 = forced 1 (bad). 2 = not forced.

Example: X AOL501.QMSK

Get high alarm mask *name*.**HAMSK**

Parameter	Description
name	Name of the AOL function block.
name.**HAMSK**	Current high alarm mask: 0 = forced 0 (good). 1 = forced 1 (bad). 2 = not forced.

Example: X AOL501.HAMSK

ANALOG EXCEPTION REPORT *(continued)*

function block

Get low alarm mask *name*.**LAMSK**

Parameter	Description
name	Name of the AOL function block.
name.**LAMSK**	Current low alarm mask: 0 = forced 0 (not low). 1 = forced 1 (low). 2 = not forced.

Example: X AOL501.LAMSK

Get high alarm limit *name*.**HALIM**

Parameter	Description
name	Name of the AOL function block.
name.**HALIM**	Current high alarm limit (specification S5).

Example: X AOL501.HALIM

Get low alarm limit *name*.**LALIM**

Parameter	Description
name	Name of the AOL function block.
name.**LALIM**	Current low alarm limit (specification S6).

Example: X AOL501.LALIM

COMMANDS:

The following statements are used to control AOL function blocks:

Set value **SET** *name*.**VAL** **=** *exp*

Parameter	Description
name	Name of the AOL function block.
exp	Value.

Example: SET AOL501.VAL = 32.0

ANALOG EXCEPTION REPORT *(continued)*

function block

Set quality mask **SET** *name***.QMSK = *exp***

Parameter	Description
name	Name of the AOL function block.
exp	Quality mask: 0 = forced 0 (good). 1 = forced 1 (bad). 2 = not forced.

Examples:

SET AOL501.QMSK = 0
SET AOL501.QMSK = FORCED_OFF

Set high alarm mask **SET** *name***.HAMSK = *exp***

Parameter	Description
name	Name of the AOL function block.
exp	High alarm mask: 0 = forced 0 (no alarm). 1 = forced 1 (alarm). 2 = not forced.

Examples:

SET AOL501.HAMSK = 0
SET AOL501.HAMSK = FORCED_OFF

Set low alarm mask **SET** *name***.LAMSK = *exp***

Parameter	Description
name	Name of the AOL function block.
exp	Low alarm mask: 0 = forced 0 (no alarm). 1 = forced 1 (alarm). 2 = not forced.

Examples:

SET AOL501.LAMSK = 0
SET AOL501.LAMSK = FORCED_OFF

Set watch status (Batch 90 only) **SET** *name***.WATCH = *exp***

Parameter	Description
name	Name of the AOL function block.
*name***.WATCH**	Batch Historian watch status: 0 = off. 1 = on.

Example:

SET AOL501.WATCH = ON

ARRAY

*Refer to **BLOCK ARRAY**, **CONST ARRAY**, **CONST STRING ARRAY**, **VAR ARRAY**, and **VAR STRING ARRAY**.*

ASSIGNMENT

statement

PURPOSE: This statement assigns a value to a variable.

LANGUAGE:

BATCH 90	UDF

FORMAT: *var* = *exp*

Parameter	Description
var	Name of variable where value is to be stored.
exp	Expression to be evaluated.

Examples:

```
x = 123.0
x = y + z + 123.0
x = (FT101.VAL + FT102.VAL) / 2.0
STATUS = FT101.STS
```

REMARKS: Refer to ***VAR ARRAY*** for more information.

BATCH

variable

PURPOSE: BATCH is a built-in status variable.

LANGUAGE:

BATCH 90	UDF

DECLARATION: None required.

REFERENCES: BATCH ID is read by using BATCH in an expression.

Examples:

```
PRODUCT_NO = BATCH + 1
SUBPROD_ID_NO = BATCH
```

REMARKS: The Batch 90 language includes three built-in variables (CAMPAIGN, BATCH and LOT). These variables may be used by the Batch Historian function block for record keeping or by the common sequence function block for controlling resource allocation. Refer to ***CAMPAIGN*** and ***LOT*** for more information.

BATCH DATA

statement

PURPOSE:

This is the first statement of a batch data section within a batch program. This section defines global data, function codes, and symbols. All declarations in the batch data section are global since they can be referenced by all following phase, function, and monitor subroutines in the batch program.

There may be more than one batch data section in a program. The second batch data section is often used to declare global monitors. Since global monitor subroutines must be defined in the program before they are declared, the second batch data section should appear after the monitor subroutine definitions.

LANGUAGE:

BATCH 90	UDF

FORMAT:

BATCH DATA
type1 name1
type2 name2
•
•
•
END DATA

Parameter	Description
type	Any one of the following: **BLOCK** — Function block. *block* **ARRAY** — Array of function codes. **CONST** — Constant. **CONST ARRAY** — Constant array. **CONST STRING** — Constant string. **CONST STRING ARRAY** — Constant string array. **FGEN** — Function generator. **INTEGRATOR** — Integrator. **MONITOR** — Monitor call. **RAMP** — Ramp. **TIMER** — Timer. **VAR** — Variable. **VAR ARRAY** — Variable array. **VAR STRING** — Variable string. **VAR STRING ARRAY** — Variable string array.
name	Name of the data.

Example:

```
BATCH DATA
    VAR SUM                                       {variable}
    CONST NOT_OPEN = 0                            {constant}
    BLOCK DI02A, BLK = 100, FC = 84, \
    OUT (0) = FV501, OUT (1) = FV501              {function block}
    BLOCK FV501, BLK = 205, FC = DD               {device driver}
END DATA
```

REMARKS:

For function block declaration details, refer to:

AOL	Analog exception report
APID	Advanced PID
BBUF	Boolean data buffer
CS	Control station
CSEQ	Common sequence
DATAEXPT	User defined data export
DOL	Digital exception report
DD	Device driver
MSDD	Multi-state device driver
RBUF	Real data buffer
RCM	Remote control memory
REMSET	Remote manual set constant
RMC	Remote motor control
SMITH	Smith block
TEXT	Text

Other function codes may be referenced by their function code number.

Batch data is ***permanent***; the values remain from one phase subroutine to the next. Batch data is initialized when a batch program is run for the first time (that is, on the transition from batch complete to running).

The active data structures (for example, timers and monitor calls) execute independently of the active phase subroutine. They will execute even while the active phase subroutine is holding. Active data structures declared in a batch data section are sometimes referred to as being global. Their values are initialized to zero at the start of the batch sequence. They are never initialized again unless explicitly done so through a statement such as RESET.

BATCH SEQUENCE

function block

PURPOSE: The following reference extensions will retrieve the recipe, phase, campaign, batch, and lot ID and the execution status of a batch sequence function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Each BSEQ function block to be used by a batch program must be declared as follows:

NOTE: Input S1 of the AOL function block must be set to 2. Otherwise, batch program commands will be ignored.

BLOCK *bseqx*, **BLK = ***const*, **FC = BSEQ**

Parameter	Description
bseqx	Name of the BSEQ function block.
const	Function block address.

Example: BLOCK BSEQ501, BLK = 240, FC = BSEQ

REFERENCES: BSEQ data may be read by using the appropriate name in an expression.

Get phase ID *bseqx***.PHASE**

Parameter	Description
bseqx	Name of the BSEQ function block.
*bseqx***.PHASE**	Number of the phase subroutine being executed by the BSEQ function block declared by bseqx.

Example: X = BSEQ501.PHASE

Get execution status *bseqx***.XSTS**

Parameter	Description
bseqx	Name of the BSEQ function block.
*bseqx***.XSTS**	Execution status of the BSEQ function block declared by bseqx: -1 = batch inactive (complete). 0 = holding. 1 = running (normal logic active). 2 = hold (hold logic active). 3 = fault (fault logic active). 4 = restart (restart logic active).

Example: X = BSEQ501.XSTS

BBUF

Refer to ***DATA BUFFER****.*

BLOCK ARRAY

data structure

PURPOSE:

A block array is a group of previously defined function blocks of the same function code type, organized into an array. An array may have 1, 2, or 3 dimensions. The array as a whole has a name. An individual block (element) of an array is identified by an array index which specifies the position of the element within the array. An array index consists of subscripts - one subscript for each dimension.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

The following statements are used to declare (in data sections) block arrays.

{for one dimensional array}
type **ARRAY** *name* **(***low***:***high1***)** **(***list***)**

{for two dimensional array}
type **ARRAY** *name* **(***low***:***high1,* [*low***:**]*high2***)** **(***list***)**

{for three dimensional array}
type **ARRAY** *name* **(***low***:***high1,* [*low***:**]*high2,* [*low***:**]*high3***)** **(***list***)**

Parameter	Description
type	Any one of the following: **AOL** Analog output over the loop. **BBUF** Boolean buffer. **CS** Control station. **DATAEXPT** Data export. **DD** Device driver. **DOL** Digital output over the loop. **MSDD** Multi-state device driver. **RBUF** Real buffer. **RCM** Remote control memory. **REMSET** Remote manual set constant. **RMC** Remote motor control block. **TEXT** Text selector.
name	Name of the array.
low	Lowest subscript value must be 0 or 1. For second and third dimensional arrays, the lowest subscript is optional but must match the value from the first dimension.
highN	Highest subscript value for the Nth dimension. This must a constant. The total number of elements in an array must not exceed 16K.

BLOCK ARRAY *(continued)*

data structure

Parameter	Description
list	List of array element names. This 1 dimensional list has one entry for each element of the array. The order of entries is determined by varying the 1st (left-most) subscript the fastest, then the 2nd and 3rd.

Examples: In the following examples, the names included in the list correspond to names assigned to function block declarations in the data.

```
TEXT ARRAY TXT_MSGS (1:5) \
(LINE1, LINE2, LINE3, LINE4, LINE5)

CS ARRAY CONTROL_VLVS (1:3, 1:2) \
(FIC110, PIC118, TIC122, \
FIC207, PIC209, TIC225)

DD ARRAY REACTOR_FUNCTION_BLOCKS (1:4, 3, 2) \
(PV52, PV58, PV60, PV66, \
TV41, TV42, TV71, TV72, \
FV53, FV57, FV63, FV67, \
PV152, PV158, PV160, PV166, \
TV141, TV142, TV171, TV172, \
FV153, FV157, FV163, FV167)
```

REFERENCE:

Element of an array

{for one dimensional array}
name **(***exp1***)** *.ref ext*

{for two dimensional array}
name **(***exp1, exp2***)** *.ref ext*

{for three dimensional array}
name **(***exp1, exp2, exp3***)** *.ref ext*

Parameter	Description
name	Name of the array.
expn	Value of the nth subscript. Subscript boundary checking will be performed as is for all other array types.
.ref ext	Valid reference extension for the function block type of the array.

Examples:

```
F1_PV = CONTROL_VLVS (1, 1) .PV
F1_TGT = CONTROL_VLVS (1, 1) .SP
T2_CO = CONTROL_VLV (3, 2) .CO
SOV_MODE = REACTOR_FUNCTION_BLOCKS (4, 1, 2) .MODE
DISPLAY MESSAGE 1234 COLOR GREEN USING TXT_MSGS(I)
```

BLOCK ARRAY *(continued)*

data structure

Entire array

In many operations it is necessary to refer to an entire array as a whole. An entire array of function blocks will be referenced as follows:

name [*.ref ext*]

- or -

name **(***…**)** [*.ref ext*]

Parameter	Description
name	Name of the array.
(*…)	Number of dimensions (that is, (*), (*, *), (*, *, *)).
.ref ext	Valid reference extension for the function block type of the array.

The first form (just the name) is used when an array is passed as an argument to a function or monitor subroutine. The second form is used in other whole array operations.

Examples:

CALL CLEAR_MSGS (TXT_MSGS1)
TARGETS (*, *) = CONTROL_VLVS (*, *) .SP
PROCESS_VARIABLES (*, *) = CONTROL_VLVS (*, *) .PV

COMMANDS:

Assign (write a value to a block array element)

{for one dimensional array}
SET *name* **(***exp1***)** *.ref ext* **=** *exp*

{for two dimensional array}
SET *name* **(***exp1, exp2***)** *.ref ext* **=** *exp*

{for three dimensional array}
SET *name* **(***exp1, exp2, exp3***)** *.ref ext* **=** *exp*

Parameter	Description
name	Name of the array.
expN	Value of the Nth subscript. Subscript boundary checking will be performed as is for all other array types.
.ref ext	A valid reference extension for the function code block type of the array.
exp	Command value for the reference extension of the element.

data structure

Examples:

```
SET CONTROL_VLVS (1, 1) .SP = 0
SET REACTOR_FUNCTION_BLOCKS (1, 2, 1) .MODE = AUTO
SET INDICATORS (2, 2, 2) .VAL = 1
```

Entire array

Command references to block arrays may be made to the array as a whole. Thus, an entire array of function blocks is written to:

SET *name* (*...) *.ref ext* **=** *exp*

Parameter	Description
name	Name of the array.
(*...)	Number of dimensions.
.ref ext	A valid reference extension for the function block type of the array.
exp	Command value for the reference extension of the element.

Examples:

```
SET CONTROL_VLVS (*, *) .SP = 0
SET REACTOR_FUNCTION_BLOCKS (1, *, *).MODE = AUTO
SET INDICATORS (1, *, *) .VAL = 1.0
```

NOTE: Only function blocks explicitly defined in data sections may be included in an array (that is, no unit procedure dependent function blocks are allowed).

Elements within the array are static. Once defined in the list of the array declaration, the program is not able to dynamically change which function blocks are included in the array.

Subscript range checking is done by the compiler if all of the subscripts of an array are constants. Range checking is always performed during program execution. Each time an array element is read, the subscript values are checked. If a subscript value is out of bounds, some error action is taken. If the error occurs during any mode other than fault mode (for example, normal mode), the normal flow of control is aborted. The mode is changed to fault and control is transferred to fault logic with the system generated fault code of -17 indicating the error in array subscript boundary. If the error occurs during fault mode, flow of control is not aborted. A value of zero is returned for the incorrectly subscribed array element.

BREAK

statement

PURPOSE: This statement terminates the smallest enclosing loop structure (for example, REPEAT-UNTIL, WHILE-ENDWHILE, DO CASE or FOR LOOP).

LANGUAGE:

BATCH 90	UDF

FORMAT: **BREAK**

REMARKS: When this statement is executed, control passes to the statement following the terminated loop (for example, following an UNTIL, ENDWHILE, END CASE, or FOR LOOP).

Example:

```
REPEAT
    IF (TIMER1.VAL > 30.0) BREAK
    IF (TIMER2.VAL > 10.0) BREAK
UNTIL (A > 10)
```

The BREAK statement will not break out of an IF (...) THEN (...) ELSE (...) statement, because this IF statement is not classified as a looping structure.

CALL

Refer to ***FUNCTION****.*

CAMPAIGN

variable

PURPOSE: CAMPAIGN is a built-in status variable.

LANGUAGE:

BATCH 90	UDF

DECLARATION: None required

REFERENCES: CAMPAIGN ID is read by using CAMPAIGN in an expression.

Example:

```
TRAIN_NO = CAMPAIGN
PROD_ID_NO = CAMPAIGN + 1000
```

REMARKS: The Batch 90 language includes three built-in variables (BATCH, CAMPAIGN, and LOT). These variables may be used by the BSEQ function block for record keeping or by the CSEQ function block for controlling resource allocation. Refer to ***BATCH*** and ***LOT*** statements for more information.

CASE

statement

PURPOSE: This structure organizes statements into one or more cases (a block of statements with an identifier) and a switch to select one of the cases for execution.

LANGUAGE:

BATCH 90	UDF

FORMAT:

DO CASE *exp*
  CASE *id_list*
    statement 1a
    •
    •
    •
    statement 1n
  CASE *id_list2*
    statement 2a
    •
    •
    •
    statement 2n
  [*other*]
    statement
    statement
END CASE

Parameter	Description
exp	This value selects the case to be executed.
id_list	Each case is identified by its id_list. An id_list may be single number (for example, 3), a number range (for example, 5 TO 10), or a combination of numbers and ranges (for example, 3, 5 TO 10). All ranges are inclusive).
other	Identifies the case to be executed when none of the previous id_lists include the switch value.

NOTES:

1. The ”other” case specification must be the last case specified.

2. The switch value is rounded to the nearest integer value prior to evaluation.

3. The maximum number of cases that can be specified is 500.

CASE *(continued)*

statement

Example:

```
DO  CASE  X
    CASE  1
        SET  TIC220.SP  =  55
        SET  TIC221.SP  =  45
    CASE  4  TO  6,  8
        SET  TIC220.SP  =  45
        SET  TIC221.SP  =  55
    OTHER
        SET  TIC220.SP  =  50
        SET  TIC221.SP  =  50
END  CASE
```

REMARKS:

When the DO CASE statement is executed, the cases are searched in order (starting from the top) for the first id_list that contains the switch value. If a match is found, then the block of statements for that case is executed, and control then passes to the statement following the END CASE.

Id_lists may be overlapping (although the compiler will generate a warning message). The first case that matches the switch is the one executed. For example, the cases:

```
CASE 5
CASE 1 TO 10
```

are equivalent to the cases:

```
CASE 5
CASE 1 TO 4, 6 TO 10
```

Execution of a BREAK statement within a case passes control to the statement following the END CASE.

CASE statements may be nested.

#CASESENSE

directive

PURPOSE:

This compiler directive enables or disables case sensitivity for string comparison operations only.

LANGUAGE:

BATCH 90	UDF

FORMAT:

#CASESENSE = *n*

Parameter	Description
n	Case sense comparison options: 0 = all characters compared as upper case. 1 = characters compared as they exist.

REMARKS:

The default case sensitivity is disabled (set to zero). Once enabled, the sensitivity remains enabled until another directive is encountered. This directive may be included in any type of subroutine but must be the first characters of the first line of the source program. This directive does not effect write operations to the output string of export string function blocks.

CASE CONVERSION

command

PURPOSE: This command converts the case of characters within a variable string. The result of the conversion is stored in the original string variable.

LANGUAGE:

BATCH 90	UDF

FORMAT:

TO UPPER (*vstrname***)**

- or -

TO LOWER (*vstrname***)**

Parameter	Description
vstrname	Name of VAR STRING data only. **NOTE:** Variable strings associated with data export and CSEQ function blocks can not be directly effected by TO UPPER or TO LOWER commands.

Examples:

```
TO UPPER (PROMPT_MSG)          {converts to upper case}
TO LOWER (REQUIRED_ACTION)     {converts to lower case}
```

COMMON SEQUENCE

data structure

PURPOSE:

The common sequence (CSEQ) function block provides the communication interface with batch sequence (BSEQ) function blocks. This interface is established when one batch sequence function block (the client program) connects to a second batch sequence function block (the server program) via a CSEQ function block attached to the server BSEQ function block.

Once the client program has established the connection, the server program sends the following status information about the server to the client:

- Connection status.
- Reservation status.
- Ownership status.
- Control mode (operator, remote).
- Program execution status (inactive, running, holding).
- Unit recipe ID.
- Operation number.
- Fault code.
- Program ID number.
- Campaign, batch and lot identifiers.
- Status variables (.SV1 through .SV8).
- Status strings (.STSSTRG).

This information is available to all connected client programs by references to the server programs CSEQ name as identified in the CSEQ function block declaration.

For a client program to gain control of a server program, the client program must acquire ownership of the server. The client program that has acquired ownership of the server may perform the following operations on the server via commands to the CSEQ function block.

- Select the unit recipe, operation and start/restart for the program.
- Select the program to execute.
- Put the program into hold logic.
- Send command variables to the CSEQ function block (.CV1 through .CV8).
- Send command strings (.CMDSTRG) to the CSEQ function block.
- Unacquire (release ownership of) the CSEQ function block.

COMMON SEQUENCE (continued)

data structure

The client server relationship between programs, with the client capable of controlling the server programs execution, is realized with the use of the CSEQ function block. One program may be the owner (client) of a second program or server, as well as a server to a third program. Thus, production trains contain batch units linked via CSEQ function blocks. Refer to Figure 4-1 for an example production train.

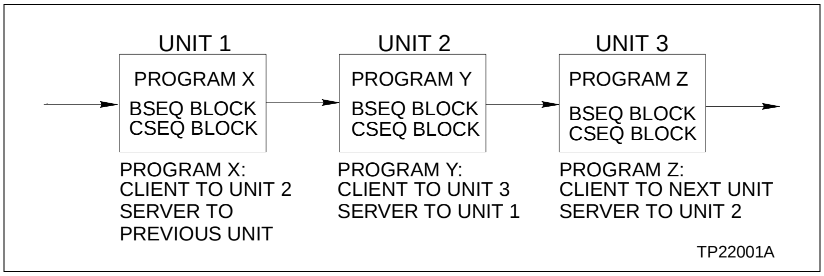


Figure 4-1. Example Production Train

LANGUAGE:

BATCH 90	UDF

DECLARATION:

A CSEQ function block must be declared in a batch data, unit data, or unit parameter section of the program before it can be accessed from a subroutine. The CSEQ reserve allow, ownership allow, and control mode inputs affect what access is given to connected client programs. Reserve allow and ownership allow inputs determine whether the server is eligible to grant reserve and acquire requests. The control mode input determines the source of program execution control for the server. In operator mode, the unit recipe, operation, and run/hold inputs from the BSEQ function block control the program execution. In operator mode, reservation and ownership requests may be granted. However, control of program execution remains in the operators control until the control mode is changed to remote. In remote mode, the owner (a client program that has acquired ownership) controls program execution.

data structure

Declaration in batch data

BLOCK *cseqname*, **BLK** = *const1*, **FC** = **CSEQ** \
[**RING** = *const2*,] [**PCU** = *const3*,] [**MODULE** = *const4*,] \
[**TPR** = *const5*,] [**PROGRAM** = *const6*]

Parameter	Description
cseqname	Name of the CSEQ function block.
const1	Block address of the BSEQ function block attached to the CSEQ function block.
const2	Loop address (0 = this loop).
const3	Process control unit address (0 = this process control unit).
const4	Module address (0 = this module).
const5	Target partial unit recipe number.
const6	Target program ID number.

NOTE: If not specified by the declaration, the default values for loop, PCU, and module address will be assigned the value of zero.

REFERENCES:

When the following references are operative from the client programs side of the connection, *cseqname* is the name assigned to the CSEQ data structure in the clients program that represents the server. Other references are operative from the server programs side of the connection. In these cases, *cseqname* is THIS (that is, THIS refers to itself).

Get connection status

cseqname.**CSTS**

- or -

THIS.CSTS

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**CSTS**	Connection status: -1 = failed 0 = not connected 1 = connected ≥2 = trying to connect.
THIS.CSTS	Number of active connections currently being supported by the CSEQ function block.

Examples:

```
WAIT WHILE (FEED_TANK.CSTS = WAITING)
IF (FEED_TANK.CSTS = FAILED) FAULT1
IF (THIS.CSTS < 1) NEXT OPERATION 1
WAIT UNTIL (THIS.CSTS > 1)
```

COMMON SEQUENCE *(continued)*

data structure

Get reserved status *cseqname*.**RSTS**

- or -

THIS.RSTS

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**RSTS**	Reserved status: 1 = reservation not allowed. 0 = not reserved. 1 = reserved (production ID qualify). 2 = reserved (production ID does not qualify).
THIS.RSTS	Reserved status: -1 = reservation not allowed. 0 = not reserved. 1 = reserved.

Examples:

IF (DELIVERY_HEADER.RSTS IS UNRESOLVED) THEN...
IF (DELIVERY_HEADER.RSTS IS ANOTHER) THEN...
IF (THIS.RSTS IS RESERVED) THEN...
IF (THIS.RSTS IS NOT_ALLOWED) THEN...

Get ownership status *cseqname*.**OSTS**

- or -

THIS.OSTS

Parameter	Description
cseqname	Name of CSEQ function block.
cseqname.**OSTS**	Ownership status: -1 = ownership not allowed. 0 = not owned. 1 = owned by me. 2 = owned by another.
THIS.OSTS	Ownership status: -1 = ownership not allowed. 0 = not owned. 1 = owned.

Examples:

IF (PREP_TANK.OSTS = ANOTHER THEN...
WAIT UNTIL (THIS.OSTS IS OWNED)
IF (THIS.OSTS IS NOT_ALLOWED) THEN...
WAIT UNTIL (THIS.OSTS IS OWNED)

COMMON SEQUENCE *(continued)*

data structure

Get ownership queue status from a client program

cseqname.**QSTS**

- or -

THIS.QSTS

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**QSTS**	Ownership queue status: -1 = ownership queue off and erased. 0 = ownership queue off. 1 = ownership queue on.
THIS.QSTS	Ownership queue status: -1 = ownership queue off and erased. 0 = ownership queue off. 1 = ownership queue on.

Examples:

```
WAIT UNTIL (HEADER.QSTS IS ON)
IF (HEADER.QSTS IS OFF) UNCONNECT HEADER
REPEAT ... UNTIL (THIS.QSTS IS ON)
IF (THIS.QSTS = FLUSHED) SET FLAGS.VAL = OFF
```

Get control mode

cseqname.**MODE**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**MODE**	Control mode: 0 = operator (manual). 1 = remote (auto).

REMARKS:

In the operator mode, the server program execution is controlled normally. In the remote mode, inputs to the BSEQ function block are disabled. Control of these items is passed to the CSEQ function block which responds to commands received from the client program that owns the server. The emergency stop and acknowledge inputs are active in both modes.

Examples:

```
IF (TANK_1.MODE IS OPERATOR) CONNECT TANK_2
WAIT UNTIL (COOLING_SYS.MODE IS REMOTE)
```

COMMON SEQUENCE *(continued)*

data structure

Get program execute status

cseqname.**XSTS**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**XSTS**	Program status: -1 = batch inactive (complete). 0 = holding. 1 = running (normal logic active). 2 = hold (hold logic active). 3 = fault (fault logic active). 4 = restart (restart logic active).

Examples:

```
IF (REACTOR.XSTS <> INACTIVE) THEN
WAIT UNTIL (REACTOR.XSTS IS RUNNING)
```

Get fault code

cseqname.**FAULT**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**FAULT**	Fault code.

Examples:

```
IF (HEADER.FAULT > 0) SET XFER_ALARM.VAL = ON
IF (FEED_TANK.FAULT = -10) RECIPE_ERROR = ON
```

Get operation number

cseqname.**OPERATION**

{ISA-S88.01-1995compliantterminology}

- or -

cseqname.**PHASE** {Elsag Bailey traditional terminology}

Parameter	Description
cseqname. **OPERATION**	Operation number.
cseqname	Name of the CSEQ function block.
cseqname.**PHASE**	Phase number.

Examples:

```
WAIT UNTIL (REACTOR.OPERATION > 1)
IF (TANK.OPERATION < 3) THEN
```

REMARKS:

The data type for the BATCH, CAMPAIGN, LOT, and RECIPE identifiers depends on the identifier mode. Default mode is numeric. Using the #ALPHA directive changes the numeric identifiers to 16 character alphanumeric character strings.

Get campaign indentifier

cseqname.**CAMPAIGN**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**CAMPAIGN**	Campaign identifier.

Examples:

IF (STRG_TANK.CAMPAIGN <> CAMPAIGN) THEN...
IF (HEADER.CAMPAIGN > 0) THEN...

Get batch identifier

cseqname.**BATCH**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**BATCH**	Batch identifier.

Examples:

IF (STRG_TANK.BATCH <> BATCH) BNUM_MATCH = OFF
IF (REACTOR.BATCH <> STRG_TANK.BATCH) THEN...

Get lot identifier

cseqname.**LOT**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**LOT**	Lot identifier.

Examples:

IF (STRG_TANK.LOT <> LOG) LNUM_MATCH = OFF
IF (HEADER.LOT < TANK.LOT) THEN...

Get unit recipe identifier

cseqname.**RECIPE**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**RECIPE**	Unit recipe identifier.

REMARKS:

The unit procedure number is the number of the unit procedure that is currently being run by the server program.

Examples:

IF (REACTOR.RECIPE <> REACTOR_UNIT_PROCEDURE_NO) THEN
IF (PREP_TANK.RECIPE <> INGRED_UNIT_PROCEDURE_NO) THEN

Get target partial unit recipe number

cseqname.**TPR**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**TPR**	Unit recipe identifier.

COMMON SEQUENCE (continued)

data structure

REMARKS: The target partial unit recipe identifier is the unit recipe identifier supplied by the owners unit recipe or in the declaration, along with the CSEQ function block location. The primary use for this value is to select the unit recipe in the start command.

Examples:

```
IF (PREP_TANK.TPR = PREP_TANK.RECIPE) THEN...
IF (REACTOR.TPR = <> PRODUCT_ID_NO) THEN...
```

Get program ID number *cseqname*.**PROGRAM**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**PROGRAM**	Program ID number.

REMARKS: The program ID number is the ID number of the program that is currently being run by the server program.

Examples:

```
IF (DIGESTER.PROGRAM <> VAPOR_PHASE) THEN...
IF (PREP_TANK.PROGRAM <> INGRED_TYPE) THEN...
```

Get target program ID number *cseqname*.**TPID**

Parameter	Description
cseqname	Name of the CSEQ block.
cseqname.**TPID**	Target program ID number.

REMARKS: The *cseqname*.**TPID** is the program ID number supplied by the owners unit procedure or in the declaration, along with the CSEQ function blocks location. The primary use for this value is to select the program to be executed by the CSEQ function block in the start command.

Examples:

```
IF (DIGESTER.PROGRAM <> DIGESTER.TPID) THEN...
IF (PREP_TANK.TPID <> PREP_TANK.PROGRAM) THEN...
```

Get status variable *cseqname*.**SV***n*

- or -

THIS.SV*n*

***COMMON SEQUENCE** (continued)*

data structure

Parameter	Description
cseqname	Name of the CSEQ function block.
n	Number of status variable (1 - 8).
*cseqname***.SV***n*	Value of status variable *n*.
THIS.SV*n*	Value of status variable *n*.

REMARKS: Status variables provide a convenient way for a server program to send status data to its owner and other client programs. The server program writes to a status variable using the reference to itself (THIS). A client program reads a status variable using the name of the server defined in the declaration of the CSEQ function block.

Examples:

COOLING_STS = COOLING_SYS.SV2
BLOW_TIIME = DIGESTER.SV8
PARTIAL.STEP_NO = THIS.SV5

Get status string variable

*cseqname***.STATSTRG**

- or -

THIS.STATSTRG

Parameter	Description
cseqname	Name of the CSEQ function block.
*cseqname***.** **STATSTRG**	Value of the status string variable.
THIS.STATSTRG	Value of the status string variable.

REMARKS: Status string variables provide a convenient way for a server program to send alphanumeric data to its owner and other client programs. The server program writes to the status string variable using the reference to itself (THIS). A client program reads the status string variable using the name of the server defined in the declaration of the CSEQ function block.

Example:

REACTOR_STATE = REACTOR_EXE.STATSTRG
STRING_VAR = FEED_TANK.STATSTRG

COMMON SEQUENCE (continued)

data structure

Get command variable

cseqname.**CV***n*

- or -

THIS.CV*n*

Parameter	Description
cseqname	Name of the CSEQ function block.
n	Number of command variable (1 - 8).
cseqname.**CV***n*	Target value of command variable *n*.
THIS.CV*n*	Target value of command variable *n*.

REMARKS:

The actual value of a server command variable is equal to the target value of the owner. If the client is not the owner, its target values are UNDEFINED.

Examples (client):

FIQ1025.VAL = COOLING_SYS.CV6
TGT_BLOW_TIME = DIGESTER.CV4

Examples (server):

SEQ_STEP_CMD = THIS.CV3
SET STEP_NO_IND.VAL = THIS.CV7

Get command string variable

cseqname.**CMDSTRG**

- or -

THIS.CMDSTRG

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname. **CMDSTRG**	Target value of the command string variable.
THIS.CMDSTRG	Actual value of the command string variable.

REMARKS:

The actual value of a server command variable is equal to the target value of the owner. If the client is not the owner, its target values are UNDEFINED.

Example (client):

RX_STRING = REACTOR_EXEC.CMDSTRG
STRINGX = FEED_TANK.CMDSTGR

Example (server):

OP_MSG = THIS.CMDSTRG
STRINGY = THIS.CMDSTRG

data structure

Get command pending status

cseqname.**CPEND**

Parameter	Description
cseqname	Name of the CSEQ function block.
cseqname.**CPEND**	Value of command pending status: 0 = not waiting (good). 1 = waiting.

REMARKS:

This form of the reference is operative only from a client program. This reference is intended to give the owner program a means to verify that CSEQ commands have actually been received by the server programs CSEQ function block. The CSEQ function code will not guarantee the synchronous receipt of the sequential CSEQ commands.

Example:

```
WAIT WHILE (FEED_TANK.CPEND IS WAITING)
WAIT UNTIL (COOLING_SYS.CPEND <> WAITING)
```

COMMANDS:

Common sequence commands are executed in a manner similar to function block commands. Commands do not communicate directly with the CSEQ function block. The CSEQ command is buffered in the CSEQ data structure and the program goes on to the next statement. When the client program reaches a wait state, the command is sent from the CSEQ data structure to the servers CSEQ function block. There are important consequences to this approach:

1. CSEQ commands are allowed in all program logic sections (that is, normal, continuous, fault, etc.).

2. There is a time delay between the issuance of a command by the client and action taken by the server.

3. If a command is pending (a command statement has been issued but the command has not been acted upon) and a counteracting command is executed, the second command may overwrite the first. For example, if a hold command is followed too quickly by a start command, the CSEQ function block will see only the start command. As another example, if a command variable update is followed too quickly by another update of the same command variable, the first value is lost. In some cases, it may be necessary for the program to control the order in which commands take affect. This can be accomplished by checking the appropriate status reference between commands.

COMMON SEQUENCE *(continued)*

data structure

Example:

```
HOLD FILL_TANK
WAIT UNTIL (FILL_TANK.XSTS = WAITING)
START FILL_TANK
WAIT UNTIL (FILL_TANK.XSTS = RUNNING)
```

Connect to a CSEQ

CONNECT *cseqname*

Parameter	Description
cseqname	Name of the CSEQ function block.

Examples:

```
CONNECT FEED_TANK
CONNECT REACTOR
```

REMARKS:

This command causes a client program to establish a connection with the specified server CSEQ function block.

If a connection cannot be established or if an established connection fails, the connection status indicates failed. A program should monitor all of its connections and take appropriate action in case of a failure.

Unconnect from a CSEQ

UNCONNECT *cseqname*

Parameter	Description
cseqname	Name of the CSEQ function block.

Example:

```
UNCONNECT REACTOR
UNCONNECT FEED_TANK
```

REMARKS:

This command causes the client program to unconnect from the specified server CSEQ function block.

NOTES: A connected client program will be unconnected from a server CSEQ function block if the following occurs:

1. The connected program terminates or its module leaves execute mode.

2. The module containing the CSEQ function block leaves execute mode.

3. The communication system fails.

When a program becomes unconnected, all pending commands are lost.

data structure

Acquire ownership of a CSEQ

ACQUIRE *cseqname* **CAMPAIGN** = *exp1*, **BATCH** = *exp2*,****
LOT = *exp3*, [**PRIORITY** = *exp4*] [**OPTIONS** = *exp5*]

Parameter	Description
cseqname	Name of the CSEQ function block.
exp1 through *exp3*	Campaign, batch and lot identifiers to be passed through to the CSEQ function block. By use of the reserved words MINE and YOURS, the client program may specify the CSEQ function block to use values associated with the client program (MINE) or the server program (YOURS).
exp4	Optional request priority level (1 through 127). One is the highest priority and 127 is the lowest. If not specified, the request is given a priority of 127. A priority value of -1 specifies a one time request (not queued).
exp5	Specifies the execution status of the CSEQ function block that is required before ownership is granted. 0 = server program status must be complete. 1 = server program status may be complete, running or holding.

REMARKS:

When a CSEQ function block receives an acquire request and the CSEQ function block is eligible to be acquired and the request is qualified, it will grant the request. If the CSEQ function block is not eligible or the request is not qualified, the request will either queued or discarded (one time).

A common sequence is eligible to be acquired if:

1. The CSEQ function block is not owned.

   - and -

2. The ownership allow input of the CSEQ function block is 1.0 or 2.0.

   - and -

3. The CSEQ function block is inactive, OPTIONS = NORMAL.

   - or -

   The CSEQ function block is in any state, OPTIONS = ALLOWED.

COMMON SEQUENCE (continued)

data structure

An acquire request is qualified if:

1. The CSEQ function block is not reserved.

   - or -

2. The CSEQ function block is reserved.

   - and -

   The production ID specified in the ACQUIRE command by the requestor matches the reservation.

   NOTE: When a CSEQ function block is acquired, it receives production ID information (BATCH, CAMPAIGN, and LOT identifiers) from the owner. The production ID identifiers do not become active until the CSEQ function block performs an inactive to run transition. For example, the owner starts the CSEQ function block from the batch complete state.

Example:

```
ACQUIRE FEED_TANK CAMPAIGN = MINE, \
BATCH = YOURS, LOT = 1, PRIORITY = 1
WAIT UNTIL (FEED_TANK.OSTS = OWNED)
```

Unacquire a CSEQ

UNACQUIRE *cseqname*

Parameter	Description
cseqname	Name of the CSEQ or THIS function block.

REMARKS:

The purpose of this command is to counteract an earlier acquire command.

If the acquire command was queued, the unacquire command cancels it. In addition to the unacquire command, a queued acquire command is canceled if the requesting program terminates or stops communicating.

If the acquire command was granted, the unacquire command (issued by the owner) terminates ownership. The CSEQ function block becomes unowned and eligible for acquire in the following ways:

NOTES: Unacquire of a CSEQ function block can also happen in the following ways.
1. The ownership-allow input of the CSEQ function block changes to 0.0 or -1.0.

2. The owner program terminates or stops communicating.

COMMON SEQUENCE *(continued)*

data structure

Example: UNACQUIRE FEED_TANK
UNACQUIRE REACTOR

Reserve a CSEQ

RESERVE *cseqname* [**CAMPAIGN** = *exp1*,] [**BATCH** = *exp2*,]****
[**LOT** = *exp3*,] [**MAINTAIN** = *exp4*]

Parameter	Description
cseqname	Name of the CSEQ or THIS function block.
exp1 through *exp3*	Campaign, batch and lot identifiers to be passed through to the CSEQ function block. By use of the reserved words MINE and YOURS, the client program may specify the CSEQ function block to be reserved use values associated with the client program (MINE) or the CSEQ function block (YOURS).
exp4	Reservation status: 0 = reservation is cancelled when ownership is granted. 1 = reservation is maintained when ownership is granted.

REMARKS:

The purpose of this command is to restrict future ownership of the CSEQ function block. The requesting program specifies which parameters (BATCH, CAMPAIGN, and LOT) are to be included in the reservation. If campaign, batch or lot is not specified in the reserve command then the omitted parameter is considered as a don't care.

When a CSEQ function block receives a reserve request and conditions permit, it will grant the request. Otherwise, it will discard the request. The following conditions must be true for a reserve request to be granted:

CSEQ function block is not reserved.

- and -

CSEQ function block reserve allow input is 1.0.

The data type for the BATCH, CAMPAIGN, and LOT identifiers depends on the identifier mode. Default mode is numeric. Using the #ALPHA directive changes the numeric identifiers to 16 character alphanumeric character strings.

Examples:

```
RESERVE REACTOR CAMPAIGN = MINE, BATCH = MINE, \
LOT = YOURS
RESERVE FEEDTANK BATCH = 456
```

COMMON SEQUENCE *(continued)*

data structure

Unreserve a CSEQ **UNRESERVE** *cseqname*

Parameter	Description
cseqname	Name of the CSEQ or THIS function block.

REMARKS:

The purpose of this command is to unreserve a CSEQ function block. The command is operative only if issued by the CSEQ function block or its owner.

In addition to this command, a CSEQ function block can become unreserved in the following ways:

1. The reserve allow input of the CSEQ function block becomes 0.0.

2. The CSEQ program becomes owned by a qualified program and the reservation was not specified as MAINTAIN.

Example UNRESERVE REACTOR

Start/restart a CSEQ **START** *cseqname* [**RECIPE** = *exp1*] [**OPERATION** = *exp2*]****
[**PROGRAM** = *exp3*] {ISA-S88.01-1995 compliant terminology}

- or -

START *cseqname* [**RECIPE** = *exp1*] [**PHASE** = *exp2*]****
[**PROGRAM** = *exp3*] {Elsag Bailey traditional terminology}

Parameter	Description
cseqname	Name of the CSEQ function block.
exp1	Unit recipe identifier.
exp2	Operation number.
exp3	Program ID number.

REMARKS:

This command starts or restarts a CSEQ function block. The CSEQ function block starts if the current status is complete and restarts if the current status is holding. The command is ignored if the CSEQ function block is already running.

The unit procedure, operation, and program ID specifications in this command are optional. If not specified, the CSEQ function block will use its current value for the unspecified item.

This command is operative only if issued by the owner of the CSEQ function block. This command is disabled while the CSEQ function block is in the operator mode.

data structure

The data type for the RECIPE identifier depends on the identifier mode. Default mode is numeric. Using the #ALPHA directive changes the numeric identifiers to 16 character alphanumeric character strings.

Example:

```
START REACTOR RECIPE = “PROD_123”, OPERATION = 1
START FEEDTNK RECIPE = 123
```

Hold a CSEQ

HOLD *cseqname*

Parameter	Description
cseqname	Name of the CSEQ function block.

REMARKS:

This command requests a CSEQ function block to switch into HOLD LOGIC.

The command is ignored if:

1. The CSEQ function block is in operator mode.

   - or -

2. The CSEQ function block is holding or complete.

   - or -

3. The CSEQ function block is in hold or fault logic.

   - or -

4. External hold is disabled.

This command is operative only if issued by the owner of a CSEQ function block.

Example:

```
HOLD REACTOR
HOLD FEEDTNK
```

COMMON SEQUENCE *(continued)*

data structure

COMMANDS:

Set a command variable

SET *cseqname***.CV***n* **=** *exp*

- or -

SET THIS.CV*n* **=** *exp*

Parameter	Description
cseqname	Name of the CSEQ function block.
n	Number of command variable (1 through 8).
exp	Value.

REMARKS:

The command variable provides a convenient way for an owner of a CSEQ function block to send numeric data to the server. The owner program writes to the command variables using the name of the CSEQ function block. The server program writes to the command variables using the name THIS only when the server is not owned.

This command is operative from the client program only if it is the owner.

This command is operative from the server program only when the server is not owned.

Examples:

SET FILLTANK.CV2 = 3
SET THIS.CV2 = X

Set a command string variable

SET *cseqname***.CMDSTRG** **=** *exp*

- or -

SET THIS.CMDSTRG **=** *exp*

Parameter	Description
cseqname	Name of the CSEQ function block.
exp	Value.

REMARKS:

The command string variable provides a convenient way for an owner of a CSEQ function block to send alphanumeric data to the server. The owner program writes to the command string variables using the name of the CSEQ function block. The server program writes to the command string variables using the name THIS only when the server is not owned.

data structure

This command is operative from the client program only if it is the owner.

This command is operative from the server program only when the server is not owned.

Examples:

SET REACTOR_EXEC.CMDSTRG = "REACTOR READY"
SET THIS.CMDSTRG = STRING_VAR

Set a status variable

SET THIS.SV*n* **=** *exp*

Parameter	Description
n	Number of status variable (1 through 8).
exp	Value.

REMARKS:

The status variables provide a convenient way for a CSEQ function block to send numeric data to its owner and other connected programs. The server program writes to a status variable using the name THIS. The owner or other connected program reads a status variable using the name of the CSEQ function block. This command is operative only if issued by the server.

Example:

SET THIS.SV2 = 3
SET THIS.SV7 - THIS.BATCH

Set a CSEQ status string variable

SET THIS.STATSTRG = *exp*

Parameter	Description
exp	Value.

REMARKS:

The status string variables provide a convenient way for a CSEQ function block to send alphanumeric data to its owner and other connected programs. The CSEQ function block writes to a status string variable using the name THIS. The owner or other connected program reads a status string variable using the name of the CSEQ function block. This command is operative only if issued by the server.

Example:

SET THIS.STATSTRG = "CHARGE ABORTED"
SET THIS.STATSTRG = STRING_VAR

CONNECT

Refer to ***COMMON SEQUENCE****.*

CONST

data

PURPOSE: To define a constant.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The following declaration is used to define a constant.

CONST *name* **=** *const*

Parameter	Description
name	Name of the constant.
const	Value of the constant.

Example: CONST MAX_LNT = 10.0

REMARKS: Refer to ***BATCH DATA*** and ***DECLARATIONS***.

REFERENCES: The value of a constant is read by using its name in an expression or a definition.

Example: IF (X > MAX_LNT) THEN

CONST ARRAY

data structure

PURPOSE:

A CONST ARRAY is a group of constants organized in a table. An array may have 1, 2 or 3 dimensions. The array as a whole will have a name. An individual constant (element) of an array is identified by an array index which specifies the position of the element within the array. An array index consists of subscripts - one subscript for each dimension.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

The following statements are used to declare CONST ARRAYS.

{for one dimensional array}
CONST ARRAY *name* **(***low***:***high1***)** **(***list***)**

- or -

CONST ARRAY *name* **(***low***:*******)** **(***list***)**

{for two dimensional array}
CONST ARRAY *name* **(***low***:***high1, low***:***high2***)** **(***list***)**

{for three dimensional array}
CONST ARRAY *name* **(***low***:***high1, low***:***high2, low***:***high3***)** **(***list***)**

Parameter	Description
name	Name of the array.
low	Lowest subscript value (must be 0 or 1).
highN	Highest subscript value for the Nth dimension. Must be a constant. The total number of elements in an array must not exceed 16K.
list	List of array element names. This one-dimensional list has one entry for each element of the array. The order of the entries is determined by varying the first (leftmost) subscript the fastest, then the second and third.

NOTE: For two and three dimensional arrays the lowest subscript value for the second and third dimension must match the value that was specified by the first dimension.

Examples:

In the following examples, the values in the constant list correspond to the respective subscript values.

```
CONST ARRAY C (1:5) (1, 2, 3, 4, 5)
CONST ARRAY CZ (0:5) (0, 1, 2, 3, 4, 5)
CONST ARRAY CC (1:3, 1:2) (11, 21, 31, 12, 22, 32)
```

CONST ARRAY *(continued)*

data structure

```
CONST ARRAY CCC (1:2, 1:2, 1:2) (111, 211, 121, 221, \
112, 212, 122, 222)
```

REFERENCE:

Array element

{for one dimensional array}
name **(***exp1***)**

{for two dimensional array}
name **(***exp1*, *exp2***)**

{for three dimensional array}
name **(***exp1*, *exp2*, *exp3***)**

Parameter	Description
name	Name of the array.
expn	Value of the nth subscript. If a subscript value exceeds its specified range, it causes an error.

Examples:

```
A = C (5)
A = CC (i, j)
A = CCC (1, 2, i)
CALL MAX (CC (i, j), X)
```

Entire array

In some operations it is necessary to refer to an entire array as a whole. An entire array is referenced as follows:

name

- or -

name (*...)

Parameter	Description
name	Name of the array.
(*...)	Indicates the number of dimensions, (that is (*),(*, *) or (*, *, *)).

The first form (just the name) is used when an array is passed as an argument to a function or monitor subroutine. The second form is used in other whole array operations (that is, array copy).

Examples:

```
CALL MAX_ELEMENT (C)
XB (*) = C (*)
```

CONST ARRAY *(continued)*

data structure

COMMANDS: Since constant arrays are ***read only***, there are no commands to write to them.

> **NOTE:** Subscript range checking is done by the compiler if all of the subscripts of an array are constant. Range checking is always performed during program execution. Each time an array element is read, the subscript values are checked. If a subscript value is out of bounds, some error action is taken. If the error occurs during any mode other than FAULT mode (for example, NORMAL mode), the normal flow of control is aborted. The mode is changed to fault and control is transferred to fault logic with the system generated fault code of -17. If the error occurs during FAULT mode, flow of control is not aborted. A value of zero is returned for the incorrectly subscribed array element.

CONST STRING

data

PURPOSE: Constant strings can be declared in the BATCH DATA section, UNIT PARAMETERS section, or any place in the phase, function, or monitor subroutines. Constant strings can also be declared in the UNIT DATA section of a program and then declared again (given a value) in the unit definition file".

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from the BATCH DATA section of a Batch 90 program}
CONST STRING *cstrname* **=** "*string value*"

{from the UNIT DATA section of a Batch 90 program}
CONST STRING *cstrname* [*(cons)*]

{from the UNIT DATA section of a unit definition file}
CONST STRING *name* **=** "*value*"

{explicit constant string}
CONST STRING *name* **=** "*string_exp*"

Parameter	Description
cstrname	Name of the constant string.
string value	Value of the constant string.
cons	Maximum number of characters that may be included in the string value specification in the unit definition file. The default value is 80 characters.
name	The same name used in the UNIT DATA section.
value	Value of the constant string. This value can not contain more characters than were specified in the UNIT DATA section of the batch program.
string_exp	Explicit constant string used within a string command or reference.

Examples:

```
CONST STRING CLASS_NAME = "GLASS_LINED_REACTORS"
CONST STRING UNIT_NAME = "REACTOR - R552"
IF (VAR_STR = "SKIP") THEN...
```

CONST STRING *(continued)*

data

REFERENCES:

Get current string length *cstrname***.CLEN**

Parameter	Description
cstrname	Name of the constant string.
*cstrname***.CLEN**	The number of characters included in the *string value* parameter.

Examples:

```
SOME_VAR = CLASS_NAME.CLEN
OTHER_VAR = UNIT_NAME.CLEN
```

REMARKS: The .CLEN reference returns numerically the maximum and current number of characters contained within the referenced constant string. For constant strings, these numbers will be identical with the possible exception of constant strings declared within UNIT DATA sections. For constant strings declared within UNIT DATA sections, this reference will return the number of characters that were included in the specified string value from the unit definition file. In the event the specified string value is shorter than the maximum specified in the batch program, the constant string will not be back filled with spaces.

Get maximum string length *cstrname***.MLEN**

Parameter	Description
cstrname	Name of the constant string.
*cstrname***.MLEN**	The number of characters included in the *string value* parameter.

Examples:

```
SOME_VAR = CLASS_NAME.MLEN
OTHER_VAR = UNIT_NAME.MLEN
```

REMARKS: The .MLEN reference returns numerically the maximum and current number of characters contained within the referenced constant string. For constant strings, these numbers will be identical with the possible exception of constant strings declared within UNIT DATA sections. For constant strings declared in UNIT DATA sections, this reference will return the maximum number of characters that were specified by the declaration in the batch program.

CONST STRING *(continued)*

data

Get entire string value *cstrname*

Parameter	Description
cstrname	Name of the constant string.

Examples:

SOME_VAR_STRING = CLASS_NAME
OTHER_VAR_STRING = UNIT_NAME

REMARKS:

IF the referenced constant string is longer than the variable string, the first *x* (maximum number of characters for string variable) characters of the constant string shall be stored in the variable string.

Get value of a substring *cstrname* **(FROM** *exp1*[**FOR** *exp2*]**)**

Parameter	Description
cstrname	Name of the constant string.
exp1	First character or starting point of the substring.
exp2	Number of characters to be included in the substring. If not specified, the substring is terminated at the end of *cstrname*.

Examples:

SOME_VSTRING = CLASS_NAME (FROM 1 FOR 11)
OTHER_VSTRING = UNIT_NAME (FROM 11 FOR 14)

REMARKS:

Character positions begin with the leftmost character as character number one. Subsequent characters are numbered according to their position relative to character number one. References that extend beyond the end of the constant string shall be terminated at the end of the constant string.

CONST STRING ARRAY

data structure

PURPOSE: A CONST STRING ARRAY is a group of constants organized into an array. An array may have 1, 2 or 3 dimensions. The array as a whole has a name. An individual constant string (element) of an array is identified by an array index that specifies the position of the element within the array. An array index consists of subscripts - one subscript for each dimension.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The following statements are used to declare constant string arrays.

{for one dimensional array}
CONST STRING ARRAY *name* **(***low***:***high1***)** [**=**] **(***list***)**

{for two dimensional array}
CONST STRING ARRAY *name* **(***low***:***high1,* [*low***:**]*high2***)** [**=**] \
(*list***)**

{for three dimensional array}
CONST STRING ARRAY *name* **(***low***:***high1,* [*low* **:**]*high2,* \
[*low***:**] *high3***)** [**=**] **(***list***)**

Parameter	Description
name	Name of the array.
low	Lowest subscript value (must be 0 or 1).
highN	Highest subscript value for the Nth dimension. Must be a constant. The total number of elements in an array must not exceed 16K.
list	List of array element names. This one-dimensional list has one entry for each element of the array. Elements can be either explicit strings enclosed by quotation marks or names of previously defined constant strings. The order of the entries is determined by varying the first (leftmost) subscript the fastest, then the second and third.

NOTE: For two and three dimensional arrays the lowest subscript value for the second and third dimension must match the value that was specified by the first dimension.

Examples:

```
CONST STRING ARRAY C (1:3) (STRING1, STRING2, STRING3)
CONST STRING ARRAY CZ (0:4) (STRING0, STRING1, STRING2, \
STRING3)
CONST STRING ARRAY RESPONSES (1:3) (OPERATOR \
```

data structure

RESPONSE MESSAGE #1", "OPERATOR RESPONSE MESSAGE #2",\
OPERATOR RESPONSE MESSAGE #3")

REFERENCE:

Array element

{for one dimensional array}
name **(***exp1***)**

{for two dimensional array}
name **(***exp1*, *exp2***)**

{for three dimensional array}
name **(***exp1*, *exp2*, *exp3***)**

Parameter	Description
name	Name of the array.
expn	Value of the nth subscript. If a subscript value exceeds its specified range, it causes an error.

Examples:

ASTRING = C (3)
ASTRING = CC (i, j)
ASTRING = CCC (1, 2, k)
SUBSTRING = CC (i, j) (FROM 6 FOR 15)

Entire array

In some operations it is necessary to refer to an entire array as a whole. An entire array is referenced as follows:

name

- or -

name (*...)

Parameter	Description
name	Name of the array.
(*...)	Indicates the number of dimensions, (that is, (*), (*, *) or (*, *, *)).

The first form (just the name) is used when an array is passed as an argument to a function or monitor subroutine. The second form is used in other whole array operations (that is, array copy).

Example:

NEW_ARRAY (*) = OLD_ARRAY (*)

CONST STRING ARRAY *(continued)*

data structure

COMMANDS: Since constant string arrays are ***read only***, there are no commands to write to them.

> **NOTE:** Subscript range checking is done by the compiler if all of the subscripts of an array are constant. Range checking is always performed during program execution. Each time an array element is read, the subscript values are checked. If a subscript value is out of bounds, some error action is taken. If the error occurs during any mode other than FAULT mode (for example, NORMAL mode), the normal flow of control is aborted. The mode is changed to fault and control is transferred to fault logic. If the error occurs during FAULT mode, flow of control is not aborted.

CONTINUOUS

statement of a phase, state, or monitor subroutine

PURPOSE: This statement marks the beginning of the CONTINUOUS section of a phase, state, or monitor subroutine. This section provides continuous operations specific to the subroutine. This is the only executable section of a monitor subroutine.

LANGUAGE:

BATCH 90	UDF

FORMAT:

CONTINUOUS [**LOGIC**]
statement
•
•
•
statement

Example:

```
CONTINUOUS
   IF (FIC501.STS = BAD) FAULT 4
   IF (FIC502.STS = BAD) FAULT 5
```

REMARKS: When the phase or state subroutine is active, this section executes continuously (the entire section is executed once every function block cycle). It executes in all modes (that is, normal, fault, etc.). No ***loops*** (except FOR LOOP) or ***waits*** are allowed in this section.

CONTROL STATION

function block

PURPOSE: Control station function blocks (function code 80) provide an interface between the MFP module and operator interface devices (for example, OIS, DCS). A control station provides several modes for managing its set point and control output.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Before a control station can be accessed from a subroutine, it must be declared. The declaration can appear only in a data section.

BLOCK *csname***, BLK =** *const***, FC = CS** [**, WATCH**]

Parameter	Description
csname	Name of the control station function block.
const	Function block address.
WATCH (Batch 90 only)	Control station to be placed on watch by the Batch Historian upon program start-up. If not specified, the control station will be off watch.

Example: BLOCK FC501, BLK = 100, FC = CS

REFERENCES: Control station data may be read by using the appropriate name in an expression.

Get control output *csname***.CO**

Parameter	Description
csname	Name of the control station function block.
*csname***.CO**	Control output.

Example: X = FC501.CO

Get set point *csname***.SP**

Parameter	Description
csname	Name of the control station function block.
*csname***.SP**	Set point.

Example: X = FC501.SP

Get ratio index *csname*.**RX**

Parameter	Description
csname	Name of the control station function block.
csname.**RX**	Ratio index.

Example: X = FC501.RX

Get manual/auto status *csname*.**MODE**

Parameter	Description
csname	Name of the control station function block.
csname.**MODE**	Mode: 0 = manual. 1 = auto. 2 = cascade/ratio.

Example: X = FC501.MODE

Get process variable *csname*.**PV**

Parameter	Description
csname	Name of the control station function block.
csname.**PV**	Process variable.

Example: X = FC501.PV

Get high alarm status *csname*.**HAL**

Parameter	Description
csname	Name of the control station function block.
csname.**HAL**	High alarm status: 1 (true) = high alarm. 0 (false) = not high alarm.

Example: X = FC501.HAL

Get low alarm status *csname*.**LAL**

Parameter	Description
csname	Name of the control station function block.
csname.**LAL**	Low alarm status: 1 (true) = low alarm. 0 (false) = not low alarm.

Example: X = FC501.LAL

CONTROL STATION *(continued)*

function block

Get deviation alarm status *csname***.DAL**

Parameter	Description
csname	Name of the control station function block.
*csname***.DAL**	Deviation alarm status: 1 (true) = deviation alarm. 0 (false) = not deviation alarm.

Example: X = FC501.DAL

COMMANDS:

Set mode **SET** *csname***.MODE =** *exp*

Parameter	Description
csname	Name of the control station function block.
exp	Mode: 0 = manual. 1 = auto. 2 = cascade/ratio.

Example: SET FC501.MODE = AUTO

Set set point **SET** *csname***.SP =** *exp*

Parameter	Description
csname	Name of the control station function block.
exp	Set point.

Example: SET FC501.SP = 53.2

Set ratio index **SET** *csname***.RX =** *exp*

Parameter	Description
csname	Name of the control station function block.
exp	Ratio index.

Example: SET FC501.RX = 0.50

Set control output **SET** *csname***.CO =** *exp*

Parameter	Description
csname	Name of the control station function block.
exp	Control output.

Example: SET FC501.CO = 53.2

***CONTROL STATION** (continued)*

function block

Set watch status (Batch 90 only)

SET *csname***.WATCH =** *exp*

Parameter	Description
csname	Name of the control station function block.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET CS501.WATCH = ON

REMARKS:

These commands act like OIS commands. Specifically, the control station may refuse a command depending on the mode and auxiliary logic affecting the station. For example, if the output of a station is in CO track or the station is in automatic, the SET.CO statement is ignored. These commands are buffered within the function block and do not have an immediate effect on the function block. The command will not take effect until the function block executes on its next cycle.

CYCLE TIME

variable

PURPOSE: CYCLE TIME is a built in, read only variable. Value of the CYCLE TIME variable is the elapsed time (in seconds) from the start of the previous function block cycle to the start of the current cycle.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: CYCLE TIME is read by using it in an expression.

Example:

```
TOTAL = TOTAL + CYCLE TIME
```

DATA

statement

PURPOSE:

This is the first statement of the data section of a UDF program. This section defines global data, function codes, and symbols. All declarations in the data section are global since they can be referenced by all function and monitor subroutines in the UDF program.

There may be more than one data section in a program. The second data section is often used to declare global monitor subroutines. Since global monitor subroutines must be defined in the program before they are declared, the second data section should appear after the monitor subroutine definitions.

LANGUAGE:

BATCH 90	UDF

FORMAT:

DATA
type1 name1
type2 name2
•
•
•
END DATA

Parameter	Description
type	Any one of the following: **AUX** Auxiliary UDF function block. **BLOCK** Function block. **CONST** Constant. **CONST ARRAY** Constant array. **FGEN** Function generator. **VAR ARRAY** Variable array. **INPUT** UDF block input. **INTEGRATOR** Global integrator. **MONITOR** Monitor call. **OUTPUT** UDF block output. **RAMP** Ramp. **SPEC** UDF block specification. **TIMER** Timer. **UDF1** UDF type one. **UDF2** UDF type two. **VAR** Variable.
name	Name of the data.

Example:

```
DATA
    INPUT 1 P105, FC = MSDD
    INPUT 2 FIC105, FC = CS
    OUTPUT 0 FE105_DT
```

DATA *(continued)*

statement

```
    OUTPUT 1 FQ105
    SPEC 9 DELTA_T
    SPEC 10 MAX_CHARGE
    CONST NOT_OPEN = 0
    VAR SUM
    BLOCK START_PB, BLK = 125, FC = RCM
END DATA
```

REMARKS: For function block declaration details, refer to:

AOL	Analog exception report
APID	Advanced PID
BBUF	Boolean data buffer
CS	Control station
CSEQ	Common sequence
DATAEXPT	User defined data export
DOL	Digital exception report
DD	Device driver
MSDD	Multi-state device driver
RBUF	Real data buffer
RCM	Remote control memory
REMSET	Remote manual set constant
RMC	Remote motor control
SMITH	Smith block
TEXT	Text

Other function codes may be referenced by their function code number.

Data is initialized when a UDF program is run for the first time (that is, on the multi-function processor transition to execute).

The active data structures (for example, timers and monitor calls) execute independently of the active state subroutine. Active data structures declared in a data section are sometimes referred to as being global. Their values are initialized to zero at the start of the UDF sequence. They are never initialized again unless explicitly done so through a statement such as RESET.

DATA BUFFER

function block

PURPOSE: The data buffer function blocks (function codes 137, 138) provide the general mechanism for getting data from a batch or UDF program to the function blocks. Function code 137 is the real buffer, and function code 138 is the boolean buffer. A batch or UDF subroutine writes to the buffer function blocks using the statements described below. The data is then available (in the form of function block outputs) to the other function blocks.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Before a data buffer can be accessed from a subroutine, it must be declared. The declaration can appear only in the batch data section of a batch program.

BLOCK *fbname*, **BLK =** *const*, **FC =** *fc* [**, OUT (0) =** \
oname0] [**, OUT (1) =** *oname1*] [**, OUT (2) =** *oname2*] \
[**, OUT (3) =** *oname3*]

Parameter	Description
fbname	Name of the data buffer function block.
const	Function block address.
fc	Function code name: RBUF (function code 137.). BBUF (function code 138.).
oname0	Name of output 0.
oname1	Name of output 1.
oname2	Name of output 2.
oname3	Name of output 3.

Example:

```
BLOCK BUF01, BLK = 100, FC = RBUF, OUT (0) = FT501, \
OUT (1) = FT502, OUT (3) = FT504
```

REFERENCE:

Get output value *oname*.**VAL**

Parameter	Description
oname	Name of the data buffer function block output.
oname.**VAL**	Value.

Example:

```
X = FT501.VAL
```

DATA BUFFER (continued)

function block

COMMANDS:

Set output value **SET** *oname*.**VAL** **=** *exp*

Parameter	Description
oname	Name of the data buffer function block output.
exp	Value.

Example: SET FT501.VAL = 100.0

NOTE: If *oname* is an BBUF (function code 138), a value >= -0.5 and < 0.5 is considered false (0), otherwise it is true (1).

Set output quality **SET** *oname*.**Q** **=** *exp*

Parameter	Description
oname	Name of the data buffer function block output.
exp	Quality: 0 = good. 1 = bad.

Example: SET FV501.Q = BAD

DATA EXPORT

function block

PURPOSE:

Data Export function codes (DATAEXPT, function code 194) enable the batch program to display operator messages at the OIS console as well as receive text string inputs from the console. The DATAEXPT function block can be declared in the BATCH DATA or UNIT DATA sections of a batch program or in the DATA section of a UDF program.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

BLOCK *strblk*, **BLK** = *cons1*, **FC** = **DATAEXPT**

Parameter	Description
ctrblk	Name of the DATAEXPT function code.
cons1	Block address of the DATAEXPT function block.

Examples:

```
BLOCK DIALOGUE, BLK = 1024, FC = DATAEXPT
BLOCK STATUS_IND, BLK = 9021, FC = DATAEXPT
```

REMARKS:

Batch programs are able to write strings to the DATAEXPT function block only if specifications S1, S2, and S3 are set to a value of two (default value). The data export function block actually contains two separate strings. The input string corresponds to the operator entered response strings received from the OIS console. The output string corresponds to the output of the function block. The relationship between the input and output strings is determined by the mode and data interlock status of the DATAEXPT function block. Both mode and data interlock attributes may be read or written by the batch program when specifications S1, S2, and S3 are set to a value of two.

REFERENCES:

Get current input string length

strblk.**CLEN**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**CLEN**	The current number of characters included in the current input string value of the DATAEXPT function block.

Examples:

```
SOME_VAR = DIALOGUE.CLEN
OTHER_VAR = STATUS_IND.CLEN
```

DATA EXPORT (continued)

function block

Get maximum output string length

strblk.**MLEN**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**MLEN**	The maximum number of characters in either string value of the DATAEXPT function block (specification S4).

Examples:

SOME_VAR = DIALOGUE.MLEN
OTHER_VAR = STATUS_IND.MLEN

Get output string value

strblk.**OSTR**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**OSTR**	Output string value.

Examples:

PROMPT_MSG = DIALOGUE.OSTR
REQUIRED_ACTION = STATUS_IND.OSTR

REMARKS: Only the entire output string will be retrieved.

Get input string value

strblk.**ISTR**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**ISTR**	Value of the last operator entered input string response.

Examples:

PROMPT_MSG = DIALOGUE.ISTR
REQUIRED_ACTION = STATUS_IND.ISTR

REMARKS: Only the entire input string will be retrieved.

Get alarm status

strblk.**ALM**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**ALM**	Alarm status: 0 = no alarm. 1 = level one alarm. 2 = level two alarm. 3 = level three alarm.

Examples:

SOME_VAR = DIALOGUE.ALM
IF (STATUS_IND.ALM > 0) OTHER_VAR = TRUE

DATA EXPORT *(continued)*

function block

Get mode *strblk*.**MODE**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**MODE**	Control mode: 0 = manual. 1 = auto.

Examples: SOME_VAR = DIALOGUE.MODE
OTHER_VAR = STATUS_IND.MODE

REMARKS: In auto mode, only the batch program can write new output string values. In manual mode, the output string may be written by the batch program as well as echo the input string received from the console. The initial mode of the DATAEXPT function block upon start-up is manual.

Get data interlock status *strblk*.**DILK**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**DILK**	Interlock status: 0 = unlocked. 1 = locked.

Examples: SOME_VAR = DIALOGUE.DILK
OTHER_VAR = STATUS_IND.DILK

REMARKS: The data interlock status of the DATAEXPT function block determines what the function block does with input string from the OIS console. When the interlock status is locked (1), messages received from the console are ignored. When the interlock status is unlocked (0), messages from the console are read into the input string of the function block. The control mode determines whether or not the message is echoed by the output string. The initial interlock status of the DATAEXPT function block upon start-up is locked.

DATA EXPORT (continued)

function block

Get mode lock status *strblk*.**MLOCK**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**MLOCK**	Control mode change status: 0 = mode change unlocked (enabled). 1 = mode change locked (disabled).

Examples:

SOME_VAR = DIALOGUE.MLOCK
OTHER_VAR = STATUS_IND.MLOCK

REMARKS: If the mode lock status is unlocked, the mode of the function block can be changed by the operator of an OIS console or by the batch program. If the mode lock status is locked, the mode of the function block can not be changed. The initial mode lock status of the DATAEXPT function block upon start-up is unlocked.

Get new value status *strblk*.**NEW**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**NEW**	New value status: 0 = new value has not changed (false). 1 = new input string value exists (true).

Examples:

WAIT UNTIL (DIALOGUE.NEW = TRUE)
WAIT UNITL (STATUS_IND.NEW)

REMARKS: This reference may be used by a batch program to determine when a new message has been received from the OIS console. The status is latched in the true state when a new input string is received. It remains latched until reset by a read operation from the batch program.

Get truncation status *strblk*.**TRUNC**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**TRUNC**	Truncation status: 0 = string value not truncated (false). 1 = string value truncated (true).

Examples:

IF (DIALOGUE.TRUNC = TRUE)...
IF (STATUS_IND.TRUNC)...

REMARKS: When a string value exceeding the maximum string length specification of the function block is being written, the function block truncates the string, writes the value, and sets a flag.

Get quality status *strblk*.**Q**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**Q**	Current quality status: 0 = good. 1 = bad.

Examples:

SOME_VAR = DIALOGUE.Q
OTHER_VAR = STATUS_IND.Q

REMARKS: The quality status of the DATAEXPT function block upon start-up is determined by the start-up control specification S6. If the specification is set to a value of zero, the quality on start-up is bad. If the specification is set to one, the quality on start-up is forced to good.

Get re-alarm status *strblk*.**REALM**

Parameter	Description
strblk	Name of the DATAEXPT function block.
strblk.**REALM**	A 0 to 1 or 1 to 0 transition in value indicates the function block has gone into or out of re-alarm condition.

Examples:

SOME_VAR = DIALOGUE.REALM
IF (STATUS_IND.REALM <> OLD_STATUS)...

REMARKS: The re-alarm status may be read by batch programs. The re-alarm condition is indicated by a change in the .REALM value.

DATA EXPORT (continued)

function block

COMMANDS:

Write to the output string

SET *strblk*.**OSTR = ** *strname* [**(FROM** *exp1* **FOR** *exp2***)**]

- or -

SET *strblk*.**OSTR = "***string_exp***"**

Parameter	Description
ctrblk	Name of the DATAEXPT function block.
strname	Name of the constant or variable string.
exp1	Starting point within the string.
exp2	Number of characters to be included in the substring. If not specified, the entire contents of *strname* is written to the output string.
string_exp	Explicit constant string to be written to the output string

Examples:

SET DIALOGUE.OSTR = "PROCESS CRITICAL !"
SET STATUS_IND.OSTR = REQUIRED_ACTION (FROM 1 FOR 32)

REMARKS:

The entire output string will be written. The ability to write to substrings within the output string is not supported.

Set mode

SET *strblk*.**MODE =** *exp*

Parameter	Description
strblk	Name of the DATAEXPT function block.
exp	Control mode: 0 = manual. 1 = auto.

Examples:

SET DIALOGUE.MODE = AUTO
SET STATUS_IND.MODE = MANUAL

REMARKS:

In auto mode, only the batch program can write new output string values. In manual mode, the output string may be written by the batch program as well as echo the input string received from the console. The initial mode of the DATAEXPT function block upon start-up is manual.

***DATA EXPORT** (continued)*

function block

Set data interlock status **SET** *strblk*.**MLOCK** **=** *exp*

Parameter	Description
strblk	Name of the DATAEXPT function block.
exp	Data interlock status: 0 = unlocked. 1 = locked.

Examples:

SET DIALOGUE.DILK = 0
SET STATUS_IND.DILK = 1

REMARKS:

The data interlock status of the DATAEXPT function block determines what the function block does with input string from the OIS console. When the interlock status is locked (1), messages received from the console are ignored. When the interlock status is unlocked (0), messages from the console are read into the input string of the function block. The control mode determines whether or not the message is echoed by the output string. The initial interlock status of the DATAEXPT function block upon start-up is unlocked.

Set mode lock status **SET** *strblk*.**MLOCK** **=** *exp*

Parameter	Description
strblk	Name of the DATAEXPT function block.
exp	Control mode lock status: 0 = unlocked. 1 = locked.

Examples:

SET DIALOGUE.MLOCK = OFF
SET STATUS_IND.MLOCK = ON

REMARKS:

If the mode lock status is unlocked, the mode of the function block can be changed by the operator of a OIS console or by the batch program. If the mode lock status is locked, the mode of the function block can not be changed. The initial mode lock status of the DATAEXPT function block upon start-up is unlocked.

DATA EXPORT *(continued)*

function block

Set alarm status **SET** *strblk***.ALM** **=** *exp*

Parameter	Description
strblk	Name of the DATAEXPT function block.
exp	Alarm status: 0 = no alarm. 1 = level one alarm. 2 = level two alarm. 3 = level three alarm.

Examples:

SET DIALOGUE.ALM = 0
SET STATUS_IND.ALM = 2

Set quality **SET** *strblk***.Q** **=** *exp*

Parameter	Description
strblk	Name of the DATAEXPT function block.
exp	Quality status: 0 = good. 1 = bad.

Examples:

SET DIALOGUE.Q = GOOD
SET STATUS_IND.Q = 1

REMARKS:

Character positions begin with the leftmost character as character number one. Subsequent characters are numbered according to their position relative to character number one. References that extend beyond the end of the constant string shall be terminated at the end of the constant string.

#DEBUG LEVEL

directive

PURPOSE:

The #DEBUG LEVEL directive tells the compiler to set the debug level for the statements that follow the directive. A higher debug level causes more statements to be debugable but causes the programs object file to be larger. A lower debug level causes fewer statements to be debugable but causes the object file to be smaller.

LANGUAGE:

BATCH 90	UDF

FORMAT:

#DEBUG LEVEL = *const*

Parameter	Description
const	A number from 1 to 4.

Example: #DEBUG LEVEL = 4

REMARKS:

Refer to the ***Batch Data Manager (Release 4.0)*** instruction for a description of the debug level function and how the debugger operates. Debug level determines which lines of the program may be stopped at and what the N+7 (line number of the program) output of the BSEQ function block is. The debug levels are:

Level 4 Can stop at all executable lines of code.

Level 3 Can stop at every occurrence of:

CALL
CASE
CONTINUOUS logic declaration
ENDSUBR for monitor and function subroutines
FAULT logic declaration
FOR...END FOR loop
FUNCTION subroutine declaration
HOLD logic declaration
IF
MONITOR subroutine declaration
NORMAL logic declaration
PHASE subroutine declaration
REPEAT...UNTIL
RESTART logic declaration
SET AND WAIT
WAIT FOR
WAIT UNTIL
WAIT WHILE
WHILE... ENDWHILE

#DEBUG LEVEL *(continued)*

directive

Level 2 Can stop at every occurrence of:

CALL
ENDSUBR of monitor and function subroutines
MONITOR subroutine declaration
SET AND WAIT
WAIT

Level 1 Can stop at every occurrence of:

SET AND WAIT
WAIT

Level 0 Will not stop at any line of code.

This directive can be used anywhere in a program and as many times as desired. Different parts of the program may be at different debug levels.

DECLARATIONS

statement

PURPOSE: This statement marks the beginning of a DECLARATIONS section. Refer to ***BATCH DATA***, ***DATA***, ***FUNCTION***, ***MONITOR***, ***PHASE SUBR***, and ***STATE SUBR*** for details.

LANGUAGE:

BATCH 90	UDF

#DESCRIPTOR

directive

PURPOSE: This directive assigns a 16 character descriptor to the program. This descriptor is displayed by the various utilities to make it very clear what each object file represents.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#DESCRIPTOR** "*string*"

Parameter	Description
string	Up to 16 ASCII characters.

Example: #DESCRIPTOR "K41 BATCH LOGIC"

NOTE: If the descriptor specified by "string" contains more than 16 characters, the compiler truncates the descriptor to the first 16 characters and generates a warning.

DEVICE DRIVER

function block

PURPOSE:

Device driver function blocks (function code 123) are used to control two-state devices (for example, opened and closed).

LANGUAGE:

BATCH 90	UDF

DECLARATION:

A device driver must be declared before it can be accessed from a subroutine. The declaration must appear in a BATCH DATA (Batch 90), UNIT DATA (Batch 90), or DATA (UDF) sections. Batch 90 and UDF languages will only manipulate device driver function blocks in which specification S1 is set to two.

BLOCK *ddname*, **BLK** = *const*, **FC** = **DD** [*co_action*]
[, **WATCH**]

Parameter	Description
ddname	Device driver function block name.
const	Function block address.
co_action	Specifies whether the control output value is to be inverted as it passes to and from the function block domain (NORMAL-normal CO value, INVERT-inverted CO value). If no action is specified, NORMAL is assumed.
WATCH (Batch 90 only)	Specifies device driver to be placed on watch by the Batch Historian upon program start-up. If not specified, the device drive will be off watch.

Examples:

```
BLOCK FV501, BLK = 100, FC = DD NORMAL
BLOCK FV502, BLK = 150, FC = DD INVER
```

REFERENCES:

Device driver data may be read by using the appropriate name in an expression.

Get control output

ddname.**CO**

Parameter	Description
ddname	Device driver function block name.
ddname.**CO**	Control output.

Example:

```
X = FV501.CO
```

NOTE: Refer to Co-Action in ***DECLARATIONS***.

DEVICE DRIVER (continued)

function block

Get status *ddname*.**STS**

Parameter	Description
ddname	Device driver function block name.
ddname.**STS**	Status: 0 = good. 1 = bad. 2 = waiting.

REMARKS: The time duration used by the device driver function block when waiting to verify its feedback inputs is determined by specification S9 of the function block.

Example: IF (FV501.STS = BAD) THEN

Get manual/auto mode *ddname*.**MODE**

Parameter	Description
ddname	Device driver function block name.
ddname.**MODE**	Mode: 0 = manual. 1 = auto.

Example: IF (FV501.MODE IS AUTO) THEN

Get tracking status *ddname*.**TRK**

Parameter	Description
ddname	Device driver function block name.
ddname.**TRK**	Tracking status: 0 = tracking off. 1 = tracking on.

Example: IF (FV501.TRK = 0) FAULT 15

COMMANDS:

Set mode **SET** *ddname*.**MODE** **=** *exp*

Parameter	Description
ddname	Device driver function block name.
exp	Mode: 0 = manual. 1 = auto.

Example: SET FV501.MODE = AUTO
SET FV501.MODE = 1

DEVICE DRIVER *(continued)*

function block

Set control output

SET AND WAIT *ddname*.**CO** = *exp*

- or -

SET *ddname*.**CO** = *exp*

Parameter	Description
ddname	Device driver function block name.
exp	Control output.

NOTE: The operation of this command depends on the mode and tracking status of the function block.

1. If the mode is auto, the control output state is set to exp.

2. If the mode is not auto and tracking is disabled, exp is recorded inside the function block, but the control output state is not updated. If the mode is later changed to auto, then the control output state is updated.

3. If the mode is not auto and tracking is enabled, the command has no effect.

The SET AND WAIT version of this command performs this operation and then waits for confirmation (that is, the control output state is equal to exp and the feedback status is good). The command retries the operation (once per cycle) while waiting.

Confirmation does not require that the mode is auto. It requires the feedback status is good and the control output is in the specified state.

The SET version performs the same operation but does not wait.

Examples:

```
SET FV501.CO = OPEN
SET FV501.CO = 1
```

DEVICE DRIVER *(continued)*

function block

Set status override **SET** *ddname***.OVR** **=** *exp*

Parameter	Description
ddname	Device driver function block name.
exp	Value: 0 = override off. 1 = override on.

Example: SET FV501.OVR = ON
SET FV501.OVR = 1

Set tracking status **SET** *ddname***.TRK** **=** *exp*

Parameter	Description
ddname	Device driver function block name.
exp	Device tracking: 0 = off. 1 = on.

Example: SET FV505.TRK = ON

REMARKS:

The purpose of control output tracking is to provide a bumpless transfer of the control output during a mode change from manual to auto. In manual mode, the operator controls the control output value. If tracking is disabled and the mode is changed from manual to auto, the control output will go to the last value that was written to the function block by the program (that is, the control output may bump). If tracking is enabled, the control output value will remain unchanged during the mode change. The initial value of tracking status (when the MFP module is placed into the execute mode) is off.

Set watch status (Batch 90 only) **SET** *ddname***.WATCH** **=** *exp*

Parameter	Description
ddname	Device driver function block name.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET FV505.WATCH = ON

REMARKS:

These commands are buffered within the function block and do not have an immediate effect on the function block. The command will not take effect until the function block executes on its next cycle.

DIGITAL EXCEPTION REPORT

function block

PURPOSE:

The digital exception report (DOL, function code 45) transmits a digital (boolean) value and its attributes (quality and alarm) via INFI-NET network to operator consoles and other nodes.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

Each DOL function block to be used by a batch program must be declared as follows:

NOTE: Input S1 of the DOL function block must be set to 2. Other-wise, batch program commands will be ignored.

BLOCK *name*, **BLK** = const, **FC** = **DOL** [, **WATCH**]

Parameter	Description
name	Name of the function block.
const	Function block address.
WATCH (Batch 90 only)	The DOL function block placed on watch by the Batch Historian on program start-up. If WATCH is not specified, the initial watch status is off.

Example: BLOCK DOL501, BLK = 240, FC = DOL

REFERENCES:

DOL data may be read by using the appropriate name in expression.

Get value *name*.**VAL**

Parameter	Description
name	Name of the DOL function block.
name.**VAL**	Current value.

Example: X = DOL501.VAL

Get quality *name*.**Q**

Parameter	Description
name	Name of the DOL function block.
name.**Q**	Current quality: 0 = good. 1 = bad.

Example: X = DOL501.Q

DIGITAL EXCEPTION REPORT *(continued)*

function block

Get alarm status *name*.**ALM**

Parameter	Description
name	Name of the DOL function block.
name.**ALM**	Current alarm status: 0 = no alarm. 1 = alarm.

Example: X = DOL501.ALM

Get quality mask *name*.**QMSK**

Parameter	Description
name	Name of the DOL function block.
name.**QMSK**	Current quality mask: 0 = forced good. 1 = forced bad. 2 = not forced.

Example: X = DOL501.QMSK

Get alarm mask *name*.**AMSK**

Parameter	Description
name	Name of the DOL function block.
name.**AMSK**	Current alarm mask: 0 = forced no alarm. 1 = forced alarm. 2 = not forced.

Example: X = DOL501.AMSK

COMMANDS: The following statements are used to control DOL function blocks:

Set value **SET** *name*.**VAL** **=** *exp*

Parameter	Description
name	Name of the DOL function block.
exp	Value.

Example: SET DOL501.VAL = 0

DIGITAL EXCEPTION REPORT *(continued)*

function block

Set quality mask

SET *name***.QMSK = ***exp*

Parameter	Description
name	Name of the DOL function block.
exp	Quality mask: 0 = forced 0 (good). 1 = forced 1 (bad). 2 = not forced.

Examples:

SET DOL501.QMSK = 0
SET DOL501.QMSK = FORCED_OFF

Set alarm mask

SET *name***.AMSK = ***exp*

Parameter	Description
name	Name of the DOL function block.
exp	Alarm mask: 0 = forced 0 (good). 1 = forced 1 (bad). 2 = not forced.

Examples:

SET DOL501.AMSK = 0
SET DOL501.AMSK = FORCED_OFF

Set watch status (Batch 90 only)

SET *name***.WATCH = ***exp*

Parameter	Description
name	Name of the DOL function block.
exp	Batch historian watch status: 0 = off. 1 = on.

Example:

SET DOL501.WATCH = ON

DIGIT

function

PURPOSE: The digit function extracts a single digit from a decimal number.

LANGUAGE:

BATCH 90	UDF

FORMAT: **DIGIT (***exp1***,***exp2***)**

Parameter	Description
exp1	Number from which digit is to be extracted.
exp2	Digit position.

Example:

```
IF (DIGIT (FV501.FBT, 2) = 1) THEN
IF (DIGIT (DE201, 1) = 5) THEN
```

NOTE: Digits are numbered from right to left, starting with 0 for the units digit. For example, the result of digit (678, 0) is 8. The result of digit (678, 1) is 7.

REMARKS: A common use for this operation is to examine feedback targets and values of a multi-state device driver function block. The .FBT reference extension returns a four digit value. Figure 4-2 shows the digit positions for the feedbacks on the multi-state device driver:

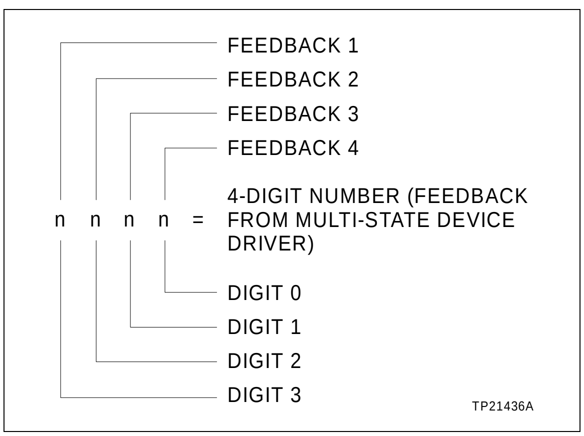


Figure 4-2. Multi-State Device Driver Digit Positions

Examples:

```
FEEDBACK_1 = DIGIT (MSDD_1.FBT, 3)
FEEDBACK_3 = DIGIT (MSDD_1.FBT, 1)
```

DISABLE EXTERNAL HOLD

statement

PURPOSE: This statement disables EXTERNAL HOLD; that is, it causes the batch sequence block to ignore the hold state of its run/hold input (specification S3). There is a complementary ENABLE EXTERNAL HOLD statement. These two statements allow a batch program to have ***critical sections*** of code that cannot be interrupted by the operator.

LANGUAGE:

BATCH 90	UDF

FORMAT: **DISABLE EXTERNAL HOLD**

Example:

```
DISABLE  EXTERNAL  HOLD
          •
          •
          •
ENABLE  EXTERNAL  HOLD
```

REMARKS: Once EXTERNAL HOLD is disabled, it remains disabled until explicitly enabled by the program.

EXTERNAL HOLD can also be disabled by the debugger utility.

DISABLE FAULT CODE

statement

PURPOSE: This statement disables fault code: that is, it causes subsequent FAULT statements to be ignored.

LANGUAGE:

BATCH 90	UDF

FORMAT: **DISABLE FAULT CODE**

Example:

```
DISABLE FAULT CODE
        •
        •
        •
ENABLE FAULT CODE
```

REMARKS: Refer to ***ENABLE FAULT CODE*** statement and ***FAULT*** statement.

NOTE: Run-time errors indicated by negative fault code numbers are not disabled by this statement.

DONE

statement

PURPOSE: The DONE statement indicates that a batch program has finished.

LANGUAGE: BATCH 90 | UDF

FORMAT: **DONE**

Example: DONE

REMARKS: This statement terminates execution of the batch program and sets the program status to BATCH COMPLETE (inactive).

#EJECT

directive

PURPOSE: The #EJECT directive tells the compiler to insert a form feed in the listing file. This is used to separate pages in the printed listing.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#EJECT**

Example: #EJECT

ENABLE EXTERNAL HOLD

statement

PURPOSE: This statement enables EXTERNAL HOLD; that is, it causes the batch sequence function block to recognize the hold state of its Run/Hold input (S3). There is a complementary DISABLE EXTERNAL HOLD statement. These two statements allow a batch program to have critical sections of code which cannot be interrupted by the operator.

LANGUAGE:

BATCH 90	UDF

FORMAT: **ENABLE EXTERNAL HOLD**

Example:

```
DISABLE  EXTERNAL  HOLD
        •
        •
        •
ENABLE  EXTERNAL  HOLD
```

REMARKS: Once EXTERNAL HOLD is disabled, it remains disabled until explicitly enabled by the program.

EXTERNAL HOLD can also be disabled by the debugger utility. This statement does not affect the disable request of the debugger utility.

ENABLE FAULT CODE

statement

PURPOSE: This statement is the inverse of the DISABLE FAULT CODE statement (it clears the condition set by the DISABLE FAULT CODE statement).

LANGUAGE:

BATCH 90	UDF

FORMAT: **ENABLE FAULT CODE**

Example:

```
DISABLE FAULT CODE
      •
      •
      •
ENABLE FAULT CODE
```

REMARKS: Refer to ***FAULT*** statement.

ENDSUBR

statement

PURPOSE: This statement marks the end of a subroutine (phase (Batch 90), state (UDF), function, or monitor).

LANGUAGE:

BATCH 90	UDF

FORMAT: **ENDSUBR** [*name*]

Parameter	Description
name	Name of the subroutine (optional).

Example: ENDSUBR ADD_A

NOTE: A warning is generated if *name* does not match the subroutines name.

FAULT

statement

PURPOSE: The FAULT statement changes the mode of a phase (Batch 90) or state (UDF) subroutine to FAULT; that is, it transfers control from the current logic section (NORMAL, HOLD or RESTART) to the FAULT LOGIC section.

LANGUAGE:

BATCH 90	UDF

FORMAT: **FAULT** *exp*

Parameter	Description
exp	Value to be assigned to the built-in status variable named fault code. This value must be positive.

Examples:

```
CONTINUOUS
      •
      •
      •
   IF ( ...) FAULT 10
      •
      •
      •
FAULT LOGIC

IF (FAULT CODE IS 10) THEN
   SET FV501.CO = OPEN
   SET AG1.CO = ON
ENDIF
```

REMARKS: FAULT CODE provides a convenient way to pass a small amount of information to the FAULT LOGIC. The value is read by using FAULT CODE in an expression.

The FAULT statement may not be used in the FAULT LOGIC section.

The FAULT statement has no effect if fault code is disabled. Any one of the following conditions disables the fault code.

1. Program disable - set by DISABLE FAULT CODE, cleared by ENABLE FAULT CODE.

2. Restart disable - set by transition to RESTART LOGIC, cleared by ENABLE FAULT CODE or transition to NORMAL LOGIC.

3. Debug disable - set and cleared by debugger utility.

FAULT LOGIC

section

PURPOSE: This statement marks the beginning of a phase (Batch 90) or state (UDF) subroutines FAULT LOGIC section. It provides the sequential control of an operation in fault mode.

LANGUAGE:

BATCH 90	UDF

FORMAT:

FAULT LOGIC
statement
•
•
•
statement

Example:

```
FAULT LOGIC
   SET FIC401.CO = OPEN
   SET FIC402.CO = CLOSED
```

REMARKS: The FAULT LOGIC section is activated when the program sequence mode is changed from HOLD, NORMAL or RESTART to FAULT.

When the last statement of this section is executed, the program sequence enters the holding state and all phase activity is suspended. Refer to Appendix A for more information about parallel phase execution.

When the program sequence state is changed to run, sequential control is transferred to the RESTART LOGIC of the current phase, or if a new phase has been selected, then control starts with the NORMAL LOGIC of the new phase.

If the FAULT LOGIC executes a RESTART statement, control is transferred directly to the RESTART LOGIC.

FIRST CYCLE

variable

PURPOSE: FIRST CYCLE is a built in status variable. It is true during the first cycle of any subroutine and false thereafter.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: FIRST CYCLE is read by using it in an expression.

Example:

```
IF (FIRST CYCLE) THEN
    X = 10
ENDIF
```

FOR...END FOR LOOP

control structure

PURPOSE: This looping structure repeatedly executes a block of statements while the specified loop test is true.

LANGUAGE:

BATCH 90	UDF

FORMAT:

FOR *var* **=** *const1* [**STEP** *const1*]**TO** *const2* **DO**
    statement1
        •
        •
        •
    statementN
END FOR [*var*]

- or -

FOR *var* **=** *const1* [**STEP** *const1*]**WHILE** *exp3* **DO**
    statement1
        •
        •
        •
    statementN
END FOR [*var*]

Parameter	Description
var	Loop variable.
exp1	Initial value of *var*.
const1	Step value. This value may be positive or negative. This term (**STEP** *const1*) is optional. If omitted value is taken as one.
exp2	Final value of loop variable. The loop test compares loop variable to final value. The sense of comparison depends on sign of step value. If step is positive, loop test is var <= final value. If step is negative, loop test is var >= final value.
exp3	Loop test expression.

Examples:

```
FOR I = 1 TO 5 DO A(I) = 1          {Executes 5 times}
END FOR
FOR I = 5 STEP -1 TO 2 DO A(I) = 1  {Executes 4 times}
END FOR
FOR I = 1 WHILE (I <= X) DO A(I) = 1  {Executes X times}
END FOR
```

FOR...END FOR LOOP (continued)

control structure

REMARKS: Following is a description of the FOR LOOP operation.

Initialize loop (var = exp). Perform loop test. If result is:

True	Execute statements 1 through N. Add step value to loop variable (var = var + step). Repeat loop test.
False	Exit loop (transfer control to statement following END FOR).

A FOR loop may be terminated by a BREAK statement (control transfers to the statement following END FOR).

Other looping structures (for example, REPEAT-UNTIL) suspend the program on each cycle of the loop. The FOR loop does not suspend the program. Therefore:

- The FOR loop may be used in continuous sections.
- The FOR loop should be used only for ***closed*** loops (where the number of iterations is predictable); for example, indexing through an array. The FOR loop should not be used for open-ended loops (for example, waiting for some function block event). Keep in mind that function blocks (in the same segment as the program) cannot execute while a FOR loop is executing.

FUNCTION

subroutine

PURPOSE:

A function subroutine is a unit of repetitive logic. The function subroutine may be called by a phase (Batch 90), state (UDF), monitor or another function subroutine. Information is passed to a function subroutine by means of parameters. A function subroutine can also return a value to the caller. This means that a function subroutine call can be used in an expression in the same way that a simple variable is used.

Figure 4-3 shows the basic function subroutine structure.

```
FUNCTION name
    Parameter list
DECLARATIONS
    Local data
EXECUTABLE
    Executable statements
ENDSUBR  [name]
```

Figure 4-3. Function Subroutine Structure

LANGUAGE:

BATCH 90	UDF

HEADING:

A function subroutine heading defines the name of the subroutine and its formal parameters. The heading format is explained below:

FUNCTION *name*
  type1 fname1
  type2 fname2
   •
   •
   •

NOTE: When passing a numeric recipe item to a function, it must be defined as type ANY.

When a formal parameter is type ARRAY, the parameter specification must indicate the number of dimensions as follows:

VAR ARRAY *fnameN (*...)*

Parameter	Description
name	Name assigned to this subroutine.
fname	Local name assigned to the nth formal parameter.

FUNCTION (continued)

subroutine

Parameter	Description
typeN	Data type of the Nth formal parameter. Available data types are: **ANY** Any data type or expression. **AOL** Analog exception report. **APID** Advanced PID controller. **ARRAY** Any array type. **BBUF** Boolean data buffer. **BLOCK** Function block. **CS** Control station block. **CSEQ** Common sequence. **DATAEXPT** User defined data export block **DD** Device driver block. **DOL** Digital exception report. **FGEN** Function generator. **INTEGRATOR** Integrator. **MSDD** Multi-state device driver block. **RAMP** Ramp. **RBUF** Real data buffer. **RCM** Remote control memory. **REMSET** Remote manual set constant. **RMC** Remote motor control block. **SMITH** Smith controller block. **TEXT** Text block. **TIMER** Timer. **VAR** Variable.
(*…)	Indicates number of dimensions. (that is, (*), (*,*) or (*,*,*))

There is no fixed limit on the number of parameters. However, only one parameter per line is permitted. The parameter list is terminated by the start of the next section.

Parameters whose data type is ANY are passed by value and cannot be modified by the subroutine. All other parameters are passed by reference and may be modified by the subroutine.

Example:

```
FUNCTION  SET_DEVICE
    DD  DEVICE
    ANY  CMD_POSN
    VAR  FAULT_ID
DECLARATIONS
    VAR  ORIGINAL_MODE
EXECUTABLE
    ORIGINAL_MODE  =  DEVICE.MODE
    IF  (DEVICE.MODE  <>  AUTO)  SET  DEVICE.MODE  =  AUTO
    SET  DEVICE.CO  =  CMD_POSN
    WAIT  WHILE  (DEVICE.STS  =  WAITING)
```

```
    IF (DEVICE.STS = GOOD) THEN
        SET DEVICE.MODE = ORIGINAL_MODE
        FAULT_ID = 0
    ELSE
        FAULT FAULT_ID
    ENDIF
ENDSUBR SET_DEVICE
```

The formal parameter list defines local names (those to be used inside the subroutine). Each formal parameter can be thought of as a place holder for an actual parameter. The actual parameters are specified by the caller when it calls the function subroutine. Each actual parameter in the call refers to a value (type ANY) or to some object of data (for example, timer, variable, function block). When the function subroutine is executed, each reference to a formal parameter becomes reference to the corresponding actual parameter.

DECLARATIONS: The declarations statement marks the beginning of the declarations section. This section specifies data that is local to (owned by) the function. The following shows the format of the declarations section.

DECLARATIONS
type1 *name1*
•
•
•
typeN *nameN*

Parameter	Description
name	Name of data object.
type	Data type; must be one of the following: **VAR** Variable. **VAR ARRAY** Variable array. **VAR STRING** Variable string. **VAR STRING ARRAY** Variable string array. **CONST** Constant. **CONST ARRAY** Constant array. **CONST STRING** Constant string. **CONST STRING ARRAY** Constant string array. **BLOCK ARRAY** Function block array.

FUNCTION (continued)

subroutine

Example:

```
DECLARATIONS
    VAR       COUNTER
    CONST     TEN = 10.0
```

NOTE: The local data of a function exists only while the function subroutine is executing. The data disappears when the function subroutine returns to the caller.

Local data of a function subroutine is not initialized to any particular value.

FUNCTION CALL: There are two methods for calling function subroutines. The syntax used depends on whether or not the function subroutine has a return value.

Function does not have a value

The following syntax is used if the function subroutine does not have a value:

CALL *name* **(***param1, param2, ...***)**

Parameter	Description
name	Name of the function subroutine parameter.
paramN	Nth actual parameter.

Example: CALL CHARGE (FV501, TOTAL_A, FIQA)

Function does have a value

If a function subroutine does have a return value, the call must be embedded as an operand of an expression. The following syntax is used:

name **(***param1, param2, ...***)**

Parameter	Description
name	Name of the function subroutine.
paramN	Nth actual parameter.

Example: Consider a function subroutine called MAX_VAL whose value is the larger of two parameters. Following is an example of a call to MAX_VAL:

```
FUNCTION MAX_VAL
    ANY VAL1
    ANY VAL2
    ANY VAL3
DECLARATIONS
    VAR ARRAY VALUES (1, 3)
EXECUTABLE
    VALUES (1) = VAL1
```

subroutine

```
    VALUES (2) = VAL2
    VALUES (3) = VAL3
    RETURN (MAX VALUES (*))
ENDSUBR MAX_VAL
        •
        •
        •
SOME_VAR = 1 + MAX_VAL (X, Y, Z)
```

NOTES:

1. The data type of each actual parameter must match the data type of the corresponding formal parameter. For example, if a formal parameter type is timer, then the corresponding actual parameter must be a timer. However, if the data type of a formal parameter is ANY, then the corresponding actual parameter may be a value, a constant, or an expression.

2. A function subroutine must be defined before it is called.

3. Function subroutine is an ordinary subroutine in that it takes control from the caller, executes to completion and returns control to the caller (as opposed to a monitor subroutine that executes in parallel with the caller).

4. RETURN statements may appear anywhere in the executable section. The ENDSUBR statement is equivalent to a RETURN.

5. If any RETURN statement in a function subroutine specifies a return value, all RETURNS must specify values.

FUNCTION BLOCK

function block

PURPOSE:

Function blocks in an MFP batch system perform continuous functions including:

- Continuous control and computation.
- Process I/O (field and peer-to-peer).
- Operator I/O.

The function blocks are accessible from Batch 90 and UDF programs. General purpose access methods are described below. Function blocks are also accessible through special methods such as data buffer, device driver, control station resource manager, remote control memory, remote manual set constant.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

Function blocks must be declared before they can be accessed from a Batch 90 or UDF program. The declaration can appear only in the data section.

BLOCK *fbname*, **BLK =** *const1*, [**FC =** *const2*, **OUT (***0***) =** \
*0name0***, OUT (***1***) =** *oname1* ...]

Parameter	Description
fbname	Name of the function block.
const1	Function block address.
const2	Function code number.
oname0	Name of output 0.
oname1	Name of output 1.

Example:

BLOCK DI02A, BLK = 100, FC = 84, OUT (0) = FV501, OUT (1) \
= FV502

NOTE: Currently, the compiler does not use the function code database. Therefore, it is unable to tell if too many output names are assigned.

REFERENCE:

Subroutines may read the value of any declared function block output by using the name in an expression. If alarm and quality are associated with a function block output, these attributes are readable. Alarm and quality are associated with exception report generating function blocks and I/O function blocks.

Get output value *oname*.**VAL**

Parameter	Description
oname	Name of the function block output.
oname.**VAL**	Output value.

Examples:

X = FT501.VAL
X = PB101.VAL

Get output quality *oname*.**Q**

Parameter	Description
oname	Name of the function block output.
oname.**Q**	Block output quality: 0 = good. 1 = bad.

Example: IF (FT501.Q = BAD) SET FV500.CO = CLOSED

Get high alarm status *oname*.**HAL**

Parameter	Description
oname	Name of the function block output.
oname.**HAL**	High alarm status: 0 = false (not high). 1 = true (high).

Example: IF (FT501.HAL) THEN

Get low alarm status *oname*.**LAL**

Parameter	Description
oname	Name of the function block output.
oname.**LAL**	Low alarm status: 0 = false (not low). 1 = true (low).

Example: IF FT501.LAL) THEN

FUNCTION BLOCK (continued)

data structure

Get alarm status *oname*.**ALM**

Parameter	Description
oname	Name of the function block output.
oname.**ALM**	Alarm status: 0 = false (no alarm). 1 = true (alarm).

Example: IF (FT501.ALM) THEN

FUNCTION BLOCK ARRAY

data structure

PURPOSE: A function block array is a group of previously defined function block outputs (from any function code type) organized in a table. An array may have 1, 2 or 3 dimensions. The array as a whole has a name. An individual function block output (element) of an array is identified by an array index that specifies the position of the element within the array. An array index consists of subscripts - one subscript for each dimension. A function block array is a ***read only*** array that is used exclusively for referencing common reference extensions of function block outputs listed in the array.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The following statements are used to declare function block arrays:

{for one dimensional array}
BLOCK ARRAY *name* **(***low***:***high1***)** **(***list***)**

{for two dimensional array}
BLOCK ARRAY *name* **(***low***:***high1***,** [*low***:**]*high2***)** **(***list***)**

{for three dimensional array}
BLOCK ARRAY *name* **(***low***:***high1,* [*low***:**]*high2,* [*low***:**]*high3***)** **(***list***)**

Parameter	Description
name	Name of the array.
low	Lowest subscript value. Must be 0 or 1.
highN	Highest subscript value for the Nth dimension. Must be a constant. The total number of elements in an array must not exceed 16k.
list	List of array element names. This one dimensional list has one entry for each element of the array. The order of the entries is determined by varying the first (leftmost) subscript the fastest, then the second and third.

NOTE: For two and three dimensional arrays the lowest subscript value for the second and third dimension must match the value that was specified by the first dimension.

Examples:

```
BLOCK ARRAY FLOW_SIGNALS (1:3) \
(FT201, FT202, FT203)
BLOCK ARRAY CONTROL_FEED (1:2, 1:2), \
(PY504,FY504,PU504,TV504)
BLOCK ARRAY LOGIC_VLV (1:3, 1:2, 1:3) \
```

FUNCTION BLOCK ARRAY *(continued)*

data structure

(LS111,LS211,LS311,LS121,LS221,LS321, \
PV112,PV212,PV312,PV122,PV222,PV322, \
FY113,FY213,FY313,FY123,FY223,FY323)

REFERENCE:

Element of an array An individual element of an array is referenced as follows:

{for one dimensional array}
name **(***exp1***)** *.refext*

{for two dimensional array}
name **(***exp1, exp2***)** *.refext*

for three dimensional array}
name **(***exp1, exp2, exp3***)** *.refext*

Parameter	Description
name	Name of the array.
expN	Value of the Nth subscript. If a subscript value exceeds its specified range, it causes an error.
.refext	Valid reference extension for a generic function block array. If the output does not contain the attribute referenced, the resulting value is meaningless. Valid references are .VAL, .ALM, .Q, .HAL, and .LAL.

Examples: NEXT_TARGET = FEED_VALUES (1,3) .VAL
PRO_STAT = CELL_STATUS (15) .VAL

Entire array In some operations it is necessary to refer to an entire array as a whole. An entire array is referenced as follows:

name [*.refext*]

- or -

name (...)* [*.refext*]

Parameter	Description
name	Name of the array.
(...)*	Number of dimensions (that is, (*), (*,*), (*,*,*)).
.refext	Valid reference extension for a generic function block array.

The first form (just the name) is used when an array is passed as an argument to a function or monitor subroutine. The

FUNCTION BLOCK ARRAY *(continued)*

data structure

second form is used in other whole array operations (that is, array copy).

Examples:

```
CALL GET_ALARM (VALUE_STAT)
NEW_POSITION (*,*) = OLD_POSITION (*,*) .VAL
```

FUNCTION GENERATOR

data structure

PURPOSE: A function generator is an active data structure. It produces a value that varies in a piece-wise linear function of time.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The following is used to declare a function generator:

FGEN *fgname* **((***0*, *value0***)**, **(***time1***,** *value1***)...,** *units***)**

Parameter	Description
fgname	Name of the function generator.
time	Time of Nth breakpoint (must be a constant).
valueN	Value of Nth breakpoint (must be a constant).
units	Units of time (must be positive): **SEC** Seconds. **MIN** Minutes. **HOUR** Hours.

Example: FGEN PROFILE1 ((0, 80.0), (30, 100.0), (60, 100.0), (90, 20.0), \
MIN)

Refer to Figure 4-4 for an example function generator.

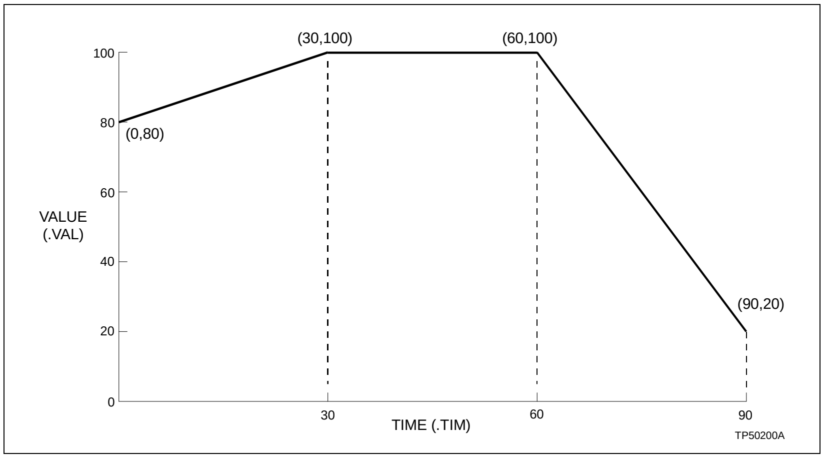


Figure 4-4. Example Function Generator

REFERENCE: Function generator data may be read by using the appropriate name in an expression.

FUNCTION GENERATOR *(continued)*

data structure

Get value *fgname*.**VAL**

Parameter	Description
fgname	Name of the function generator.
fgname.**VAL**	Current value.

Example: X = FGEN1.VAL

Get time value *fgname*.**TIM**

Parameter	Description
fgname	Name of the function generator.
fgname.**TIM**	Current time value.

Example: X = FGEN1.TIM

Get status *fgname*.**STS**

Parameter	Description
fgname	Name of the function generator.
fgname.**STS**	Current status: 0 = holding. 1 = running.

Example: X = FGEN1.STS

COMMANDS:

Start **START** *fgname*

Parameter	Description
fgname	Name of the function generator.

Example: START FGEN1

Hold **HOLD** *fgname*

Parameter	Description
fgname	Name of the function generator.

Example: HOLD FGEN1

FUNCTION GENERATOR (continued)

data structure

Reset **RESET** *fgname (exp)*

Parameter	Description
fgname	Name of the function generator.
exp	Initial time.

This sets the time to exp, the value (.VAL) to the appropriate value for the specified time and the status to holding.

Example: RESET FGEN1 (0)

GLOBAL DATA STRUCTURES (INTEGRATORS, TIMERS, ETC)

*Refer to **BATCH DATA** and **DATA** statements.*

#HIHI

directive

PURPOSE:

The #HIHI compiler directive forces the numeric recipe parameters to include the LOW-LOW and HI-HI boundary values. These values function identically to LOW and HI boundary values except these are absolute boundaries that can be set only from within the program.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

#HIHI

REMARKS:

The result of this directive is that the syntax for recipe arguments of type ANY are expanded to:

ANY *name* **=** **(***const1***<***const2***<***const3***<***const4***<***const5***)**

Parameter	Description
name	Name of the parameter.
const1	Default low-low limit.
const2	Default low limit.
const3	Default parameter value.
const4	Default high value.
const5	Default high-high value.

Any combination of default values may be specified by the program and is completely optional. For example:

(1) - Low-low limit only.
(<<3) - Value only.
(<<<4) - High limit only.
(<<3<4<5) - Value, high, and high-high only.
(1<2<3<4<5) - Low-low, low, value, high, and high-high.

HISTORIAN BUFFER

variable

PURPOSE: HISTORIAN BUFFER is a built-in status variable that indicates the fraction of the historian event queue that is currently in use.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: The value is read by using HISTORIAN BUFFER in an expression. The range of the value is 0 through 1.0. The value 0 indicates that the buffer is empty or no historians are expected.

Example:

```
WAIT UNTIL (HISTORIAN BUFFER < 0.5)
```

HISTORIAN STATUS

variable

PURPOSE: HISTORIAN STATUS is a built-in status variable that indicates the summary status of the expected batch historians.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: The value is read by using HISTORIAN STATUS in an expression.

0	Good. At least one expected historian has good status (or no historians are expected).
1	Bad. All expected historians have bad status.

Examples:

```
WAIT UNTIL (HISTORIAN STATUS IS GOOD)
WAIT WHILE (HISTORIAN STATUS)
```

HOLD

Refer to ***COMMON SEQUENCE****,* ***FUNCTION GENERATOR****,* ***INTEGRATOR****,* ***MONITOR****,* ***RAMP DATA****, or* ***TIMER****.*

HOLD LOGIC

section

PURPOSE: This statement marks the beginning of the hold logic section of a phase subroutine. This section provides the sequential control of an operation during the transition from normal execution to holding.

LANGUAGE:

BATCH 90	UDF

FORMAT:

HOLD LOGIC
    statement
        •
        •
        •
    statement

Example:

```
HOLD LOGIC
    SET FIC401.CO = OPEN
    SET FIC402.CO = CLOSED
```

REMARKS: This section is active during the transition from normal execution to holding. When the last statement of this section is executed, all phase activity is suspended until the phase is restarted. When the batch sequence mode is changed to run, sequential control is transferred to the restart logic (refer to Appendix A for details on parallel phase subroutine execution).

IF

statement

PURPOSE: The IF statement causes a single statement to be executed or passed over, based on a logical expression.

LANGUAGE:

BATCH 90	UDF

FORMAT: **IF (***exp***)** statement

Example: IF (FC101.MODE IS MAN) SET FC101.MODE = AUTO

REMARKS: IF exp is true, the statement is executed; otherwise, the statement is not executed. If no statement is defined, no action is taken.

IF THEN...ENDIF

control structure

PURPOSE: The IF THEN statement causes a block of statements to be executed or passed over, based on a logical expression.

LANGUAGE:

BATCH 90	UDF

FORMAT:

IF (*exp***) THEN**
statement1
•
•
•
statementN
ENDIF

Example:

```
IF (FC101.MODE IS MAN) THEN
    SET FC101.MODE = AUTO
    SET FC101.CO = 50.0
ENDIF
```

REMARKS: IF exp is true, statements 1 through N are executed; otherwise, statements 1 through N are skipped.

IF THEN...ELSE...ENDIF

control structure

PURPOSE: This statement causes one of two blocks of statements to be executed, and the other to be passed over, based on a logical expression.

LANGUAGE:

BATCH 90	UDF

FORMAT:

IF (*exp*) THEN
    statement T1
        •
        •
        •
    statement TN
ELSE
    statement F1
        •
        •
        •
    statement FN
ENDIF

Example:

```
IF  (FLAG  IS  TRUE)  THEN
      SET  FC101.MODE  =  AUTO
      SET  FC102.MODE  =  AUTO
ELSE
      SET  FC101.MODE  =  MAN
      SET  FC102.MODE  =  MAN
ENDIF
```

REMARKS: IF exp is true, statements T1 through TN are executed; otherwise, statements F1 through FN are executed.

#INCLUDE

directive

PURPOSE: The #INCLUDE directive tells the compiler to replace this line with the contents of a specified file.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#INCLUDE (***path******filename***)**

Parameter	Description
filename	Name of the file.

Example: #INCLUDE (\BATCH\FILE.ABC)

REMARKS: As the compiler processes a source file, it builds a temporary version of the file. The include directive affects only the temporary version. The original source file is not changed.

The purpose of this directive is to simplify the development of application standards. Include files provide a convenient means to maintain standard function, monitor, phase, or state subroutines for use in multiple Batch 90 and UDF programs. In this manner, the subroutine to open a valve (for example) can be written only once and included in each program application that needs this functionality.

INPUT

declaration

PURPOSE:

The INPUT declaration provides an alternate means of declaring function blocks within the UDF language. The UDF language fully supports the function block declarations used in ***DATA*** and ***FUNCTION***. Additionally, the INPUT declaration enables the declaration of function blocks without specific block address information. This functionality allows one UDF application to be implemented in many instances.

The INPUT declaration defines function blocks that have been configured as inputs to the UDF Type 1, UDF Type 2, or AUXILIARY UDF function blocks. Declared as an INPUT, the UDF program can manipulate these function blocks in the same way as the explicit declarations that included block address information (that is, BATCH DATA). All references and commands associated with the function code type are available to the UDF program.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

Function blocks must be declared before they can be accessed from an UDF program. The declarations can appear only in the data section.

[**AUX** *x*] **INPUT** *y name*, **FC** **=** *type*

Parameter	Description
AUX *x*	Name of the auxiliary UDF function block that connects to the UDF type 1 or 2 function block.
y	Input number of the UDF declaration function block or auxiliary UDF function block connected to the UDF type 1 or 2 function block.
name	Name for the input.
type	Function code acronym or function code number of the UDF type1 or 2 function block connected.

Example:

```
AUX 1
    INPUT 2 TIC503, FC = DD
    INPUT 3 TIC501, FC = CS
```

REMARKS:

For function block details, refer to:

AOL	Analog exception report
APID	Advanced PID
BBUF	Boolean data buffer
BSEQ	Batch sequence
CS	Control station
DOL	Digital exception report

INPUT *(continued)*

declaration

DD	Device driver
MSDD	Multi-state device driver
RBUF	Real data buffer
RCM	Remote control memory
REMSET	Remote manual set constant
RMC	Remote motor control
SMITH	Smith block
TEXT	Text

INTEGRATOR

data structure

PURPOSE: Integrators are active data structures. Integrators can be either local or global.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Integrators must be declared as follows:

INTEGRATOR *iname* **(***input*, *units***)**

Parameter	Description
iname	Name of the integrator.
input	Name of value to be integrated. This name must refer to a data object that has already been declared (for example, batch data, data, phase subroutine, state subroutine, and local data).
units	Units of time: **SEC** Seconds. **MIN** Minutes. **HOUR** Hours.

Example:

```
INTEGRATOR TOTAL501 (FI1501.VAL, MIN)
INTEGRATOR INTGX (X, SEC)
```

When declared locally (in a phase or state subroutine) the integrator value is initialized to zero at the start of the subroutine.

REFERENCE: Integrator data may be read by using the appropriate name in an expression.

Get value of integrator *iname***.VAL**

Parameter	Description
iname	Name of the integrator.
*iname***.VAL**	Current value of the integrator.

Example:

```
X = INTG1.VAL
```

Get status of integrator *iname***.STS**

Parameter	Description
iname	Name of the integrator.
*iname***.STS**	Current status: 0 = holding. 1 = running.

INTEGRATOR (continued)

data structure

COMMANDS:

Start integrator **START** *iname*

Parameter	Description
iname	Name of the integrator.

Example: START INTG1

NOTE: This sets the status to running (value is not affected).

Hold integrator **HOLD** *iname*

Parameter	Description
iname	Name of the integrator.

Example: HOLD INTG1

NOTE: This sets the status to holding (value is not affected).

Reset integrator **RESET** *iname* **(***exp***)**

Parameter	Description
iname	Name of the integrator.
exp	Initial integrator value.

Example: RESET INTG1 (0)

LAST OPERATION

variable

PURPOSE: LAST OPERATION and LAST OPERATION (*bseqx*) are built-in read only variables that return the operation number of the last operation within the unit recipe being executed or within a unit recipe being executed by a specific BSEQ function block.

LANGUAGE:

BATCH 90	UDF

FORMAT: **LAST OPERATION** - {for batch program looking at its own recipe}

- or -

LAST OPERATION (*bseqx***)** - {for batch or UDF program looking at another BSEQ function block within the same MFP module}

Parameter	Description
bseqx	Name of the BSEQ function block that is executing the current unit recipe.

REMARKS: The LAST OPERATION variable returns the operation number of the final operation for the currently executing unit recipe. The number is returned when the variable LAST OPERATION is used in an expression.

Example: FINAL_OP_NUM = LAST OPERATION

NOTE: LAST OPERATION is an ISA-S88.01-1995 standard term. Usage of this term prohibits compiling with Elsag Bailey specific terms. The Elsag Bailey specific term is LAST PHASE. To use the Elsag Bailey specific term, replace OPERATION with PHASE.

LAST PHASE

variable

NOTE: LAST PHASE is an Elsag Bailey specific term. Usage of this term prohibits compiling to ISA-S88.01-1995 standards. The ISA-S88.01-1995 standard term is LAST OPERATION. To use the Elsag Bailey specific term, refer to ***LAST OPERATION*** and replace OPERATION with PHASE.

LAST SUBR

variable

PURPOSE: LAST SUBR and LAST SUBR (*bseqx*) are built-in read only variables that return the phase subroutine number of the last phase subroutine within the current operation of the unit recipe being executed or within a unit recipe being executed by a specific BSEQ function block.

LANGUAGE:

BATCH 90	UDF

FORMAT: **LAST SUBR**

- or -

LAST SUBR (*bseqx*)

Parameter	Description
bseqx	Name of the BSEQ function block that is executing the current unit recipe.

REMARKS: The phase subroutine number of the last phase of the current operation is returned when LAST SUBR is used in an expression.

Example: NUM_PHASES = LAST SUBR

#LIST

directive

PURPOSE: The #LIST directive turns on the generation of the listing file if it was suspended by a #NOLIST directive. If the listing was not suspended, no action is taken.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#LIST**

LOT

variable

PURPOSE: LOT is a built-in status variable.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: LOT ID is read by using LOT in an expression.

Examples:

```
EXECUTION_NO = LOT
LOT_STORAGE_NO = LOT + 100
```

REMARKS: The Batch 90 language includes three built-in variables (BATCH, CAMPAIGN, and LOT). These variables may be used by the BHIST function block for record keeping, or by the CSEQ function block for controlling resource allocation. Refer to ***BATCH*** and ***CAMPAIGN*** statements.

#MAXPARALLEL

directive

PURPOSE: The #MAXPARALLEL directive specifies the maximum number of parallel phase subroutines permitted in an operation. The actual number of parallel phase subroutines in any operation is specified by the unit recipe.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#MAXPARALLEL = ** *const*

Parameter	Description
const	Value from 1 through 32.

Example: #MAXPARALLEL = 3

REMARKS: The #MAXPARALLEL directive is optional. If not used, the value of #MAXPARALLEL defaults to 1. This parameter directly effects the amount of RAM memory that needs to be allocated for step data (corresponds to specification S12 of the BSEQ function code). The batch compiler calculates the minimum number of parallel phase subroutines required and displays that number at the end of the program listing.

MONITOR

subroutine

PURPOSE:

A monitor subroutine is a program unit that is similar, in some ways, to a function subroutine. Like a function subroutine, a monitor subroutine is defined once and may be called many times. The definition may be thought of as a template and each call is an independent instance (use) of the template.

There are two major characteristics that distinguish a monitor subroutine from a function subroutine.

1. A monitor subroutine executes in parallel with the active phase (Batch 90) or state (UDF) subroutine.

2. The call to a monitor subroutine is a data declaration (rather than an executable statement).

Monitor subroutines are typically used to watch for abnormal conditions while a phase or state subroutine performs the sequential part of an operation procedure.

LANGUAGE:

BATCH 90	UDF

REMARKS:

The call to a monitor subroutine may be declared in the local data of a phase or state subroutine or in a data section. If the call is in a phase or state subroutine, then the instance of the monitor subroutine is local to the phase or state subroutine. It exists only while the phase or state subroutine is active (running or holding). If the call is in a data section, then that instance of the monitor subroutine is global and exists all the while the program is active (running or holding).

Each instance of a monitor subroutine has an execution status - running or stopped. The active phase or state subroutine may stop or start each instance of a monitor subroutine. Once a global instance of a monitor subroutine is started, it will continue to run until commanded to stop by the active phase or state subroutine or the program completes. It may run across operations and will run even while the active phase or state subroutine is holding.

Once a local (instance of a) monitor subroutine is started, it will continue to run until one of the following occurs:

- It is commanded to stop by the active phase or state subroutine.
- The operation ends.

MONITOR *(continued)*

subroutine

- The status of the active phase or state subroutine changes from running to holding (that is, a local monitor subroutine does not run while the phase or state subroutine is holding). When the phase or state subroutine status changes back to running, the monitor subroutine will automatically resume running (if the monitor subroutine status was running).

If a monitor definition contains data declarations, then each instance of the monitor subroutine has its own unique data space for that data. Refer to Figure 4-5 for the basic structure of a monitor subroutine.

```
MONITOR name
    Parameter list
DECLARATIONS
    Local data
CONTINUOUS
    Executable statements
ENDSUBR  [name]
```

Figure 4-5. Monitor Subroutine Structure

HEADING: The monitor subroutine heading has the following format:

MONITOR *name*
type1 pname1
type2 pname2
•
•
•

Parameter	Description
name	Name of this subroutine.
typeN	Data type of the Nth formal parameter. Available data types are: **ANY** — Any data type or expression. **AOL** — Analog exception report. **APID** — Advanced PID controller. **ARRAY** — Any array type. **BBUF** — Boolean data buffer. **BLOCK** — Function block. **CS** — Control station block. **CSEQ** — Common sequence. **DATAEXPT** — User defined data export block. **DD** — Device driver block.

MONITOR *(continued)*

subroutine

Parameter	Description	
typeN (continued)	**DOL**	Digital exception report.
	FGEN	Function generator.
	INTEGRATOR	Integrator.
	MSDD	Multi-state device driver block.
	RAMP	Ramp.
	RBUF	Real data buffer.
	RCM	Remote control memory.
	REMSET	Remote manual set constant.
	RMC	Remote motor control block.
	SMITH	Smith controller block.
	TEXT	Text block.
	TIMER	Timer.
	VAR	Variable.
pname	Local name assigned to the nth formal parameter.	

Example:

```
MONITOR  CHK_TOTAL
    BLOCK  FLOW
    INTEGRATOR  AMOUNT
    VAR  LIMIT
```

REMARKS:

While there is no fixed limit on the number of parameters, the subroutine can have only one parameter per line. The parameter list is terminated by the start of the next section.

When a formal parameter is any array, the parameter specification must indicate the number of dimensions as follows:

VAR ARRAY *fnameN* (*....)
CONST ARRAY *fname* (*....)

Parameter	Description
(*....)	Number of dimensions. (that is, (*), (*, *) or (*, *, *))

PARAMETER REFERENCES:

The formal parameter list defines the local parameter names (those used inside the monitor subroutine). Each formal parameter can be thought of as a place holder for an actual parameter. The actual parameters are specified when the monitor subroutine is declared (either in a data section or in a step or state subroutine). Each actual parameter refers to a value (type ANY) or to some object of data (for example, timer, variable, function block). When the monitor subroutine is called, each reference to a formal parameter becomes a reference to the corresponding actual parameter.

MONITOR *(continued)*

subroutine

Example: Consider the following monitor subroutine:

```
MONITOR  CHK_TIMER
TIMER        TIM
VAR          LIMIT
VAR          STATUS
CONTINUOUS

IF  (TIM.VAL  <  LIMIT)  THEN
    STATUS  =  0
ELSE
    STATUS = 1
    STOP  TIM
ENDIF
ENDSUBR CHK_TIMER
```

Consider the following PHASE or STATE subroutine declaration:

```
DECLARATIONS:
    VAR  TIM_LIMIT
    VAR  TIM_STS
    TIMER  TIMER1  (SEC)
    MONITOR  CHK_TIMER  (TIMER1,  TIM_LIMIT,  TIM_STS)  ALIAS  \
    CHK_TIME1
```

DECLARATIONS: The declarations statement marks the beginning of the declaration section of the monitor subroutine. This section specifies data local to the monitor subroutine.

DECLARATIONS
type1 *name1*
•
•
•
typeN *nameN*

Parameter	Description
nameN	Name of Nth data object.
typeN	Data type of the Nth data object. Available data types are: **CONST** Constant. **CONST ARRAY** Constant array. **CONST STRING** Constant string. **CONST STRING ARRAY** Constant string array. **BLOCK ARRAY** Function block array. **VAR** Variable. **VAR ARRAY** Variable array. **VAR STRING** Variable string.

subroutine

Parameter	Description	
typeN (continued)	**VAR STRING ARRAY**	Variable string array.

Example:

```
DECLARATIONS
    VAR  COUNTER
    CONST  INDEX  =  1.0
```

The monitor local data exists for the entire length of time the monitor subroutine exists. Starting and stopping a monitor subroutine does not effect its local data.

REFERENCES:

Get monitor status *name***.STS**

Parameter	Description
name	Name or ALIAS of the monitor call.
*name***.STS**	Monitor status: 0 = stopped. 1 = running.

Example: X = MON_LEVEL1.STS

SELF-REFERENCING:

A self-referencing feature allows a monitor subroutine to stop its own execution. For example:

```
MONITOR  monitor_name
    CONTINUOUS
        statement1
            •
            •
            •
        HOLD  monitor_name
ENDSUBR  monitor_name
```

Any logic statements appearing after an unconditional **HOLD monitor_name** statement will never be executed and shall marked as such by the compiler.

MONITOR

declaration

PURPOSE: The monitor declaration is used to specify a call to a monitor subroutine. Each declaration represents a separate call and has a parameter list that specifies the actual parameters that are passed to the monitor subroutine. This parameter list must match the formal parameter list of the specified monitor subroutine in number and types of parameters.

LANGUAGE:

BATCH 90	UDF

DECLARATION: **MONITOR** *name* **(***param1*, *param2*, ...*paramN***)** *init*[**ALIAS** *name2*]

Parameter	Description
name	Name of the monitor subroutine to be called. The specified monitor subroutine must already be defined.
paramN	Nth actual parameter. Each parameter must specify a data object which has already been declared in local or batch data.
init	Initial status (running or stopped). If no status is specified, running is assumed.
name2	Name used to identify this call to a monitor subroutine.

Examples:

```
MONITOR STATUS (FV211, FV215, FC534) RUNNING
MONITOR MON_LEVEL (TMR501, LIC501) STOPPED \
ALIAS MON_LEVEL_1
```

COMMANDS: The following commands control the execution status of monitor subroutines.

Start **START** *name*

Parameter	Description
name	Name of the monitor subroutine or name of alias.

Example:

```
START MON_LEVEL
```

NOTE: This command changes the monitor subroutines status from stopped to running (the command has no effect if the status is already running). This command simply changes the status of the monitor subroutine - it does not immediately execute the monitor subroutine. The monitor subroutine will execute during the next function block cycle.

MONITOR *(continued)*

declaration

Hold **HOLD** *name*

Parameter	Description
name	Name of the monitor subroutine or name of alias.

Example: HOLD MON_LEVEL

NOTE: This command changes the status from running to stopped. It has no effect if the status is stopped. A monitor can set itself to the hold condition.

MULTI-STATE DEVICE DRIVER

function block

PURPOSE:

Multi-state device drivers (function code 129) are used to control devices with up to four states, three outputs, and four feedbacks. Applications of this function block are multi-state motors (fast, slow and stopped), valves with bidirectional motors (forward, reverse and stop states), etc. As with the device driver, function code 129 does the low level interlocking (confirmation of state within feedback time).

LANGUAGE:

BATCH 90	UDF

DECLARATION:

A multi-state device driver (MSDD) function block must be declared before it can be accessed from a subroutine. The declaration must appear in the data section of a program. Batch 90 and UDF languages can only manipulate multi-state device drivers in which specifications S1 and S2 are set to zero.

BLOCK *msddname*, **BLK** = *const*, **FC** = **MSDD** [, **WATCH**]

Parameter	Description
msddname	Name of the MSDD function block.
const	Function block address.
WATCH (Batch 90 only)	Specifies that the multi-state device driver is to be placed on watch upon program start-up. If WATCH is not specified, the multi-state device driver will be off watch.

Example: BLOCK FAN, BLK = 100, FC = MSDD

REFERENCES:

Multi-state device driver data may be read by using the appropriate name in an expression.

Get control output *msddname*.**CO**

Parameter	Description
msddname	Name of the MSDD function block.
msddname.**CO**	Control output state (0 - 3).

Example: IF (FV501.CO IS 2) THEN

MULTI-STATE DEVICE DRIVER *(continued)*

function block

Get status *msddname*.**STS**

Parameter	Description
msddname	Name of the MSDD function block.
msddname.**STS**	Status: 0 =good. 1 = bad. 2 = waiting.

Example: IF (FV501.STS IS BAD) THEN

Get mode *msddname*.**MODE**

Parameter	Description
msddname	Name of the MSDD function block.
msddname.**MODE**	Mode: 0 = manual. 1 = auto.

Example: IF (FV501.MODE IS 0) THEN

Get feedback values *msddname*.**FBV**

Parameter	Description
msddname	Name of the MSDD function block.
msddname.**FBV**	Feedback values.

Examples: IF (FV501.FBV = 1001) THEN
IF (DIGIT (FV501.FBV,2)) = 1 THEN

NOTE: The four feedback values are encoded into a single 4-digit number with each digit representing an individual feedback value. The most significant digit is associated with feedback 1, the next digit with feedback 2 and so on.

Get feedback targets *msddname*.**FBT**

Parameter	Description
msddname	Name of the MSDD function block.
msddname.**FBT**	Feedback target values.

Example: IF (FB501.FBV = FB501.FBT) THEN

NOTE: The four target values are encoded into a single 4-digit number with each digit representing an individual target value. The most significant digit is associated with target 1, the next digit with target 2 and so on.

MULTI-STATE DEVICE DRIVER (continued)

function block

COMMANDS: The following commands are used to control multi-state device drivers.

Set control output

SET *msddname***.CO = ***exp* {No Wait}

- or -

SET AND WAIT *msddname***.CO = ***exp* {Wait for confirmation}

Parameter	Description
msddname	Name of the MSDD function block.
exp	Control output value (0 - 3).

Example: SET FV501.CO = 2

NOTE: The operation of this command depends on the mode and tracking status of the function block.

1. If the mode is auto, the control output state is set to exp.

2. If the mode is not auto and tracking is disabled, exp is recorded inside the function block, but the control output state is not updated. If the mode is later changed to auto, then the control output state is updated.

3. If the mode is not auto and tracking is enabled, the command has no effect.

The SET AND WAIT version of this command performs this operation and then waits for confirmation (that is, the control output state is equal to exp and the feedback status is good). The command retries the operation (once per cycle while waiting. Confirmation does not require that the mode is auto. It requires that only the feedback status is good and the control output is in the specified state.

The SET version performs the same operation but does not wait.

Set mode **SET** *msddname***.MODE = ***exp*

Parameter	Description
msddname	Name of the MSDD function block.
exp	Mode: 0 = manual. 1 = auto.

Example: SET FV501.MODE = AUTO

Set status override **SET** *msddname***.OVR = ***exp*

Parameter	Description
msddname	Name of the MSDD function block.
exp	Override status: 0 = off. 1 = on.

Example: SET FV501.OVR = ON

Set control output track flag **SET** *msddname***.TRK = ***exp*

Parameter	Description
msddname	Name of the MSDD function block.
*msddname***.TRK**	Control output tracking flag: 0 = disable. 1 = enable.

Example: SET FV501.TRK = 1

NOTES:

1. The purpose of control output tracking is to provide a bumpless transfer of the control output during a mode change from manual to auto. In manual mode, the operator controls the control output value. If tracking is disabled and the mode is changed from manual to auto, the control output will go to the last value that was written to the block by the program (that is, the control output may bump). If tracking is enabled, the control output value will remain unchanged during the mode change.

2. The initial value of tracking status (when the MFP module is placed into the execute mode) is off.

MULTI-STATE DEVICE DRIVER (continued)

function block

Set watch status (Batch 90 only)

SET *msddname*.**WATCH = ** *exp*

Parameter	Description
msddname	Name of the MSDD function block.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET FV501.WATCH = exp

REMARKS: These commands are buffered within the function block and do not have an immediate effect on the function block. The command will not take effect until the function block executes on its next cycle.

NEXT OPERATION

statement

PURPOSE: This statement terminates the current operation and activates another operation.

LANGUAGE:

BATCH 90	UDF

FORMAT: **NEXT OPERATION** [*exp*]

Parameter	Description
exp	Number of operation to be activated. If exp is not specified, then the next sequential operation is activated (that is. current phase + 1).

Examples:

```
NEXT OPERATION
NEXT OPERATION 10
```

REMARKS: If the value of the exp is greater than the number of operations called for the unit recipe, then the unit recipe is complete (that is, a done statement will be executed).

NOTES:

1. If the value of the exp is equal to the current operation number, then control will transfer to the top of continuous logic. If the next operation statement is in continuous logic it will prevent normal logic from being executed. This has the effect of freezing the program.

2. NEXT OPERATION is an ISA-S88.01-1995 standard term. Usage of this term prohibits compiling with Elsag Bailey specific terms. The Elsag Bailey specific term is NEXT PHASE. To use the Elsag Bailey specific term, replace OPERATION with PHASE.

NEXT PHASE

statement

NOTE: NEXT PHASE is an Elsag Bailey specific term. Usage of this term prohibits compiling to ISA-S88.01-1995 standards. The ISA-S88.01-1995 standard term is NEXT OPERATION. To use the Elsag Bailey specific term, refer to ***NEXT OPERATION*** and replace OPERATION with PHASE.

NEXT STATE

statement

PURPOSE: This statement terminates the current state subroutine and activates another state subroutine.

LANGUAGE:

BATCH 90	UDF

FORMAT: **NEXT STATE** *name*

Parameter	Description
name	Name of the state subroutine to be activated.

Example: NEXT STATE MOTORSTOP

#NOLIST

directive

PURPOSE: The #NOLIST directive turns off the generation of the listing file. This directive is useful if long #INCLUDE files are being used in a number of programs and their listings were unnecessary.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#NOLIST**

NOTE: The listing will remain off until a #LIST directive is seen. Any errors detected by the compiler are still displayed.

NORMAL LOGIC

section

PURPOSE: This statement marks the beginning of a phase (Batch90) or state (UDF) subroutine's normal logic section. This section provides the normal sequential control of a phase subroutine (Batch 90) or state subroutine (UDF).

LANGUAGE:

BATCH 90	UDF

FORMAT:

NORMAL LOGIC

statement

•

•

•

statement

REMARKS: This section executes when the phase or state subroutine is active and the operation is in the normal mode. When the phase or state subroutine is first activated, the normal logic statement begins executing at the first statement. It continues to execute statement by statement until either the last statement or a next operation statement is executed or the mode is changed from normal to fault or hold (refer to Appendix A for details for parallel operation).

Example:

```
NORMAL  LOGIC
     START  TIMER1
     SET  FIC501.CO  =  OPEN
     WAIT  UNTIL  ((LT501  >=  45.0)  OR  (TIMER1.VAL  >=  30))
     SET  FIC501.CO  =  CLOSED
```

NORMALIZE STRING

command

PURPOSE: This command removes all leading and trailing blank spaces from a variable string. Multiple spaces between characters and tabs will be replaced with one space. The resultant string is stored in the original variable string.

LANGUAGE:

BATCH 90	UDF

FORMAT: **NORMALIZE (***vstrname***)**

Parameter	Description
vstrname	Name of a variable string.

Example: NORMALIZE (REQUIRED_ACTION)

OPERATION

variable

PURPOSE: OPERATION is a built-in status variable. OPERATION indicates the current operation number being executed by the batch sequence block.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: The operation number may be read by using OPERATION in an expression.

Example:

```
IF (OPERATION > 0) THEN
```

REMARKS: OPERATION may be used to determine the current operation number being executed. For example, if the batch sequence function block is executing phase subroutines in parallel (that is, 3.1, 3.2, 3.3, etc.) OPERATION returns only the whole number portion of the operation number (OPERATION = 3). Refer to the ***PHASE NUMBER*** statement.

NOTE: OPERATION is an ISA-S88.01-1995 standard term. Usage of this term prohibits compiling with Elsag Bailey specific terms. The Elsag Bailey specific term is PHASE. To use the Elsag Bailey specific term, replace OPERATION with PHASE.

OPERATION DESCRIPTOR

function

PURPOSE: To retrieve the operation descriptor from the current unit recipe executing in a BSEQ function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from a Batch 90 program}
OPERATION DESCRIPTOR (*exp*, *vstring***)**

{from a UDF or other Batch 90 program}
OPERATION DESCRIPTOR (*bseqx*, *exp*, *vstring***)**

Parameter	Description
exp	Number of the operation from which the descriptor is desired.
vstring	Variable string (capable of storing at least 64 characters) used to return the specified descriptor.
bseqx	Name of the BSEQ function block declaration.

Example:

```
O_NUM = -1
REPEAT
    O_NUM = O_NUM + 1
    OPERATION_DESCRIPTOR (O_NUM, O_DESCS (O_NUM))
UNTIL ((O_NUM >= LAST OPERATION) OR (ONUM > 19))
```

REMARKS: The OPERATION DESCRIPTOR is entered using the unit procedure editor (UPE) or the optional master recipe editor (MRE). Each operation within a unit recipe has a descriptor that can be up to 64 characters long.

> **NOTE:** OPERATION DESCRIPTOR is an ISA-S88.01-1995 standard term. Usage of this term prohibits compiling with Elsag Bailey specific terms. The Elsag Bailey specific term is PHASE DESCRIPTOR. To use the Elsag Bailey specific term, replace OPERATION with PHASE.

OUTPUT

declaration

PURPOSE: The OUTPUT declaration assigns a name to UDF Type 1, UDF Type 2, and AUXILIARY UDF function block outputs. The name specified in the declaration is used by UDF commands to write values to the function block output.

LANGUAGE:

BATCH 90	UDF

DECLARATION: An output statement is necessary to declare any output from the UDF declaration function block. An output statement defines a boolean or real value depending upon the UDF function block type. The declaration can appear only in the data section.

[**AUX** *x*] **OUTPUT** *y name*

Parameter	Description
AUX *x*	Name of the auxiliary UDF function block that connects to the UDF Type 1 or 2 function block.
y	Output of either the UDF declaration function block or auxiliary UDF function block.
name	Name for the output.

Examples:

```
AUX 1
    OUTPUT 2 PV501
    OUTPUT 3 PV503

AUX 2
    OUTPUT 1 FV40
    OUTPUT 2 FV42
```

COMMANDS: The SET command is used to write a value to the function block output.

Set output value

SET *name***.VAL** **=** *exp*

Parameter	Description
name	UDF function block output name.
exp	Output value.

Examples:

```
SET PV501.VAL = 75.5
SET FV42.VAL = ON
```

PHASE

statement

NOTE: PHASE is an Elsag Bailey specific term. Usage of this term prohibits compiling to ISA-S88.01-1995 standards. The ISA-S88.01-1995 standard term is OPERATION. To use the Elsag Bailey specific term, refer to ***OPERATION*** and replace OPERATION with PHASE.

PHASE DESCRIPTOR

function

NOTE: PHASE DESCRIPTOR is an Elsag Bailey specific term. Usage of this term prohibits compiling to ISA-S88.01-1995 standards. The ISA-S88.01-1995 standard term is OPERATION DESCRIPTOR. To use the Elsag Bailey specific term, refer to ***OPERATION DESCRIPTOR*** and replace OPERATION with PHASE.

PHASE NUMBER

variable

PURPOSE: PHASE NUMBER is a built-in status variable. PHASE NUMBER indicates the phase number of the active phase subroutine.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Not required.

REFERENCES: The phase number may be read by using PHASE NUMBER in an expression.

Example:

```
IF (PHASE NUMBER > 1) THEN
```

REMARKS: The phase number is the decimal portion of the operation number. For example, if the second phase subroutine of operation 3 is active (3.2), then OPERATION is 3 and PHASE NUMBER is 2.

NOTE: PHASE NUMBER is an ISA-S88.01-1995 standard term. Usage of this term prohibits compiling with Elsag Bailey specific terms. The Elsag Bailey specific term is SUBPHASE. To use the Elsag Bailey specific term, replace PHASE NUMBER with SUBPHASE.

PHASE SUBR

subroutine

PURPOSE:

A phase subroutine is a major unit of a batch program. It is responsible for controlling the sequence of activities within a phase.

Each operation of a unit recipe specifies the names of the phase subroutines that are to be executed for that operation. The unit recipe also specifies a list of formulation data. When an operation becomes active, the specified phase subroutines are called and the formulation data is passed to them (via the parameter list of each phase subroutine). When the phase subroutines complete their activities, control is passed to the next operation.

There is no limit to the amount of times a phase subroutine may be called.

Figure 4-6 shows the general structure of a phase subroutine.

```
PHASE SUBR name
    [Phase comment]
    Parameter list
DECLARATIONS
    Local data and monitor subroutine calls
CONTINUOUS
    Executable statement
NORMAL LOGIC
    Executable statements
FAULT LOGIC
    Executable statements
HOLD LOGIC
    Executable statements
RESTART LOGIC
    Executable statements
ENDSUBR  [name]
```

Figure 4-6. Phase Subroutine Structure

LANGUAGE:

BATCH 90	UDF

HEADING:

A phase subroutine heading defines the name of the subroutine and its formal parameters. When a phase subroutine is called, a list of formulation data is passed from the unit recipe to the phase subroutine. The phase comment is an optional 1400 character field that can be viewed as help information from the recipe editor. The parameter list defines the names to

PHASE SUBR *(continued)*

subroutine

be used within the phase subroutine for the individual items of formulation data. The heading format is:

PHASE SUBR *name*
type1 pname1
type2 pname2
•
•
•

Parameter	Description
name	Name of this subroutine. The name must be unique within the batch program. The maximum length is 16 alphanumeric characters. Embedded spaces are not allowed.
typeN	Data type of the Nth parameter. Data type must be one of the following: **ANY** Any data type or extension. **AOL** Analog exception report. **APID** Advanced PID controller. **BBUF** Boolean data buffer. **BLOCK** Function block. **CS** Control station block. **CSEQ** Common sequence. **DATAEXPT** User defined export block. **DD** Device driver block. **DOL** Digital exception report. **MSDD** Multi-state device driver block. **RBUF** Real data buffer. **RCM** Remote control memory. **REMSET** Remote manual set constant. **RMC** Remote motor control block. **SMITH** Smith controller block. **STRING** Recipe string argument. **TEXT** Text block.
pnameN	Local name assigned to the nth parameter. The Nth parameter corresponds to the nth item in the unit procedures list of formulation data. The name is restricted to a maximum of 32 characters.

Example:

PHASE SUBR ADD_COMPONENT_A
ANY RATE_A = (100<500<1000)
ANY TOTAL_A = (MIN_CHRG, NORMAL_CHRG, MAX_CHRG)
DD VALVE_A = (PV51, FV52)
STRING INAME = “COMMENT ID #XXX”

REMARKS:

1. Parameter list is terminated by the start of the next section (that is, declarations).

2. There can be no more than 100 parameters.

3. Parameters with data type ANY or BLOCK can only be read by the subroutine. They cannot be modified.

ARGUMENT INITIALIZATION:

The formal arguments to a phase subroutine cab be given initial values. These values are used as the default value by the unit procedure editor when preparing a unit procedure operation for this phase subroutine.

FORMAT:

For an argument of type ANY the following combination is allowed:

ANY *pnameN* **=** **(***const1* < *const2* < *const3***)**

Parameter	Description
const1	Default low limit.
const2	Default value.
const3	Default high limit.

Any combination is allowed (1 < 2 < 3).

(1) - Low only
(< 2) - Value only.
(< < 3) - High only.
(1 < < 3) - Low and high.
(1 < 2 < 3) - Low, value, and high.

Refer to ***#HIHI*** for low-low and hi-hi parameter details. For function block arguments, it is possible to specify a function block name or number of the proper type.

Example:

```
DD D1 = (DD_1)
RBUF B1 = (127)
```

ARGUMENT SELECTION LIST:

For an argument of data type ANY a list of allowable entries will be defined. When preparing a unit procedure, the unit procedure editor presents a list of the only valid inputs for the argument.

FORMAT:

ANY *pnameN* **=** **(***const1*, *const2* [...*constN*]**)**

Parameter	Description
constN	Name of a constant declared in the batch data section of the program, or a Batch 90 reserved constant.

Example:

```
ANY COLOR_CHOICE = (RED, BLUE, MAGENTA)
```

PHASE SUBR (continued)

subroutine

HIGH AND LOW LIMITS: The .HLIM and .LLIM reference extensions allow a program to read the high and low limits of recipe ANY parameters.

REFERENCES:

Get low limit *pname*.**LLIM**

Parameter	Description
pname	Name of the parameter.
.LLIM	Low limit value.

Get high limit *pname*.**HLIM**

Parameter	Description
pname	Name of the parameter.
.HLIM	High limit value.

Example:

```
PHASE  SUBR  EXAMPLE_SUBR
     ANY TGT_RATE
NORMAL_LOGIC
     SET  SETPOINT_MIN.VAL  =  TGT_RATE.LLIM
     SET  SETPOINT_MAX.VAL  =  TGT_RATE.HLIM
          •
          •
          •
ENDSUBR
```

REMARKS: The values returned by .LLIM and .HLIM are the low limit and high limit values specified by the active unit procedure.

DECLARATIONS: This section specifies local data and calls to monitor subroutines. The format is:

type1   name1
•
•
•
typeN   nameN

PHASE SUBR *(continued)*

subroutine

Parameter	Description
type	May be any one of the following: **ARRAY** All types. **CONST** Constant. **CONST STRING** Constant string. **FGEN** Function generator. **INTEGRATOR** Integrator. **MONITOR** Call to monitor. **RAMP** Ramp. **TIMER** Timer. **VAR** Variable. **VAR STRING** Variable string.
name	Local name assigned to the nth parameter. The nth parameter corresponds to the nth item in the unit procedures list of formulation data. The name is restricted to a maximum of 32 characters.

Example:

```
DECLARATIONS
    VAR  COUNTER
    ONST  MIN_LIMIT  =  3.0
    TIMER  TMR  (MIN)
    INTEGRATOR  INTG1
    FGEN  PROFILE1  ((0.0,  80.0),  (30.0,  100.0),  (60.0,  80.9))
    RAMP  LEVEL  RAMP  (LIC501.SP,  SEC)
    MONITOR MON1  (FC211,  FC215,  FC534)
```

NOTES:

1. The phase subroutines local data exists only while the operation is active.

2. PHASE SUBR is an ISA-S88.01-1995 standard term. Usage of this term prohibits compiling with Elsag Bailey specific terms. The Elsag Bailey specific term is STEP SUBR. To use the Elsag Bailey specific term, replace PHASE with STEP.

PROGRAM DESCRIPTOR

function

PURPOSE: To retrieve the program descriptor from the current Batch 90 program executing in a BSEQ function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from a Batch 90 program}
PROGRAM DESCRIPTOR (*vstring***)**

{from a UDF program}
PROGRAM DESCRIPTOR (*bseqx***,** *vstring***)**

Parameter	Description
vstring	Name of the variable string (capable of storing at least 64 characters) used to return the specified descriptor.
bseqx	Name of the BSEQ function block declaration.

RAMP DATA

data structure

PURPOSE: The ramp is an active data structure that changes a control station set point at a uniform rate over time.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The ramp declaration format is:

RAMP *rname* **(***vname*, *units***)**

Parameter	Description
rname	Name of the ramp.
vname	Name of the value to be ramped. This value must be one of the following types: AOL.VAL CS.CO CS.SP REMSET.VAL RBUF.VAL VAR (or an array element of one of the above types)
units	Units of time: **SEC** Seconds. **MIN** Minutes. **HOUR** Hours.

Examples:

RAMP TEMP_RAMP (TIC501.SP, MIN){station SP}
RAMP LEVEL_RAMP (LIC501.SP, SEC){station CO}

REFERENCE: Ramp data may be read by using the appropriate name in an expression.

Get value *rname***.VAL**

Parameter	Description
rname	Name of the ramp.
*rname***.VAL**	Current value.

Example: X = TEMP_RAMP.VAL

RAMP DATA (continued)

data structure

Get status *rname***.STS**

Parameter	Description
rname	Name of the ramp.
*rname***.STS**	Current status: 0 = holding. 1 = running.

Example: IF (TEMP_RAMP.STS IS RUNNING) THEN

Get upper limit *rname***.LIM**

Parameter	Description
rname	Name of the ramp.
*rname***.LIM**	Upper limit.

Example: IF (TEMP_RAMP.LIM > 150) THEN

Get rate *rname***.RATE**

Parameter	Description
rname	Name of the ramp.
*rname***.RATE**	Rate.

Example: IF (TEMP_RAMP.RATE > 2.5) THEN

COMMANDS:

Start **START** *rname* **FROM** *exp1* **TO** *exp2* **AT** *exp3*

Parameter	Description
rname	Name of the ramp.
exp1	Initial ramp value.
exp2	Upper limit of ramp.
exp3	Rate of change.

This command changes the status from holding to running; has no effect if status is running.

Example: START LEVEL_RAMP FROM 10.0 TO 15.8 AT 0.5

NOTE: If the ramp rate is positive (that is, from a low initial value to a higher final value) then the rate (exp3) must reflect this by being a positive number. Similarly, if the ramp is negative (that is, from a high initial value to a lower final value) then the rate (exp3) must be negative.

data structure

Hold **HOLD** *rname*

Parameter	Description
rname	Name of the ramp.

Example: HOLD LEVEL_RAMP

This command changes the status from running to holding; has no effect if status is not running.

Resume ramp **RESUME** *rname*

Parameter	Description
rname	Name of the ramp.

Example: RESUME LEVEL_RAMP

This command changes the status from holding to running; has no effect if status is not holding.

RBUF

Refer to ***DATA BUFFER****.*

RECIPE DESCRIPTOR

function

PURPOSE:

To retrieve the unit recipe descriptor from the current unit recipe executing in a BSEQ function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from a Batch 90 program}
RECIPE DESCRIPTOR (*const*, *vstring***)**

{from a UDF program}
RECIPE DESCRIPTOR (*bseqx*, *const*, *vstring***)**

Parameter	Description
const	Line type of the description to be returned. Descriptions that can be returned are: 1 = unit procedure description. 2 = unit description. 3 = equipment list description. 4 = master recipe description. 5 = train description.
vstring	Name of the variable string (capable of storing at least 64 characters) used to return the specified descriptor.
bseqx	Name of the BSEQ function block declaration.

Examples:

RECIPE DESCRIPTOR = (1, REC_DESC_STRING)
RECIPE DESCRIPTOR = (3, EQ_LIST_STRING)

RECIPE ENTRY

function

PURPOSE: To read descriptors from files in the Batch\UDF directory of the MFP module. This function enables a Batch 90 program to read all of the files contained in the Batch\UDF file directory and return their file ID and descriptor.

LANGUAGE: BATCH 90 | UDF

DECLARATION: **RECIPE ENTRY (***const1, exp1, var1, vstring, var2***)**

Parameter	Description
const1	Line type of the description to be returned. Descriptions that can be returned are: 1 = unit recipe description. 2 = unit description. 3 = equipment list description. 4 = master recipe description. 5 = train description.
exp1	The address of the BSEQ function block that contains the specified unit recipe descriptors.
var1	Numeric variable used to track position in the batch directory. The value of the variable at the beginning of execution specifies the starting position for the directory search. Upon completion, the command will return a: *x* = current directory position. 0 = end of the batch directory. -1 = file error.
vstring	Name of the variable string (capable of storing at least 64 characters) used to return the specified descriptor.
var2	Numeric variable used by the command to return the unit procedure ID associated with the unit procedure file selected by the search.

Example: Read the unit recipe descriptors for the first ten unit recipes associated with function block 100. RDESCS is a variable string array of ten elements and RIDS is a numeric variable array of ten elements.

```
FUNCTION  GET_DESCRS
DECLARATIONS
     VAR  NDEX
     VAR  D_POSN
     VAR  STRING  ARRAY  RDESCS  (1:10)
     VAR  ARRAY  RIDS  (1:10)
EXECUTABLE
     BNUM  =  100
     NDEX  =  0
```

***RECIPE ENTRY** (continued)*

function

```
    D_POSN = 0
    REPEAT
        D_POSN = D_POSN + 1
        NDEX = NDEX + 1
        RECIPE ENTRY (1, B_NUM, D_POSN, \
            RDESCS (NDEX), RIDS (NDEX))
    UNTIL ((D_POSN <= 0) OR (NDEX > 10))
ENDSUBR GET_DESCRS
```

REMOTE CONTROL MEMORY

function block

PURPOSE: The remote control memory (RCM) function block (function code 62) acts as a remotely operated switch (controlled by an OIS console).

LANGUAGE:

BATCH 90	UDF

DECLARATION: The RCM function block must be declared before it can be accessed from a subroutine. The declaration can appear only in the data section.

BLOCK *name*, **BLK =** *const*, **FC = RCM** [**, WATCH**]

Parameter	Description
name	Name of the RCM function block.
const	Function block address.
WATCH (Batch 90 only)	The RCM function block to be placed on WATCH by the Batch Historian upon program start-up. If not specified, the RCM function block will be off watch.

Example: BLOCK RCM501, BLK = 100, FC = RCM

REFERENCES: RCM data may be read by using the appropriate name in an expression.

Get output value *name***.VAL**

Parameter	Description
name	Name of the RCM function block.
*name***.VAL**	Output value.

Example: IF (RCM501.VAL=ON) THEN

COMMANDS:

Set output **SET** *name***.VAL =** *exp*

Parameter	Description
name	Name of the RCM function block.
*name***.VAL**	Output value.
exp	Logical expression.

Examples:

SET RCM501.VAL = ON
SET RCM501.VAL = 1

REMOTE CONTROL MEMORY *(continued)*

function block

Set watch status (Batch 90 only)

SET *rcmname***.WATCH =** *exp*

Parameter	Description
rcmname	Name of the RCM function block.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET RCM501.WATCH = ON

REMARKS:

The RCM function block treats these commands as if they were issued by an OIS console. For example, the command is ignored if the function blocks permissive input does not allow the command. These commands are buffered within the function block and do not have an immediate effect on the function block. The command will not take effect until the function block executes on its next cycle.

REMOTE MANUAL SET CONSTANT

function block

PURPOSE: A remote manual set constant (REMSET) function block (function code 68) provides a way for an operator to set a constant in an MFP module (via the OIS console).

LANGUAGE:

BATCH 90	UDF

DECLARATION: The REMSET function block must be declared before it can be accessed from a subroutine. The declaration can appear only in the data section.

BLOCK *name***, BLK =** *const***, FC = REMSET**[**, WATCH**]

Parameter	Description
name	Name of the REMSET function block.
const	Function block address.
WATCH (Batch 90 only)	Specifies that the REMSET function block is to be placed on watch by the Batch Historian upon program start-up. If not specified, the REMSET function block will be off watch.

Example: BLOCK RSET501, BLK = 100, FC = REMSET

REFERENCES: REMSET data may be read by using the appropriate name in an expression.

Get output value *name***.VAL**

Parameter	Description
name	Name of the REMSET function block.
*name***.VAL**	Output value.

Example: X = RSET501.VAL

COMMANDS:

Set output **SET** *name***.VAL** = *exp*

Parameter	Description
name	Name of the REMSET function block.
exp	Value.

Example: SET RSET501.VAL = 100

REMOTE MANUAL SET CONTANT (continued)

function block

Set watch status (Batch 90 only)

SET *name***.WATCH** = *exp*

Parameter	Description
name	Name of the REMSET function block.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET REMSET501.WATCH = ON

REMARKS:

The REMSET function block treats the set output command as if it were issued by an OIS console. For example, the command is ignored if the function block is in track mode. This command is buffered within the function block and does not have an immediate effect on the function block. The command will not take effect until the function block executes on its next cycle.

REMOTE MOTOR CONTROL

function block

PURPOSE: The remote motor control (RMC) function block (function code 136) is used by an operator or a batch program to control a motor.

LANGUAGE:

BATCH 90	UDF

DECLARATION: Specifications S1 and S2 of an RMC function block determine whether its control output is controlled by the batch program or by function blocks. If both specifications are set to two, the batch program has control.

BLOCK *name*, **BLK** = *const*, **FC** = **RMC**[, **WATCH**]

Parameter	Description
name	Name of the RMC function block.
const	Function block address.
WATCH (Batch 90 only)	Specifies that the RMC function block is to be placed on watch by the Batch Historian upon program start up. If not specified, the RMC function block will be off watch.

Example: BLOCK MOTOR501, BLK = 240, FC = RMC

REFERENCES: RMC data may be read by using the appropriate name in an expression.

Get control output *name*.**CO**

Parameter	Description
name	Name of the RMC function block.
name.**CO**	Control output value (0, 1).

Example: X = MOTOR501.STS

Get control output status *name*.**STS**

Parameter	Description
name	Name of the RMC function block.
name.**STS**	Control output status: 0 = good. 1 = bad. 2 = waiting.

Example: X = MOTOR501.STS

REMOTE MOTOR CONTROL *(continued)*

function block

Get feedback values

name.**FB***n*

Parameter	Description
name	Name of the RMC function block.
n	Feedback number (1 or 2).
name.**FB***n*	Feedback value.

Examples:

X = MOTOR501.FB1
Y = MOTOR501.FB2

Get interlock values

name.**ILK***n*

Parameter	Description
name	Name of the RMC function block.
n	Interlock number (1, 2, 3, or 4).
name.**ILK***n*	Interlock value.

Examples:

X = MOTOR501.ILK1
Y = MOTOR501.ILK4

Get permissive values

name.**PERM***n*

Parameter	Description
name	Name of the RMC function block.
n	Permissive number (1 or 2).
name.**PERM***n*	Permissive value.

Examples:

X = MOTOR501.PERM1
Y = MOTOR501.PERM2

COMMANDS:

The following statement are used to control RMC function blocks.

Set control output

SET *name*.**CO = ** *exp*

- or -

SET AND WAIT *name*.**CO = ** *exp*

Parameter	Description
name	Name of the RMC function block.
exp	Control output value.

Example:

SET MOTOR501.CO = ON

REMOTE MOTOR CONTROL *(continued)*

function block

Set watch status (Batch 90 only)

SET *name***.WATCH = ** *exp*

Parameter	Description
name	Name of the RMC function block.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET RMC501.WATCH = ON

REMARKS:

The first form of the statement (SET) writes the control output value to the block and continues on to the next statement. The second form (SET AND WAIT) writes the control outputs value to the function block and then waits for confirmation (that is, the function blocks control output value is the same as exp and the control output status is good). The statement retries the operation once per function block cycle while waiting.

REPEAT...UNTIL

control structure

PURPOSE: This statement continuously executes a block of statements until some condition becomes true.

LANGUAGE:

BATCH 90	UDF

FORMAT:

REPEAT
statement1
•
•
•
statementN
UNTIL (*exp*)

Example:

```
START  TIMER1
    REPEAT
       •
       •
       •
UNTIL  (TIMER1.VAL  >=  30.0)
```

REMARKS: Statements 1 through N are executed. If exp is FALSE, statements 1 through N are executed and exp is evaluated again. This continues until exp becomes true. If exp is TRUE, execution continues at statement following UNTIL.

A repeat loop can be terminated by a break statement.

The program is suspended for one cycle each time the exp is evaluated (including the first time). The suspension occurs before evaluating exp. This statement should not be used where execution is continuous (for example, in a monitor subroutine).

RESERVE

Refer to ***COMMON SEQUENCE****.*

RESET

*Refer to **FUNCTION GENERATOR**, **INTEGRATOR**, and **TIMER**.*

RESTART

statement

PURPOSE: This statement transfers control from the FAULT LOGIC to the RESTART LOGIC.

LANGUAGE:

BATCH 90	UDF

FORMAT: **RESTART**

Example: RESTART

REMARKS: If there is no RESTART LOGIC, control is transferred to the beginning of NORMAL LOGIC or the last executed restart point.

The RESTART statement can be used only in the FAULT LOGIC of a phase subroutine. It cannot be used in a function or monitor subroutine.

RESTART LOGIC

section

PURPOSE: This statement marks the beginning of the restart logic section of a phase subroutine. This section provides the sequential control of an operation during the transition from hold to normal.

LANGUAGE:

BATCH 90	UDF

FORMAT:

RESTART LOGIC
statement
•
•
•
statement

Example:

```
RESTART LOGIC
    SET FIC401.CO = OPEN
    SET FIC402.CO = CLOSED
RESUME AT MAJOR POINT
```

REMARKS: This section is activated when the batch sequence mode is changed from holding to normal. This section must contain at least one RESUME statement. When a RESUME statement is executed, control is transferred to the normal logic at the appropriate restart point.

When this section is first activated, fault mode is disabled. If desired, fault mode may be enabled anywhere in this section by an ENABLE FAULT CODE statement. Otherwise, fault mode is automatically enabled when control is transferred back to the normal logic.

RESTART POINT

statement

PURPOSE: The RESTART POINT statement marks a restart point in the normal logic of a phase subroutine.

LANGUAGE:

BATCH 90	UDF

FORMAT:

MAJOR RESTART POINT *id*

- or -

MINOR RESTART POINT *id*

Parameter	Description
id	Positive integer that uniquely identifies this point in normal logic.

Examples:

```
MAJOR RESTART POINT 102
MINOR RESTART POINT 3
```

REMARKS: This statement may be used only in the normal logic of a phase subroutine.

The normal logic may contain many MAJOR and MINOR RESTART POINT statements. However, at any given time, there is only one active major restart point and one active minor restart point.

When a phase subroutine is first activated, both active restart points are at the first statement of the normal logic. If a MAJOR RESTART POINT statement is executed, both active restart points move to that statement. If a minor restart point statement is executed, only the active minor restart point moves to that statement.

If the normal logic is interrupted (that is, the mode is changed to FAULT or HOLD), the current restart state is maintained. If the phase subroutine is restarted, then the restart logic may specify where to resume the normal logic based on the restart state (refer to the ***RESUME AT*** statement).

RESUME

Refer to ***RAMP DATA****.*

RESUME AT

statement

PURPOSE: This statement transfers control from the restart logic to the appropriate normal logic statement.

LANGUAGE:

BATCH 90	UDF

FORMAT: **RESUME AT MAJOR POINT**

- or -

RESUME AT MINOR POINT

- or -

RESUME AT *id*

Parameter	Description
id	An ID of a restart point statement.

Examples:

RESUME AT MAJOR POINT	{restart at active major restart point}
RESUME AT MINOR POINT	{restart at active minor restart point}
RESTART AT 101	{restart at restart point with *id* of 101}

REMARKS: If an *id* is specified, control is transferred to the normal logic restart point statement with the specified ID. Otherwise, control is transferred to the normal logic at the specified (major or minor) active restart point.

The RESUME AT statement can be used only in the restart logic of a phase subroutine. It cannot be used in any other phase, monitor, or function subroutine section.

RETURN

statement

PURPOSE: The RETURN statement (optionally) specifies a functions value and returns control from a function subroutine.

LANGUAGE:

BATCH 90	UDF

FORMAT: **RETURN** [*exp*]

Parameter	Description
exp	Value to be returned to caller.

Examples:

```
RETURN            {no value}
RETURN  (10)      {value is 10}
RETURN  (x + y)   {value is x + y}
```

REMARKS: When the RETURN statement executes, it terminates the function subroutine and returns control to the caller.

A function subroutine may contain more than one RETURN statement. If the last statement of a function is not a return then one is automatically generated by the compiler.

All RETURN statements in a function must be consistent in their use of return values. All must specify return values or none may (refer to ***FUNCTION*** subroutine).

SET

*Refer to **ADVANCED PID**, **ANALOG EXCEPTION REPORT**, **COMMON SEQUENCE**, **CONTROL STATION**, **DATA BUFFER**, **DEVICE DRIVER**, **DIGITAL EXCEPTION REPORT**, **MULTI-STATE DEVICE DRIVER**, **REMOTE CONTROL MEMORY**, **REMOTE MANUAL SET CONSTANT**, **REMOTE MOTOR CONTROL**, and **SMITH PREDICTOR**.*

SET AND WAIT

Refer to ***DEVICE DRIVER****,* ***MULTI-STATE DEVICE DRIVER****, and* ***REMOTE MOTOR CONTROL****.*

SIZEOF

function

PURPOSE: SIZEOF refers to the dimension size of an array. It returns the number of elements in the specified dimension of the array.

LANGUAGE:

BATCH 90	UDF

DECLARATION: None required.

REFERENCES:

Get total number of array elements

SIZEOF (*array_name***)**

Parameter	Description
array_name	Name of an array of any type.

Get number of elements of the nth dimension

SIZEOF (*array_name, n***)**

Parameter	Description
array_name	Name of an array of any type.
n	First, second, or third array dimension.

Example:

```
FUNCTION ARRAY_MANIPULATE
    VAR ARRAY SARRAY
DECLARATION
    VAR I, J, K, X, Y, Z
EXECUTABLE
X = SIZEOF (SARRAY, 1)
Y = SIZEOF (SARRAY, 2)
Z = SIZEOF (SARRAY, 3)
FOR I = 1 TO X DO
    FOR J = 1 TO Y DO
        FOR K = 0 TO Z DO
            •
            •
            •
        END FOR K
    END FOR J
END FOR I
ENDSUBR ARRAY_MANIPULATE
```

REMARKS: SIZEOF enables the programmer to write function subroutines that do array manipulations with arrays of variable dimensions.

SKIP CYCLE

statement

PURPOSE: The SKIP CYCLE statement causes a program section (for example, CONTINUOUS, NORMAL LOGIC) to terminate execution until the next cycle.

LANGUAGE:

BATCH 90	UDF

FORMAT: **SKIP CYCLE**

Example:

```
NORMAL  LOGIC
    SET  FC501.MODE  =  MANUAL
    SKIP  CYCLE
    SET  FC501.CO  =  0.0
```

REMARKS: When the SKIP CYCLE statement is executed, the section containing the statement terminates execution for the current cycle (that is, the remaining part of the section is skipped). When SKIP CYCLE is used in normal logic, it terminates the execution of the normal logic and returns to function block execution on the next cycle. Execution of normal logic returns to the executable statement after the SKIP CYCLE statement.

SKIP CYCLE should not be used in monitor subroutines.

SMITH PREDICTOR

function block

PURPOSE: The smith predictor function block (function code 160) provides a mechanism for the control of processes with dead time. The program interface to a smith predictor provides information about a process and its controller and allows changes to the tuning parameters.

LANGUAGE:

BATCH 90	UDF

DECLARATION: **BLOCK** *name*, **BLK = ** *const*, **FC = SMITH**

Parameter	Description
name	Name of the function block.
const	Function block address.

Example: BLOCK S501, BLK = 240, FC = SMITH

REFERENCES: Smith predictor data may be read by using the appropriate name in an expression.

Get process variable *name*.**PV**

Parameter	Description
name	Name of the function block.
name.**PV**	Value of process variable (specification S1).

Example: X = S501.PV

Get setpoint *name*.**SP**

Parameter	Description
name	Name of the function block.
name.**SP**	Value of set point (specification S2).

Example: X = S501.SP

SMITH PREDICTOR *(continued)*

function block

Get control output *name*.**CO**

Parameter	Description
name	Name of the function block.
name.**CO**	Value of control output.

Example: X = S501.CO

Get track status *name*.**TRK**

Parameter	Description
name	Name of the function block.
name.**TRK**	Tracking flag (specification S4): 0 = tracking. 1 = released.

Example: X = S501.TRK

Get gain parameter *name*.**GAIN**

Parameter	Description
name	Name of the function block.
name.**GAIN**	Gain (specification S7).

Example: X = S501.GAIN

Get dead time parameter *name*.**DTIM**

Parameter	Description
name	Name of the function block.
name.**DTIM**	Dead time (specification S8) in seconds.

Example: X = S501.DTIM

Get lag parameter *name*.**LAG**

Parameter	Description
name	Name of the function block.
name.**LAG**	Lag time (specification S9) in seconds.

Example: X = S501.LAG

SMITH PREDICTOR *(continued)*

function block

Get tuning time constant parameter *name*.**TUN**

Parameter	Description
name	Name of the function block.
name.**TUN**	Tuning time constant (specification S10) in seconds.

Example: X = S501.TUN

Get high limit parameter *name*.**HLIM**

Parameter	Description
name	Name of the function block.
name.**HLIM**	High limit (specification S11).

Example: X = S501.HLIM

Get low limit parameter *name*.**LLIM**

Parameter	Description
name	Name of the function block.
name.**LLIM**	Low limit (specification S12).

Example: X = S501.LLIM

COMMANDS: The following statements are used to tune smith predictor function blocks.

Set gain tuning parameter **SET** *name*.**GAIN** **=** *exp*

Parameter	Description
name	Name of the function block.
exp	Gain value.

Example: SET S501.GAIN = 2.0

SMITH PREDICTOR *(continued)*

function block

Set dead time parameter

SET *name*.**DTIM** **=** *exp*

Parameter	Description
name	Name of the function block.
exp	Dead time parameter.

Example: SET S501.DTIM = 30.0

Set lag parameter

SET *name*.**LAG** **=** *exp*

Parameter	Description
name	Name of the function block.
exp	Lag value.

Example: SET S501.LAG = 30.0

Set tuning time constant parameter

SET *name*.**TUN** **=** *exp*

Parameter	Description
name	Name of the function block.
exp	Tuning parameter value.

Example: SET S501.TUN = 30.0

START

*Refer to **COMMON SEQUENCE**, **FUNCTION GENERATOR**, **INTEGRATOR**, **MONITOR**, **RAMP DATA**, and **TIMER**.*

STATE SUBR

subroutine

PURPOSE:

A state subroutine is a major unit of an UDF program. It is responsible for controlling the sequence of activities within a UDF program. Figure 4-7 shows the general structure of a state subroutine.

```
STATE SUBR name [START [AND WAIT]]
DECLARATIONS
    Local data and monitor subr calls
CONTINUOUS
    Executable statement
NORMAL LOGIC
    Executable statements
FAULT LOGIC
    Executable statements
ENDSUBR [name]
```

Figure 4-7. State Subroutine Structure

LANGUAGE:

BATCH 90	UDF

HEADING:

A state subroutine heading defines the name of the subroutine. The heading format is:

STATE SUBR *name* [**START** [**AND WAIT**]]

Parameter	Description
name	Name of this subroutine. The name must be unique within the UDF program. The maximum length is 16 alphanumeric characters. Embedded spaces are not allowed.
START	Specifies that this subroutine will be executed first when the UDF program is started. Only one state subroutine may be designated as the starting point of the UDF program.
AND WAIT	Designates that the normal logic of the state subroutine will not execute until start-up of the multi-function processor module is complete.

Examples:

```
STATE SUBR INITIAL_SUBR START
STATE SUBR FIRST_SUB START AND WAIT
STATE SUBR MOTOR_STOP
```

STATE SUBR (continued)

subroutine

DECLARATIONS: The following specifies local data and calls to monitor subroutines for state subroutines:

type1 name1
•
•
•
typen namen

Parameter	Description
typen	Data type of the Nth parameter. Data type must be one of the following: **BLOCK ARRAY** Function block array. **CONST** Constant. **CONST STRING** Constant string. **CONST STRING ARRAY** Constant string array. **FGEN** Function generator. **INTEGRATOR** Integrator. **MONITOR** Monitor subroutine. **RAMP** Ramp. **TIMER** Timer. **TIMER ARRAY** Timer array. **VAR** Variable. **VAR ARRAY** Variable array. **VAR STRING** Variable string. **VAR STRING ARRAY** Variable string array.
namen	Local name of the nth parameter. The Nth parameter corresponds to the nth item in the recipes list of formulation data. The name is restricted to a maximum of 32 characters.

Example:

```
DECLARATIONS
    VAR  COUNTER
    CONST  MIN_LIMIT  =  3.0
    TIMER  TMR1  (MIN)
    INTEGRATOR  INTG1
    FGEN  PROFILE1  ((0.0,  80.0),  (30.0,  100.0),  (60.0,  80.0)))
    RAMP  LEVEL  RAMP  (LIC501.SP,  SEC)
    MONITOR  MON1  (FC211,  FC215,  FC534)
```

STEP SUBR

subroutine

NOTE: STEP SUBR is an Elsag Bailey specific term. Usage of this term prohibits compiling to ISA-S88.01-1995 standards. The ISA-S88.01-1995 standard term is PHASE SUBR. To use the Elsag Bailey specific term, refer to ***PHASE SUBR*** and replace PHASE with STEP.

SUBPHASE

variable

NOTE: SUBPHASE is an Elsag Bailey specific term. Usage of this term prohibits compiling to ISA-S88.01-1995 standards. The ISA-S88.01-1995 standard term is PHASE NUMBER. To use the Elsag Bailey specific term, refer to ***PHASE NUMBER*** and replace PHASE NUMBER with SUBPHASE.

SUBR DESCRIPTOR

function

PURPOSE:

To retrieve the phase subroutine descriptor of the currently active operation from the current unit recipe executing in a BSEQ function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from a Batch 90 program}

SUBR DESCRIPTOR (*exp***,** *vstring***)**

{from a UDF program}

SUBR DESCRIPTOR (*bseqx***,** *exp***,** *vstring***)**

Parameter	Description
exp	Phase number of the active operation within the unit recipe from which the descriptor is desired.
vstring	Name of the variable string (capable of storing at least 32 characters) used to return the specified descriptor.
bseqx	Name of the BSEQ function block declaration.

Example:

```
S_NUM = 0
REPEAT
    S_NUM = S_NUM + 1
    SUBR DESCRIPTOR (S_NUM, SUBRS(S_NUM))
UNTIL ((S_NUM >= LAST SUBR) OR (S_NUM > 31))
```

SUBR STATUS

function

PURPOSE: To retrieve the phase subroutine execution status from the current operation of the unit recipe executing in a BSEQ function block.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from a Batch 90 program}
var **= SUBR STATUS (***exp***)**

{from a UDF program}
var **= SUBR STATUS (***bseqx***,** *exp***)**

Parameter	Description
var	Name of the variable used to return the execution status which can be: -1 = complete. 0 = outside range of the current operation. 1 = active.
exp	Number of the phase subroutine from which the execution status is desired.
bseqx	Name of the BSEQ function block declaration.

Example:

```
S_NUM = 0
FOR I = 1 TO 32 DO
    S_NUM = I
    STS_VAR (S_NUM) = SUBR STATUS (BATCH_UNIT, S_NUM)
    IF (S_NUM >= LAST SUBR) BREAK
END FOR
```

TEXT SELECTOR

function block

PURPOSE: This function block displays messages on operator consoles.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The text selector function block (function code 151) must be declared before it can be accessed from a subroutine. The declaration can appear only in the data section. Batch 90 and UDF programming languages will only manipulate text selector function blocks if specifications S1 and S2 have values of 5. Specification S3 must have a value of zero. These are the function blocks default values.

BLOCK *fbname*, **BLK = ***const*, **FC = TEXT** [, **WATCH**]

Parameter	Description
fbname	Name of the function block.
const	Function block address.
WATCH (Batch 90 only)	Specifies that the text selector be placed on WATCH by the Batch Historian upon program start-up. If not specified, the text selector will be off watch.

Example: BLOCK K100_FMSG, BLK = 100, FC = TEXT

REFERENCE:

Get number *name***.MSG**

Parameter	Description
name	Name of the function block.
*name***.MSG**	Message number.

Example: X = OPMSG05.MSG

Get color *name***.CLR**

Parameter	Description
name	Name of the block.
*name***.CLR**	Message color.

Example: X = OPMSG05.CLR

TEXT SELECTOR (continued)

function block

Get blink status *name*.**BLNK**

Parameter	Description
name	Name of the function block.
name.**BLNK**	Blink status: false (0) = not blinking. true (1) = blinking.

Example: X = OPMSG.BLNK

COMMANDS:

Set number, blink status, and color **DISPLAY MESSAGE** *exp1* **COLOR** *exp2* [**BLINKING** *exp3*] \
USING *name*

Parameter	Description
exp1	A message number.
exp2	Console color code: black (0) white (1) red (2) green (3) blue (4) cyan (5) magenta (6) yellow (7) orange (8) yellow-green (9) green-cyan (10) cyan-blue (11) blue-magenta (12) magenta-red (13) dark gray (14) light gray (15) purple (17) brown (32)
exp3	Blinking status: on = message will blink. off = message will not blink.
name	Name of the text select function block.

Example: DISPLAY MESSAGE 2 COLOR GREEN BLINKING ON USING \
BIN_FILL_NO.

TEXT SELECTOR *(continued)*

function block

Set number **SET** *name***.MSG = ** *exp*

Parameter	Description
name	Name of the function block.
exp	Message number.

Example: SET OPMSG05.MSG = X

Set color **SET** *name***.CLR = ** *exp*

Parameter	Description
name	Name of the function block.
exp	Message color.

Example: SET OPMSG05.CLR = X

Set blink status **SET** *name***.BLNK = ** *exp*

Parameter	Description
name	Name of the function block.
exp	Blink status: off = 0 (not blinking). on = 1 (blinking).

Example: SET OPMSG05.BLNK = X

Set watch status (Batch 90 only) **SET** *name***.WATCH = ** *exp*

Parameter	Description
name	Name of the function block.
exp	Batch Historian watch status: 0 = off. 1 = on.

Example: SET OPMSG05.WATCH = ON

REMARKS:

These commands are buffered within the function block and do not have an immediate effect on the function block. The command will not take effect until the function block executes on its next cycle.

TIMER

data structure

PURPOSE: Timers are active data structures used to measure elapsed time. A timer may be declared either globally or locally.

LANGUAGE:

BATCH 90	UDF

DECLARATIONS: The following is used to define a timer.

TIMER *tname* **(***units***)**

Parameter	Description	
tname	Name of the timer.	
units	Unit of time:	
	SEC	Seconds.
	MIN	Minutes.
	HOUR	Hours.

Examples:

```
TIMER  TIMER1  (SEC)
TIMER  COOK_TIME  (MIN)
```

If a timer is declared locally (that is, in a phase or state subroutine) then its value is initialized to zero at the start of the subroutine.

REFERENCE: Timer data may be read by using the appropriate name in an expression.

Get value *tname***.VAL**

Parameter	Description
tname	Name of the timer.
*tname***.VAL**	Timer value.

Example: X = TIMER1.VAL

Get status *tname***.STS**

Parameter	Description
tname	Name of the timer.
*tname***.STS**	Status of timer: 0 = holding. 1 = running.

Example: IF (TIMER1.STS = RUNNING) THEN

data structure

Get alarm status *tname*.**ALM**

Parameter	Description
tname	Name of the timer.
tname.**ALM**	Alarm status: 0 = false (no alarm). 1 = true (alarm).

Example: IF (TIMER1.ALM) THEN

Get alarm limit *tname*.**LIM**

Parameter	Description
tname	Name of the timer.
tname.**LIM**	Alarm limit. This value is -1 if the START command does not specify and alarm status.

Example: IF (TIMER1.LIM > 60.0) THEN

COMMANDS: The following statements are used to control timers.

Start **START** *tname* [*exp*]

Parameter	Description
tname	Name of the timer.
exp	Alarm limit.

Examples:

```
START  TIMER1
START  TIMER1  30.0
```

This sets the alarm limit, updates the alarm status and sets the timer status to running. The timer value is not affected. If an alarm limit is not specified, the alarm status is not changed.

Hold **HOLD** *tname*

Parameter	Description
tname	Name of the timer.

Example: HOLD TIMER1

This sets the status to holding (nothing else is effected).

TIMER *(continued)*

data structure

Reset **RESET** *tname* **(***exp***)**

Parameter	Description
tname	Name of the timer.
exp	Initial time.

Example: RESET TIMER1 (0)

This sets the time to exp, updates the alarm status and sets the status to holding. If no limit is specified, it is assumed to be zero.

#TITLE

directive

PURPOSE: The #TITLE directive assigns a title to each page of the listing file. Typically this is done to make the listing easier to read.

LANGUAGE:

BATCH 90	UDF

FORMAT: **#TITLE "***string***"**

Parameter	Description
string	User defined title.

REMARKS: The #TITLE directive will cause an immediate page eject unless #TITLE is the first line of a program. The string is limited to a single line (250 characters).

UDF TYPE

function block

PURPOSE: The UDF Type 1 or UDF Type 2 function block executing the UDF program must be declared. This allows the compiler to check that all declarations to UDF Type 1 and UDF Type 2 function block inputs, outputs, and specifications are correct.

LANGUAGE:

BATCH 90	UDF

DECLARATIONS: The following is used to declare the UDF TYPE function block.

UDF TYPE *n*

Parameter	Description
n	1 for UDF Type 1 function blocks. 2 for UDF Type 2 function blocks.

Examples:

UDF TYPE 1
UDF TYPE 2

UNACQUIRE

*Refer to **COMMON SEQUENCE**.*

UNCONNECT

*Refer to **COMMON SEQUENCE**.*

UNIT DATA

statement

PURPOSE:

The is the first statement in the UNIT DATA section of a batch program. Unit data (like batch data) defines global data, function blocks, and symbols. Declarations made in unit data create generic references to specific declarations contained in unit definition files. Unit definition files are created, compiled, and downloaded separate from the batch program. For a given class of process equipment only one batch program source needs to be written.

The match between generic declarations in the UNIT DATA section of the batch program and the specific declarations with unit-specific values in the unit definition file is made by data type, name, and order. All components must match for the proper connection between declarations and references to be made.

LANGUAGE:

BATCH 90	UDF

FORMAT:

UNIT DATA
type1 name1
type2 name2
•
•
•
END DATA

Parameter	Description
type	Any one of the following: **ANY** Constant. **BLOCK** Function block. **CONST STRING** Constant string.
name	Name of the data.

Example:

```
UNIT DATA
    CONST STRING UNIT_NAME (16)
    ANY UNIT_CAPACITY = (100<500<1000)
    BLOCK INLET_VALVE, FC = DD
    BLOCK DRAIN_VALVE, FC = DD
    BLOCK TEMP_LOOP, FC = CS
    BLOCK UNIT_MSG, FC = DATAEXPT
    BLOCK FILTER_UNIT, FC = DD
END DATA
```

REMARKS:

All of the function block types supported by batch data are also supported in unit data. The only difference between the declarations is that the unit data declaration does not include the function block address information. The function block

UNIT DATA *(continued)*

statement

address information is supplied by the unit definition file. The batch language compiler generates a blank template from which the user can create the required number of unit definitions by simply filling in the blanks spaces. Refer to the following example.

Example:

```
#TITLE “UNIT DEFINITION FILE  EXAMPLE TITLE”
#DESCRIPTOR “16-CH FILE DESC”
#LOCATION LOOP = 1, PCU = 20, MODULE = 7, BSEQ = 1250
UNIT DATA
    CONST STRING UNIT_NAME = “PROD TANK T402”
    ANY UNIT_CAPACITY = (750)
    BLOCK INLET_VALVE, BLK = 2005, FC = DD
    BLOCK DRAIN VALVE, BLK = 2015, FC = DD
    BLOCK TEMP_LOOP, BLK = 2025, FC = CS
    BLOCK UNIT_MSG, BLK = 2045, FC = DATAEXPT
    BLOCK FILTER_UNIT, BLK = 2450, FC = CSEQ, MODULE = 9
END DATA
```

For function block declaration details, refer to:

AOL	Analog exception report
APID	Advanced PID
BBUF	Boolean data buffer
CS	Control station
CSEQ	Common sequence
DATAEXPT	User defined data export
DOL	Digital exception report
DD	Device driver
MSDD	Multi-state device driver
RBUF	Real data buffer
RCM	Remote control memory
REMSET	Remote manual set constant
RMC	Remote motor control
SMITH	Smith block
TEXT	Text

The block address data included on the #LOCATION line within the unit definition file is the one-time address specification for the BSEQ function block associated with the unit. This unit address data is used by other tools responsible for debugging, recipe creation, and file maintenance.

UNIT DESCRIPTOR

function

PURPOSE:

To retrieve the unit descriptor from the unit definition associated with the BSEQ function block. This command is only applicable when the batch program includes a UNIT DATA section.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

{from a Batch 90 program}
UNIT DESCRIPTOR (*vstring***)**

{from a UDF program}
UNIT DESCRIPTOR (*bseqx***,** *vstring***)**

Parameter	Description
vstring	Name of the variable string (capable of storing at least 32 characters) used to return the specified descriptor.
bseqx	Name of the BSEQ function block declaration.

UNIT PARAMETERS

declaration

PURPOSE:

Unit parameters are used to declare a list of global recipe parameters that are valid for the entire program. This list provides a means for passing data from a recipe to all phase subroutines within the program. This section contains declarations that define the name and data type of each parameter. The parameter values are supplied by the recipe and are valid for the entire recipe execution.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

UNIT PARAMETERS
  ANY *pname*
    - or -
  ANY *pname* **=** **(***min***<***val***<***max***)**
    - or -
  ANY *pname* **=** **(***const1, const2*[,..., *constn*]**)**
    - or -
  CSEQ *seqname***,** **LIST** **=** **(***filename***)**
    - or -
  STRING *str_arg* [**(***cons***)**]
    - or -
  STRING *str_arg* [**(***cons***)**] **=** **(***cstrname***)**
    - or -
  STRING *str_arg* [**(***cons***)**] **=** **(***cstr1, cstr2*[, ..., *cstrN*]**)**
    - or -
  STRING *str_arg* [**(***cons***)**] **=** **"***text_string***"**
    •
    •
    •
END DATA

Parameter	Description
pname	Name of the parameter used by the program.
min	Low limit value.
val	Parameter value.
max	High limit value.
constn	Nth constant value of the selection list.
seqname	Alias name that may be used by subroutines to reference or command a CSEQ function block.
filename	Name of the ASCII text file that contains CSEQ declarations written in the format as required by the CSEQ declarations in the BATCH DATA section.
str_arg	Name of the string recipe parameter.

UNIT PARAMETERS *(continued)*

declaration

Parameter	Description
cons	Maximum number of characters that the recipe may specify for the string parameter. The default value is 80 characters.
cstrname	Name of the constant string.
cstrN	Nth string constant value from the selection list.
text_string	Explicit constant string.

REFERENCES: Any subroutine within the program can refer to a parameter declared in the UNIT PARAMETERS section.

REMARKS: Only one UNIT PARAMETERS section is allowed within a program and it must appear before all phase subroutines.

UNRESERVE

Refer to ***COMMON SEQUENCE****.*

VAR

data

PURPOSE: To define a variable.

LANGUAGE:

BATCH 90	UDF

DECLARATION: The following is used to define a variable.

VAR *name* [**=** *cons*]

Parameter	Description
name	Name of the variable.
cons	Initial value.

Example: VAR VOLUME

REFERENCE: The value of a variable is read by using its name in an expression.

name

Parameter	Description
name	Name of the variable.

Example: X = VOLUME

COMMANDS: Refer to the ***ASSIGNMENT*** function.

VAR ARRAY

data structure

PURPOSE: A variable array is a group of variables organized in a table. An array may have 1, 2 or 3 dimensions. The array as a whole has a name. An individual variable (element) of an array is identified by an array index which specifies the position of the element within the array. An array index consists of subscripts - one subscript for each dimension.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

Variable arrays The following statements are used to declare variable arrays:

{for one dimensional array}
VAR ARRAY *name* **(***low***:***high1***)**

{for two dimensional array}
VAR ARRAY *name* **(***low***:***high1***,** [*low*]**:***high2***)**

{for three dimensional array}
VAR ARRAY *name* **(***low***:***high1***,** [*low*]**:***high2***,** [*low*]**:***high3***)**

Parameter	Description
name	Name of the variable array.
low	Lowest subscript value. Must be 0 or 1.
highN	Highest subscript value for the Nth dimension. Must be a constant. The total number of elements in an array must not exceed 16k.

Example:

```
VAR ARRAY XA (0:9), XB (0:9), XC (1:20)
VAR ARRAY Y (1:5, 1:10), YY (1:5, 1:10)
VAR ARRAY Z (0:5, 0:6, 0:7)
```

REFERENCE:

Element of an array An individual element of an array is referenced as follows:

{for one dimensional array}
name **(***exp1***)**

{for two dimensional array}
name **(***exp1, exp2***)**

{for three dimensional array}
name **(***exp1, exp2, exp3***)**

VAR ARRAY *(continued)*

data structure

Parameter	Description
name	Name of the variable array.
expN	Value of the Nth subscript. If a subscript value exceeds its specified range, it causes an error.

Examples:

```
A= XA (5)
A= Y (i, j)
A= Z (1, 2, i)
CALL MAX (Y (i, j), 105)
```

Entire array

In some operations it is necessary to refer to an entire array as a whole. An entire array is referenced as follows:

name

- or -

name **(****...***)**

Parameter	Description
name	Name of the variable array.
(**...***)**	Number of dimensions (that is, (*), (*,*), (*,*,*)).

The first form (just the name) is used when an array is passed as an argument to a function or monitor. The second form is used in other whole array operations (that is, array copy).

Examples:

```
CALL MAX_ELEMENT (Y)
XA (*) = 1
XB (*) = XA
Z (*,*,*) = 0
```

COMMANDS:

The following statements are used to operate on array elements and whole arrays.

Assign (write) a value to an array element

{for one dimensional array}
name **(***exp1***)** = *exp*

{for two dimensional array}
name **(***exp1***,** *exp2***)** = *exp*

{for three dimensional array}
name **(***exp1***,** *exp2***,** *exp3***)** = *exp*

VAR ARRAY *(continued)*

data structure

Parameter	Description
name	Name of the variable array.
expN	Value of the Nth subscript. If a subscript value exceeds its specified range, it causes an error.
exp	Value to be assigned to element.

Examples:

XA (5) = 10
Y (i, j) 10 + XA (i)
Z (0,j, k) = 1

Initialize an array (set all elements to the same value)

name **(***...***)** **=** *val*

Parameter	Description
name	Name of the variable array.
(*...***)**	Number of dimensions.
val	Value to be assigned to elements of an array. Must be a single constant or variable.

Examples:

XA (*) = 0
Y (*, *) = 10
Z (*, *, *) = 10

Copy one array to another

dname **(***...***)** **=** *sname* **(***...***)**

Parameter	Description
dname	Name of the destination variable array.
sname	Name of the source variable array.
(*...***)**	Number of dimensions.

Examples:

XA (*) = XB (*)
Y (*, *) = YY (*, *)

This statement assigns the value from each element of the source array to the corresponding element of the destination array. The source array is not affected. The source array and the destination array must be congruent (that is, the same

data structure

number of dimensions and same range for corresponding dimensions). If they are not congruent, an error occurs.

NOTE: Subscript range checking is done by the compiler if all of the subscripts of an array are constant. Range checking is always performed during program execution. Each time an array element is read, the subscript values are checked. If a subscript value is out-of-bounds, some error action is taken. If the error occurs during any mode other than FAULT mode (for example, NORMAL mode), the normal flow of control is aborted. The mode is changed to fault and control is transferred to fault logic. if the error occurs during FAULT mode, flow of control is not aborted.

VAR STRING

data

PURPOSE: Variable strings can be declared in the BATCH DATA section, UNIT PARAMETERS section, or any place in the phase, function, or monitor subroutines.

LANGUAGE:

BATCH 90	UDF

DECLARATION: **VAR STRING** *vstrname* [**(***cons***)**]

Parameter	Description
vstrname	Name of the variable string.
cons	Maximum number of characters that may be written to the variable string. Allowable values range from 1 to 80 (default value).

REFERENCES:

Get current string length *vstrname***.CLEN**

Parameter	Description
vstrname	Name of the variable string.
*vstrname***.CLEN**	Number of characters contained in the variable string.

Examples:

```
SOME_VAR = PROMPT_MSG.CLEN
OTHER_VAR = REQUIRED_ACTION.CLEN
```

Get maximum string length *vstrname***.MLEN**

Parameter	Description
vstrname	Name of the variable string.
*vstrname***.MLEN**	Number of characters that may be contained within the variable string.

Examples:

```
SOME_VAR = PROMPT_MSG.MLEN
OTHER_VAR = REQUIRED_ACTION.MLEN
```

REMARKS: The value defined by the .MLEN reference corresponds to the value specified by the variable string declaration. This value matches the value returned by the .CLEN reference only when the variable string is full.

VAR STRING *(continued)*

data

Get entire string value *vstrname*

Parameter	Description
vstrname	Name of the variable string.

Examples:

SOME_VAR_STRING = PROMPT_MSG
OTHER_VAR_STRING = REQUIRED_ACTION

Get value of a substring *vstrname* **(FROM** *exp1*[**FOR** *exp2*]**)**

Parameter	Description
vstrname	Name of the variable string that contains the referenced substring.
exp1	First character or starting point of the substring.
exp2	Number of characters to be included in the substring. If not specified, the substring is terminated at the end of *vstrname.*

Examples:

SOME_VSTRING = PROMPT_MSG (FROM 32)
OTHER_VSTRING = REQUIRED_ACTION (FROM 5 FOR 25)

REMARKS:

Character positions begin with the left-most character as character number one. Subsequent characters are numbered according to their position relative to character number one. References that extend beyond the end of the constant string shall be terminated at the end of the variable string.

COMMANDS:

When the source string or substring of a write string value operation exceeds the capacity of the destination variable string, only the amount of the source string that will fit shall be shifted to the destination. If the starting point of a substring write operation lies beyond the last character of the initial string value, the operation is treated as an append operation.

Write to an entire string variable

vstrname **=** *strname*[**(FROM** *exp1* **FOR** *exp2***)**]

- or -

vstrname **= "***string_exp***"**

- or -

vstrname **=** *strblk.ref_ext*

VAR STRING (continued)

data

Parameter	Description
vstrname	Name of the variable string.
strname	Name of any type of string, string constant, or string variable.
exp1	Starting point within *strname*.
exp2	Number of characters from *exp1* to be included in the substring. If no substring references are made, the entire contents of *strname* shall be written to *vstrname*.
string_exp	Explicit constant string to be written to *vstrname*.
strblk	Name of a DATAEXPT function block.
ref_ext	Input or output string reference extension (.ISTR or .OSTR)

Examples:

REQUIRED_ACTION = “VERIFY PROCESS CONDITION”
PROMPT_MSG = REQUIRED_ACTION (FROM 1 FOR 8)
SOME_VSTRING = STATUS_IND.ISTR

Write to substring of string variable

vstrname **(FROM** *exp1***)** **=** *strname*[**(FROM** *exp2* **FOR** *exp3***)**]

- or -

vstrname **(FROM** *exp1***)** **=** **“***string_exp***”**

- or -

vstrname **(FROM** *exp1***)** **=** *strblk.ref_ext*

Parameter	Description
vstrname	Name of the variable string.
exp1	First character or starting point of the substring, within *vstrname*, that is to be written to.
strname	Name of any string constant or string variable.
exp2	Starting point within *strname*.
exp3	Number of characters from *exp1* to be included in the substring. If not substring references are made, the entire contents of *strname* shall be written to *vstrname*.
string_exp	Explicit constant string to be inserted into *vstrname* beginning at the *exp1* character.
strblk	Name of an export string function block.
ref_ext	Input or output string reference extension (.ISTR or .OSTR).

data

Examples:

REQUIRED_ACTION (FROM 32) = “CLEAN VESSELS”
PROMPT_MSG (FROM 24) = UNIT_NAME (FROM 1 FOR 16)
SOME_VSTRING (FROM 17) = DIALOGUE.OSTR

REMARKS:

Character positions begin with the left-most character as character number one. Subsequent characters are numbered according to their position relative to character number one.

Fill an entire string with a character

vstrname **(*) = “***a***”**

Parameter	Description
vstrname	Name of the variable string.
(*)	Denotes that the entire variable string is to be filled with the character specified.
a	The single character that is to used to file the entire variable string.

Examples:

PROMPT_MSG (*) = “ “
REQUIRED_ACTION (*) = “?”

Append a string to the end of a variable string

vstrname **(END) =** *strname* [**(FROM** *exp1* **FOR** *exp2***)**]

Parameter	Description
vstrname	Name of the variable string.
(END)	Denotes that the string write operation will add the string to the end of the current string contained in *vstrname*.
strname	Name of any type of string, string constant (symbolic or explicit), string variable, or string function block.
exp1	Starting point within *strname*.
exp2	Number of characters from *exp1* to be included in the substring. If not substring references are made, the entire contents of *strname* shall be written to *vstrname.*

Examples:

PROMPT_MSG (END) = “SAMPLE REQUIRED!”
REQUIRED_ACTION (END) = UNIT_NAME (FROM 17 FOR 32)

VAR STRING ARRAY

data structure

PURPOSE:

A variable string array is a group of variable strings organized in a table. An array may have 1, 2 or 3 dimensions. The array as a whole has a name. An individual variable string (element) of an array is identified by an array index which specifies the position of the element within the array. An array index consists of subscripts - one subscript for each dimension.

LANGUAGE:

BATCH 90	UDF

DECLARATION:

Variable arrays

The following statements are used to declare variable string arrays:

{for one dimensional array}
VAR STRING ARRAY *name* **(**[*length,*]*low***:***high1***)** [*list*]

{for two dimensional array}
VAR STRING ARRAY *name* **(**[*length,*]*low***:***high1***,** [*low***:**]*high2***) **
[*list*]

{for three dimensional array}
VAR STRING ARRAY *name* **(**[*length,*]*low***:***high1***,** [*low***:**]*high2***,** \\
[*low***:**]*high3***)** [*list*]

Parameter	Description
name	Name of the array.
length	Maximum string length for each array element. Allowable values range from 1 to 80 (default value).
low	Lowest subscript value. Must be 0 or 1.
highN	Highest subscript value for the Nth dimension. Must be a constant. The total number of elements in an array must not exceed 16k.
list	List of array element names. This one dimensional list has one entry for each element of the array. The order of the entries is determined by varying the first (leftmost) subscript the fastest, then the second and the third.

NOTE: For two and three dimensional arrays, the lowest subscript value for the second and third dimension must match the value that was specified by the first dimension.

Examples:

VAR STRING ARRAY XA (20, 1:3) (“STRING1”, “STRING2”, \\
“STRING3”)

VAR STRING ARRAY Y (1:3, 1:2) (“STRING11”, “STRING21”, \
“STRING31”, “STRING12”, “STRING22”, “STRING32”)

VAR STRING ARRAY Z (60, 1:2, 1:2, 1:2) (“STRING111”, \
“STRING211”, “STRING121”, “STRING221”, “STRING112”, \
“STRING212”, “STRING122”, “STRING222”)

REFERENCE:

Element of an array

An individual element of an array is referenced as follows:

{for one dimensional array}
name **(***exp1***)**

{for two dimensional array}
name **(***exp1, exp2***)**

{for three dimensional array}
name **(***exp1, exp2, exp3***)**

Parameter	Description
name	Name of the array.
expN	Value of the Nth subscript. If a subscript value exceeds its specified range, it causes an array error.

Examples:

STRING = XA (5)
STRING = Y (I, J)
STRING = Z (1, 2, K)
SUBSTRING = B (I, 3) (FROM 1 FOR 8)

Entire array

In some operations it is necessary to refer to an entire array as a whole. An entire array is referenced as follows:

name

- or -

name **(****...***)**

Parameter	Description
name	Name of the array.
(**...***)**	Number of dimensions (that is, (*), (*,*), (*,*,*)).

The first form (just the name) is used when an array is passed as an argument to a function or monitor. The second form is used in other whole array operations (that is, array copy).

VAR STRING ARRAY *(continued)*

data structure

Examples:

NEW ARRAY (*) = OLD_ARRAY(*)
INIT_ARRAY (*,*) = 1

COMMANDS:

The following statements are used to operate on array elements and whole arrays.

Assign (write) a value to an array element

{for one dimensional array}
name **(***exp1***)** = *exp*

{for two dimensional array}
name **(***exp1***,** *exp2***)** = *exp*

{for three dimensional array}
name **(***exp1***,** *exp2***,** *exp3***)** = *exp*

Parameter	Description
name	Name of the array.
expN	Value of the Nth subscript. If a subscript value exceeds its specified range, it causes an array error.
exp	Value to be assigned to element.

Examples:

STRING1 (5) = “ELEMENT 3”
STRING2 = “ELEMENT_IJ”
STRING3 (1, 3, K) = “ELEMENT13K”

Initialize an array (set all elements to the same value)

name **(****...***)** **= STRING**

Parameter	Description
name	Name of the array.
(**...***)**	Number of dimensions.
STRING	Value to be assigned to elements of an array. Must be a single string constant or variable whose length is less than or equal to the specified maximum string length.

Examples:

STRING1 (*) = “INIT ON”
STRING2 (*, *) = “EMPTY”
STRING3 (*, *, *) = “NULL STRING”

VAR STRING ARRAY *(continued)*

data structure

Copy one array to another

dname **(***...***)** **=** *sname* **(***...***)**

Parameter	Description
dname	Name of the destination array.
sname	Name of the source array.
(*...***)**	Number of dimensions.

Examples:

```
NEW_STRING (*) = OLD_STRING (*)
STRING2 (*, *) = DATA_ARRAY (*, *)
```

This statement assigns the value from each element of the source array to the corresponding element of the destination array. The source array is not affected. The source array and the destination array must be congruent (that is, the same number of dimensions and same range for corresponding dimensions). If they are not congruent, an array error occurs.

NOTE: Subscript range checking is done by the compiler if all of the subscripts of an array are constant. Range checking is always performed during program execution. Each time an array element is read, the subscript values are checked. If a subscript value is out-of-bounds, some error action is taken. If the error occurs during any mode other than FAULT mode (for example, NORMAL mode), the normal flow of control is aborted. The mode is changed to fault and control is transferred to fault logic. If the error occurs during FAULT mode, flow of control is not aborted.

WAIT FOR

statement

PURPOSE: This statement delays sequential execution for the specified period of time.

LANGUAGE:

BATCH 90	UDF

FORMAT: **WAIT FOR** *exp units*

Parameter	Description
exp	Time interval.
units	Units of time: **SEC** Seconds. **MIN** Minutes. **HOUR** Hours.

Examples:

WAIT FOR 4.5 MIN {wait for 4.5 minutes}
WAIT FOR X SEC {wait for X seconds}

REMARKS: The program is suspended for one cycle each time the timer is checked. This statement can not be used where execution is continuous (for example, in a monitor subroutine).

WAIT UNTIL

statement

PURPOSE: This statement causes a phase (Batch 90) or state (UDF) subroutine to wait until a specified condition is met.

LANGUAGE:

BATCH 90	UDF

FORMAT: **WAIT UNTIL (***exp***)**

Examples:

```
WAIT UNTIL (LT102 >= 45.0) OR (TIMER1.VAL >= 30))
WAIT UNTIL (TRUE)        {one cycle delay}
```

REMARKS: If exp is false, exp is evaluated until it becomes true. If exp is true, execution resumes at the following statement.

The program is suspended for one cycle each time exp is evaluated (including the first time). The suspension occurs before evaluating exp. This statement can not be used where execution is continuous (for example, in a monitor subroutine).

WAIT WHILE

statement

PURPOSE: This statement causes a phase (Batch 90) or state (UDF) subroutine to wait as long as a specified condition is true.

LANGUAGE:

BATCH 90	UDF

FORMAT: **WAIT WHILE (***exp***)**

Examples:

```
WAIT WHILE (LT102 >= 45.0) OR (TIMER1.VAL >= 30))
WAIT WHILE (FALSE)          {one cycle delay}
```

REMARKS: If exp is true, exp is evaluated until it becomes false. If exp is false, execution resumes at the following statement.

The program is suspended for one cycle each time exp is evaluated (including the first time). The suspension occurs before evaluating exp. This statement can not be used where execution is continuous (for example, in a monitor subroutine).

WHILE...ENDWHILE

control structure

PURPOSE: This statement continuously executes a block of statement while some condition is true.

LANGUAGE:

BATCH 90	UDF

FORMAT:

WHILE (*exp***)**
statement1
•
•
•
statementN
ENDWHILE

Example:

```
WHILE (FC101.PV < 50)
    SET FC101.CO = FC101.CO + 1
ENDWHILE
```

REMARKS: If exp is false, execution transfers to statement following ENDWHILE. A while loop can be terminated by a BREAK statement.

The program is suspended for one cycle each time exp is evaluated (including the first time). The suspension occurs before evaluating exp. This statement can not be used where execution is continuous (for example, in a monitor subroutine).

APPENDIX A - SAMPLE APPLICATIONS

BATCH 90

There are many applications for the Batch 90 programming language. This section will detail a few applications in order to point out some important concepts. Some sample applications are:

- Parallel phase subroutines.
- Common sequences.

Parallel Phase Subroutines

In many batch applications it is advantageous to perform more than one activity simultaneously. Usually, it is not practical to embed logic for parallel activities within programs. This is particularly true when the parallel activities depend upon which unit procedure is being run. Therefore, it is necessary for the unit procedure to select phase subroutines for parallel execution within an operation. Batch 90 allows unit procedures to specify parallel phase subroutines. The major advantages of this capability are:

1. Each phase subroutine is self contained and does not need to include complex logic for monitoring parallel activities.

2. Phase subroutine logic is simplified. Unit procedures more readily depict the activities to be performed and the order in which they are to be executed (sequential or parallel).

Figure A-1 shows a typical example of a unit procedure that contains parallel phase subroutines. The manner in which parallel subroutines are assigned operation numbers is as follows:

xx.yy

where:

xx	Operation number containing parallel phase subroutines.
yy	Phase number of parallel phase subroutine within the operation (specify zero if there are no parallel phase subroutines).

In Figure A-1, operation two has two parallel phase subroutines. Operations three and four have three parallel phase subroutines. Operations zero, one, five, and six have one phase subroutine.

RECIPE ID: 1
RECIPE DESCRIPTOR: PARALLEL_TEST

OPERATION	PHASE SUBR	LO		VALUES		HI
0	EMERGENCY_STOP					
1	RINSE_HEADER					
	RINSE_TYPE			FULL		
2.1	CHARGE_REACTOR					
	INGREDIENT			D_WATER		
	QUANTITY	0.0	<	100.0	<	500.0
	MAX_TIME		<	5.0	<	
	TIME_UNITS			MINUTES		
2.2	HEAT_REACTOR					
	END_TARGET_TEMP	20.0	<	25.0	<	30.0
	TARGET_RATE_CHANGE		<	5.0	<	
3.1	CHARGE_REACTOR					
	INGREDIENT			INGRED_A		
	QUANTITY	10.0	<	25.5	<	125.0
	MAX_TIME		<	2.5	<	
	TIME_UNITS			MINUTES		
3.2	CHARGE_REACTOR					
	INGREDIENT			INGRED_B		
	QUANTITY	20.0	<	75.5	<	250.0
	MAX_TIME		<	250.0	<	
	TIME_UNITS			SECONDS		
3.3	RUN_AGITATOR					
	TARGET_SPEED	10.0	<	50.0	<	60.0
	MAX_RATE_CHANGE		<	2.5	<	
4.1	HEAT_REACTOR					
	END_TARGET_TEMP	20.0	<	95.0	<	150.0
	TARGET_RATE_CHANGE		<	1.5	<	
4.2	DOSE_REACTOR					
	INGREDIENT			CATALYST_1		
	QUANTITY	0.0	<	10.0	<	50.0
4.3	CHARGE_REACTOR					
	INGREDIENT			INGRED_C		
	QUANTITY	5.0	<	100.0	<	500.0
	MAX_TIME		<	1.0	<	
	TIME_UNITS			HOURS		
5	PRODUCT_SAMPLE					
	QUALITY_PARAM1_TOLERANCE			ONE_PERCENT		
	QUALITY_PARAM2_TOLERANCE			TEN_PERCENT		
	QUALITY_PARAM3_TOLERANCE			NOT_APPLICABLE		
	SIGN_OFF_REQD			YES		
6	XFER_PRODUCT					

Figure A-1. Example Parallel Phase Subroutine Unit Procedure

BATCH 90 PROGRAM IMPLEMENTATION REQUIREMENTS

The structure of a parallel phase subroutine is no different from other phase subroutines. The phase subroutine may include unit procedure parameters, local data, continuous logic, normal logic, fault logic, etc. The only additional requirement is that the compiler directive known as #MAXPARALLEL must be included within the Batch 90 program. This compiler directive determines the maximum number of parallel phase subroutines allowed by the unit procedure editor.

#MAXPARALLEL = *n*

where:

n Value between two and 32.

The compiler uses the #MAXPARALLEL directive to calculate the required data space for the program (specification S12 of the BSEQ function block).

S12 = [*largest_phase_subroutine* * #MAXPARALLEL] + *batch data*

The built in status variables OPERATION and PHASE NUMBER enable the programmer to include logic that determines the phase subroutine's location within the unit procedure (that is, if the phase subroutine is part of an operation that includes parallel phase subroutines). Within the example unit procedure (refer to Figure A-1) the parallel phase subroutine CHARGE_REACTOR is assigned to four locations (refer to Table A-1).

Table A-1. Parallel Phase Subroutine CHARGE_REACTOR Locations

Operation Number	Phase Subroutine	Operation	Phase
2.1	CHARGE_REACTOR	2	1
3.1	CHARGE_REACTOR	3	1
3.2	CHARGE_REACTOR	3	2
4.3	CHARGE_REACTOR	4	3

LOGIC EXECUTION CONTROL FLOW

It is important to understand the manner in which parallel phase subroutines are executed. Within the boundaries of an operation that contains parallel phase subroutines, the phase subroutines are executed sequentially by phase number. Figure A-2 shows the program logic flow for operation three of the example unit procedure.

Data structures include items such as timers, integrators, ramps, and function generators. All phase subroutines within an operation will be in the same sequential logic mode (that is, all executing normal logic or all executing fault logic, etc.). Thus, when any phase subroutine executes a fault command, all phase subroutines transfer from normal to fault logic. Similarly, if any phase subroutine executes a restart command, all phase subroutines will transfer to restart logic. If the operator puts the sequence into hold, all phase subroutines go to hold logic.

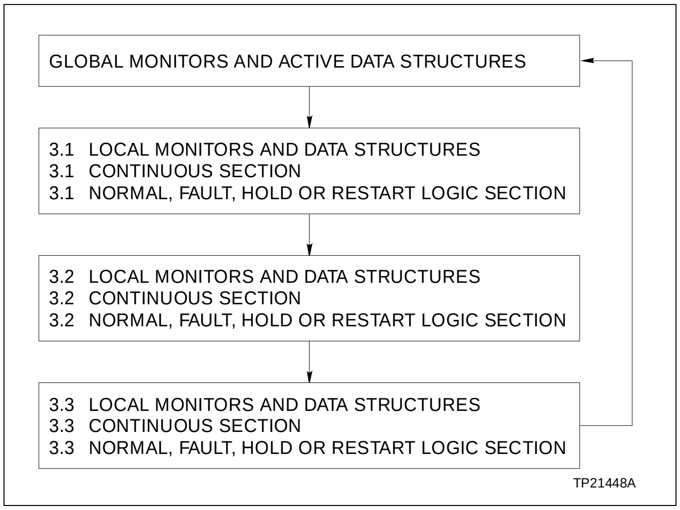


Figure A-2. Example Phase 3 Program Logic Flow Diagram

An operation with parallel phase subroutines is complete only if all phase subroutines have completed (finished normal logic) or one of the phase subroutine executes a NEXT OPERATION statement. The NEXT OPERATION statement will terminate all parallel phase subroutines and change the operation number to the value specified by the command.

If the parallel phase subroutines are executing restart logic, program execution is not returned to normal logic until all phase subroutines have completed restart logic.

Common Sequences

In many applications, certain batch units may serve as common resources to other batch units. For example, a series of reactors might share common feed tanks, headers, or cooling systems. Each of the shared resources have batch sequences associated with them. Because control of these sequences is required by more than one batch unit, they are referred to as common sequences (Common Sequence Function Block, function code 219).

COMMON SEQUENCE FUNCTION BLOCK DIAGRAM

Figure A-3 shows an example Common Sequence Function Block (CSEQ) configuration. The Batch Historian Function Block (BHIST) is optional, and is not required for common sequence operation.

The Batch Sequence Function Block (BSEQ) executes the batch program. The CSEQ function block provides an interface by which the batch sequence is shared or controlled by other batch programs.

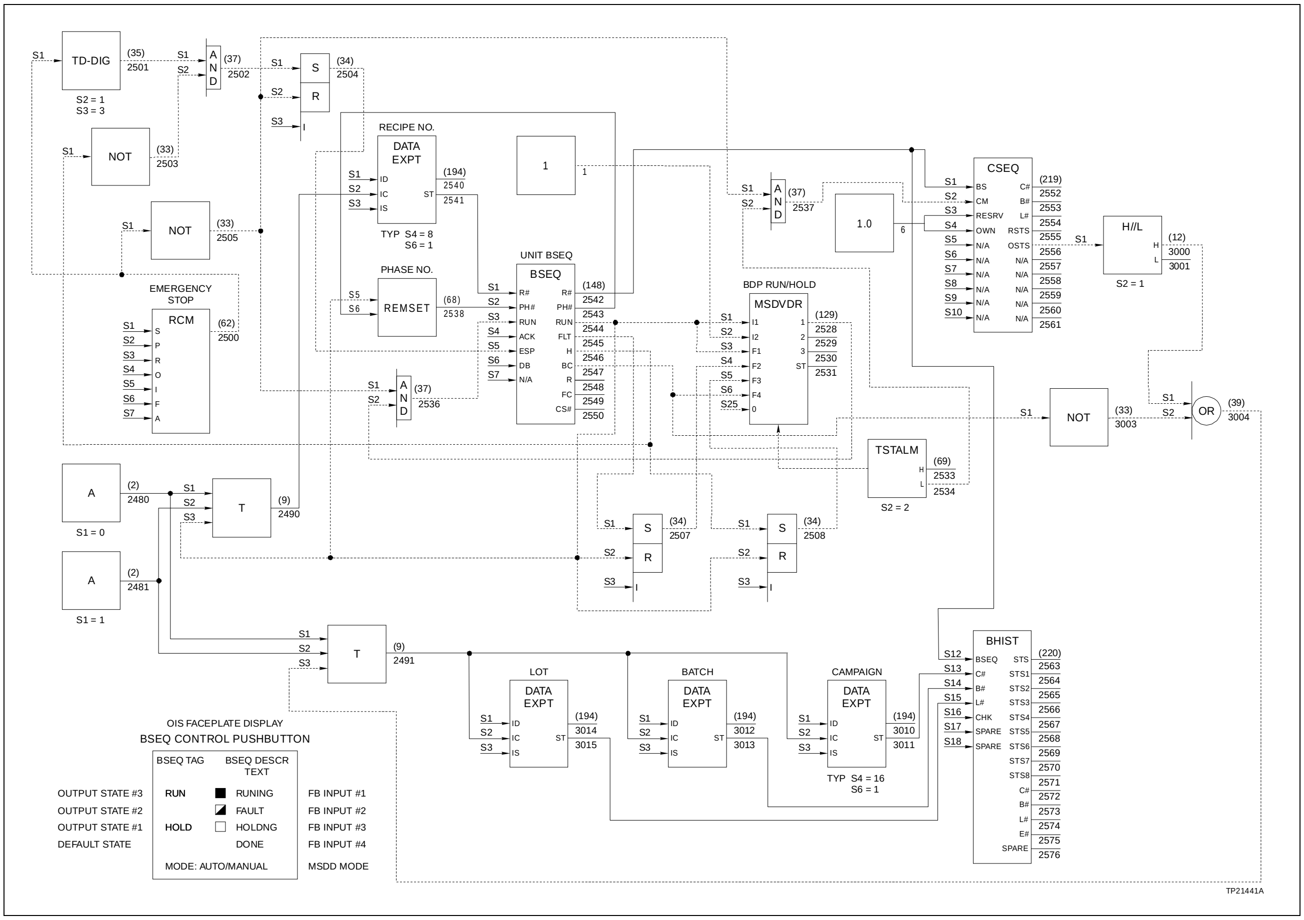


Figure A-3. Example Common Sequence Function Block Configuration

Common sequence affects control of some inputs to the BSEQ and BHIST function blocks. The control mode of the CSEQ function block determines what effect it has on the BSEQ and BHIST function block inputs.

Operator Mode

In the operator mode, BSEQ and BHIST function block inputs are enabled and operate according to their normal function.

Remote Mode

In the remote mode, BSEQ function block inputs (that is, run/hold, unit procedure, and operation) are disabled. The BSEQ function block responds to remote commands received through the CSEQ function block or these input functions. The BHIST inputs (that is, campaign, batch and lot number) are disabled. The BHIST function block receives the input parameters from the CSEQ function block.

COMMON SEQUENCE COMMUNICATION

The CSEQ function block serves as an interface between the common sequence and other batch or common sequences. The CSEQ function block receives commands from other programs and sends status information to these programs.

Before a program can communicate with a CSEQ function block, it must establish a connection with the CSEQ function block. The CSEQ block can only support a limited number of program connections. The limit of program connections is configurable, not tunable and set through specification S13 of the CSEQ function block.

Once a client program connects to a server CSEQ function block, the client program will receive status information from the server program. The server program sends status information via its CSEQ function block to all connected client programs on the following basis:

1. Upon initial connection, the CSEQ function block sends the client program all of the following data:

- Connection status.
- Reservation status.
- Ownership status.
- Control mode (operator or remote).
- Program status (inactive, running or holding).
- Unit procedure number.
- Operation number.
- Fault code.
- Campaign, batch and lot numbers.
- Eight status variables (set by the server program).

2. After the server CSEQ function block sends the initial status information, it sends only changing status information to the connected client programs. A CSEQ function block message will not be sent more often then the time period determined by specification S11 (update minimum time) of the CSEQ function block.

3. If status information does not change, the CSEQ function block periodically sends a message to confirm that the communication to client programs still exists. Specification S12 of the CSEQ function block determines how often this message is sent.

ACQUIRING CONTROL OF A CSEQ FUNCTION BLOCK

Establishing a connection with a CSEQ function block allows the client program to communicate with the server program. In some cases, this is all that is required by the application. For example, the client program connects to a server program to determine the status of the batch unit being controlled by the common sequence.

In many applications the client programs will need to gain control of the common sequence and manipulate it. For example, the reactor client program may need to send material transfer requests (quantities, flow rates, etc.) to a common feed tank (the server). In another example, the client may need to initiate the execution of a particular unit procedure by the server. In this instance, the client program must acquire ownership of the server.

The CSEQ function block may only have one owner at a time. Therefore, the CSEQ function block has an ownership queue. ACQUIRE commands issued by client programs have a priority associated with each request (refer to Section 4 for information on all CSEQ commands). The ownership queue stores ACQUIRE commands received from clients by priority. Queued ACQUIRE commands that meet the reservation criteria, are granted in their order of priority.

Client programs may reserve a CSEQ function block for later use. The reserve command enables client programs to specify which values for campaign, batch and lot numbers must match for an ACQUIRE command to be qualified. The CSEQ function block may only have one active reservation at a time.

Reservation, ownership and ownership queuing combine to provide the means for the first level of batch unit management. The CSEQ application is a focal point for applications that involve production scheduling or variable production paths.

TYPICAL APPLICATION

The process shown in Figure A-4 represents the simplest use of the CSEQ function block. The process has a main reactor that acquires control of a feed tank.

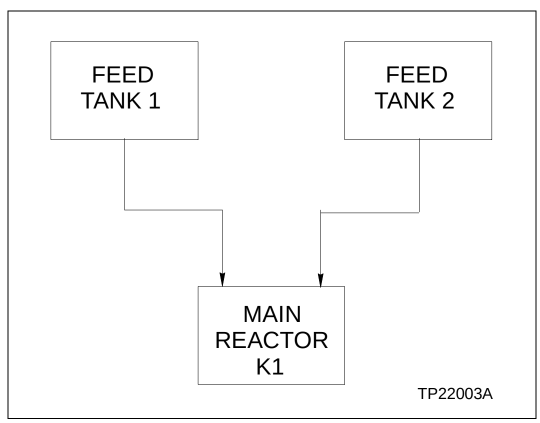


Figure A-4. Example Usage of Common Sequence

In order for the reactor to control the batch programs associated with the feed tanks, the CSEQ function block for the feed tanks must be declared within the batch data section of the reactor program (refer to Section 4 for CSEQ declaration specifics). Figure A-5 shows typical logic used within a phase subroutine of the reactor program.

```
BATCH DATA
    BLOCK FEEDTANK, BLK = 700, FC = CSEQ, /*BLOCK 700 IS THE ADDRESS OF THE BSEQ \
    FOR THE FEEDTANK*/
END DATA
    •
    •
    •
PHASE SUBR FEED_FROM_TNK1
    ANY QUANTITY
CONTINUOUS
    IF (FEEDTANK.CSTS = CONNECTED) THEN
        IF (FEEDTANK.XSTS = IN_FAULT) FAULT 1
    ENDIF
    IF (FEEDTANK.CSTS = FAILED) FAULT 2
NORMAL LOGIC
    CONNECT FEEDTANK
    WAIT UNTIL (FEEDTANK.CSTS = CONNECTED)
    ACQUIRE FEEDTANK CAMPAIGN = MINE, BATCH = MINE, LOT = MINE
    WAIT UNTIL (FEEDTANK.OSTS = OWNED_BY_ME)
    HOLD FEEDTANK
    WAIT UNTIL ((FEEDTANK.XSTS = HOLDING) OR (FEEDTANK.XSTS = COMPLETED))
    START FEEDTANK RECIPE = 21, OPERATION = 1
    WAIT UNTIL (FEEDTANK.XSTS = RUNNING)
    SET FEED_TANK,CV1 = EXP1
    SET FEED_TANK.CV2 = EXP2
    WAIT WHILE (FEEDTANK.CPEND >= WAITING
        •
        •
        •
    WAIT UNTIL (FEEDTANK.XSTS = COMPLETED
    UNCONNECT FEEDTANK
ENDSUBR
```

Figure A-5. Example Phase Subroutine Using CSEQ Commands

In Figure A-5, the logic issues a CSEQ command and then waits for confirmation that the desired action has occurred. This ensures proper operation of the CSEQ function block. There is a time delay between the issuing of CSEQ commands and the response from the CSEQ function block. The typical pattern of events used in common sequence applications is:

1. **CONNECT AND CONFIRM** ***cseq*****.CSTS**

2. **ACQUIRE AND CONFIRM** ***cseq*****.OSTS**

3. Manipulate required CSEQ attribute and confirm:

   START AND CONFIRM *cseq***.XSTS**

   - or -

   HOLD AND CONFIRM *cseq***.XSTS**

   - or -

   SET *cseq***.CV(***x***) AND CONFIRM** *cseq***.SV(***y***)**

4. **UNACQUIRE AND CONFIRM** *cseq***.OSTS**

5. **UNCONNECT AND CONFIRM** *cseq***.CSTS**

The continuous section checks the validity of the CSEQ connection and the actual fault status of the common sequence. This type of logic implemented in a continuous logic section is typical for most common sequence applications.

USER DEFINED FUNCTION (UDF)

The following example program demonstrates how to use the UDF programming language to control the operation of a parallel grouping of filters. Filter condition is monitored by measuring the differential pressure across the group of filters. Figure A-6 shows a graphical representation of the process and associated equipment.

When a run input signal is sent to the program, the start-up sequence is executed. During the start-up sequence, filter one is backwashed for tunable amount of time (BWASH_DURATION) and put into operation. After a tunable amount of time (BWASH_INTERVAL) filter two is put through the same procedure. Filter three is put through the procedure last. At this point all filters are functioning properly and the program is monitoring differential pressure across the filters. As the filters start to fill with material, the differential pressure rises. When the differential pressure exceeds the tunable set point (TRIGGER_PRESSURE), the program executes the cleaning procedure on filter one. The cleaning procedure consists of opening the inlet valve to the drain piping, closing the outlet valve, and letting the process fluid flow into the filter and out the drain pipes taking filtered material with it. After a tunable amount of time (BWASH_DURATION), the outlet valve is opened and the inlet valve is set to allow for normal flow through the filter. The filter is put into operation, and a timer value (BWASH_INTERVAL) is set. The program continues to monitor differential pressure. If the differential pressure is still higher than the set point when the timer value reaches zero, the next filter is cleaned. This sequence continues until the differential pressure drops below the set point. When the

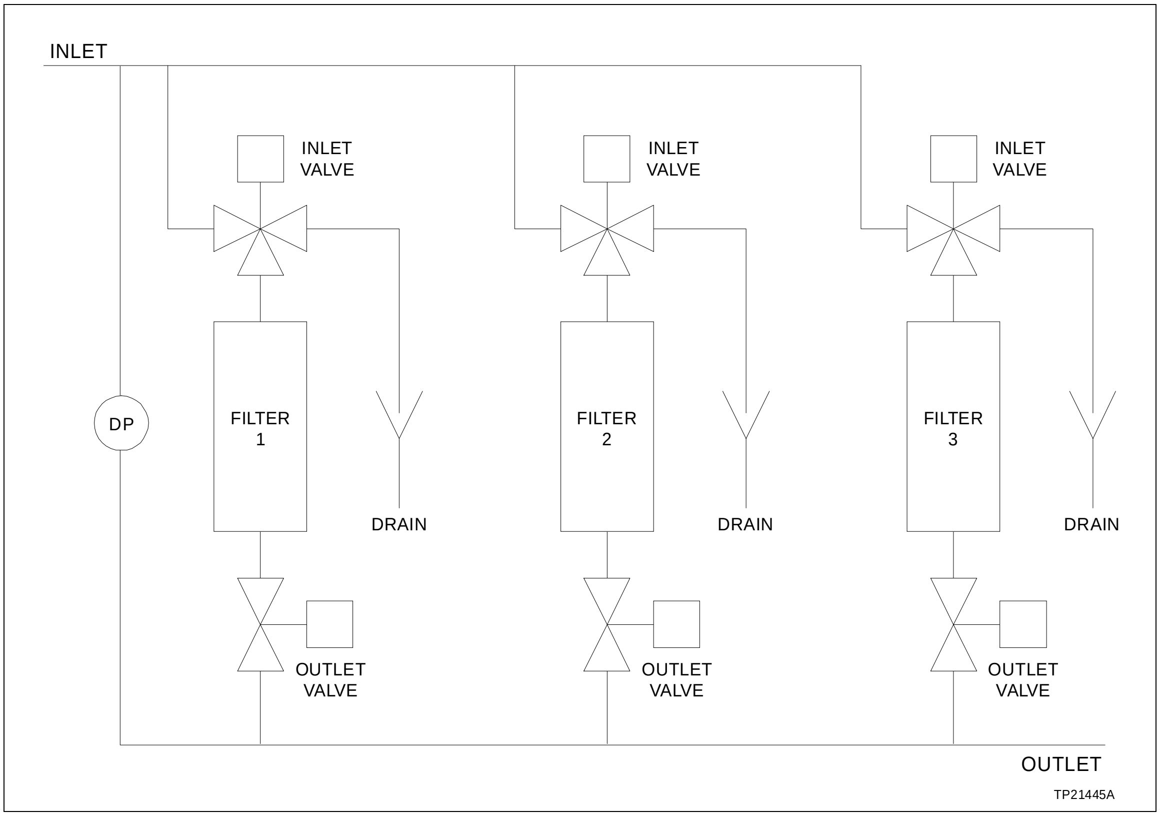


Figure A-6. Example UDF Process

differential pressure once again exceeds the set point, the next filter is cleaned and the sequence continues until the differential pressure drops below the set point. When a stop input signal is sent to the program, the shut down procedure is executed. The shut down procedure consists of closing the inlet valve and then the output valve of filter one and then filter two and finally filter three.

Figure A-7 shows the UDF program used to control the filters. Figure A-8 shows the function block configuration for the program.

```
#DEBUG LEVEL = 4
#DESCRIPTOR "UDF EXAMPLE"
#TITLE "UDF CONTROL PROGRAM FOR MULTI-SECTION FILTER"

DATA
    UDF TYPE 2
    {UDF BLOCK INPUT DECLARATIONS}
        INPUT 1 FLTR1_INLET, FC = MSDD
        INPUT 2 FLTR1_OUTLET, FC = MSDD
        INPUT 3 FLTR2_INLET, FC = MSDD
        INPUT 4 FLTR2_OUTLET, FC = MSDD
        INPUT 5 FLTR3_INLET, FC = MSDD
        INPUT 6 FLTR3_OUTLET, FC = MSDD
        INPUT 16 OPERATOR_MSG, FC = DATAEXPT
        INPUT 17 DIFF_PRESSURE, FC = AOL
        INPUT 18 RUN_STOP_PB, FC = RCM
    {UDF BLOCK OUTPUT DECLARATIONS}
        OUTPUT 0 FLTR1_BWASH_IND
        OUTPUT 1 FLTR2_BWASH_IND
        OUTPUT 2 FLTR3_BWASH_IND
        OUTPUT 4 FLTR1_SERVICE_IND
        OUTPUT 5 FLTR2_SERVICE_IND
        OUTPUT 6 FLTR3_SERVICE_IND
        OUTPUT 8 BWASH_TIME_REMAINING
        OUTPUT 9 TIME_TO_NEXT_BWASH
    {UDF BLOCK TUNABLE PARAMTER SPECIFICATIONS}
        SPEC 19 BWASH_DURATION
        SPEC 20 BWASH_INTERVAL
        SPEC 21 SYSTEM_STARTUP_DURATION
        SPEC 22 TRIGGER_PRESSURE
    {UDF PROGRAM GLOBAL DATA DECLARATIONS}
        VAR SYSTEM_STATUS, FLTR1_STS, FLTR2_STS, FLTR3_STS
        CONST TO_FLTR = 3, TO_DRAIN = 2, NUMBER_OF_FLTRS = 3
        CONST FILTERING = 3, BACKWASH = 2, SHUTDOWN = 1
        TIMER BWASH_TIMER (MIN), FLTR_INTERVAL (MIN)
END DATA

MONITOR DEV_STATUS_MON
    CONTINUOUS
        IF ((FLTR1_INLET.STS = BAD) OR (FLTR1_OUTLET.STS = BAD)) THEN
            FLTR1_STS = BAD
        ELSE
            FLTR1_STS = GOOD
        ENDIF

        IF ((FLTR2_INLET.STS = BAD) OR (FLTR2_OUTLET.STS = BAD)) THEN
            FLTR2_STS = BAD
        ELSE
            FLTR2_STS = GOOD
        ENDIF

        IF ((FLTR3_INLET.STS = BAD) OR (FLTR3_OUTLET.STS = BAD)) THEN
            FLTR3_STS = BAD
        ELSE
            FLTR3_STS = GOOD
        ENDIF

        IF ((FLTR1_STS = BAD) OR (FLTR2_STS = BAD) OR (FLTR3_STS = BAD)) THEN
            SYSTEM_STATUS = BAD
        ELSE
            SYSTEM_STATUS = GOOD
```

Figure A-7. Example UDF Program (Page 1 of 4)

```
            ENDIF
    ENDSUBR  DEV_STATUS_MON

    DATA
        MONITOR  DEV_STATUS_MON  RUNNING
    END  DATA

    FUNCTION  MOVE_VALVE
            MSDD  VALVE
            ANY  COMMAND_POSITION
        EXECUTABLE
            IF  (VALVE.MODE  <>  AUTO)  SET  VALVE.MODE  =  AUTO
            SET  VALVE.CO  =  COMMAND_POSITION
            WAIT  WHILE  (VALVE.STS  =  WAITING)
            IF  (VALVE.STS  =  BAD)  THEN
                SET  VALVE.MODE  =  MANUAL
                SET  OPERATOR_MSG.OSTR  =  "VALVE  FAILURE  IN  FILTER  SYSTEM"
                FAULT 1
            ELSE
                RETURN
            ENDIF
    ENDSUBR  MOVE_VALVE

    FUNCTION  FLTR_ACTION
            MSDD  INLET
            MSDD  OUTLET
            OUTPUT  BWASH_IND
            OUTPUT  SERVICE_IND
            ANY  FILTER_OP
            ANY  DURATION
        EXECUTABLE
            RESET  BWASH_TIMER  0
            START  BWASH_TIMER  (DURATION)
            DO  CASE  FILTER_OP
                CASE  SHUTDOWN
                    SET  SERVICE_IND.VAL  =  OFF
                    CALL  MOVE_VALVE  (INLET,  CLOSED)
                    CALL  MOVE_VALVE  (OUTLET,  CLOSED)
                    SET  BWASH_IND.VAL  =  OFF
                CASE  BACKWASH
                    SET  SERVICE_IND.VAL  =  OFF
                    CALL  MOVE_VALVE  (INLET,  TO_DRAIN)
                    CALL  MOVE_VALVE  (OUTLET,  CLOSED)
                    SET  BWASH_IND.VAL = ON
                CASE  FILTERING
                    SET  BWASH_IND.VAL  =  OFF
                    CALL  MOVE_VALVE  (OUTLET,  OPEN)
                    CALL  MOVE_VALVE  (INLET,  TO_FLTR)
                    SET  SERVICE_IND.VAL  =  ON
            END  CASE
            WAIT  UNTIL  (BWASH_TIMER.ALM)
            RESET  BWASH_TIMER  0
    ENDSUBR  FLTR_ACTION

    STATE  SUBR  STARTUP_SYSTEM  START  AND  WAIT
        DECLARATIONS
            VAR  STARTUP_INTERVAL
         CONTINUOUS
            SET  BWASH_TIME_REMAINING.VAL  =  BWASH_TIMER.LIM  -  BWASH_TIMER.VAL
            SET  TIME_TO_NEXT_BWASH.VAL  =  BWASH_TIME_REMAINING.VAL
```

Figure A-7. Example UDF Program (Page 2 of 4)

```
        NORMAL  LOGIC
            IF  (RUN_STOP_PB.VAL  <>  RUNNING)  SET  RUN_STOP_PB.VAL  =  RUNNING
            SET  OPERATOR_MSG.OSTR  =  "FILTER  SYSTEM  START-UP  INITIATED"
            STARTUP_INTERVAL  =  SYSTEM_STARTUP_DURATION.VAL  /  NUMBER_OF_FLTRS

            CALL  FLTR_ACTION  (FLTR1_INLET,  FLTR1_OUTLET,  FLTR1_BWASH_IND,  \
                FLTR1_SERVICE_IND,  BACKWASH,  STARTUP_INTERVAL)
            CALL  FLTR_ACTION  (FLTR1_INLET,  FLTR1_OUTLET,  FLTR1_BWASH_IND,  \
                FLTR1_SERVICE_IND,  FILTERING,  RUNNING)
            SET  OPERATOR_MSG.OSTR  =  "FILTER  #1  START-UP  COMPLETE"

            CALL  FLTR_ACTION(FLTR2_INLET,  FLTR2_OUTLET,  FLTR2_BWASH_IND,  \
                FLTR2_SERVICE_IND,  BACKWASH,  STARTUP_INTERVAL)
            CALL  FLTR_ACTION  (FLTR2_INLET,  FLTR2_OUTLET,  FLTR2_BWASH_IND,  \
                FLTR2_SERVICE_IND,  FILTERING,  RUNNING)
            SET  OPERATOR_MSG.OSTR =  "FILTER  #2  START-UP  COMPLETE"

            CALL  FLTR_ACTION  (FLTR3_INLET,  FLTR3_OUTLET,  FLTR3_BWASH_IND,  \
                FLTR3_SERVICE_IND,  BACKWASH,  STARTUP_INTERVAL)
            CALL  FLTR_ACTION  (FLTR3_INLET,  FLTR3_OUTLET,  FLTR3_BWASH_IND,  \
                FLTR3_SERVICE_IND,  FILTERING,  RUNNING)
            SET  OPERATOR_MSG.OSTR  =  "FILTER  #3  START-UP  COMPLETE"

            NEXT  STATE  SYSTEM_OPERATION
        FAULT  LOGIC
            WAIT  UNTIL  (SYSTEM_STATUS  =  GOOD)
            NEXT  STATE  SYSTEM_SHUTDOWN
    ENDSUBR  STARTUP_SYSTEM

    STATE  SUBR  SYSTEM_OPERATION
        DECLARATIONS
            VAR  NEXT_FILTER
            CONST  FILTER1  =  1,  FILTER2  =  2,  FILTER3  =  3
        CONTINUOUS
            SET  BWASH_TIME_REMAINING.VAL  =  BWASH_TIMER.LIM  -  BWASH_TIMER.VAL
            SET  TIME_TO_NEXT_BWASH.VAL  =  BWASH_INTERVAL.VAL  -  FLTR_INTERVAL.VAL
        NORMAL  LOGIC
            NEXT_FILTER  =  FILTER1
            WHILE  (RUN_STOP_PB.VAL  =  RUNNING)
                WAIT  UNTIL  (DIFF_PRESSURE.VAL  >=  TRIGGER_PRESSURE.VAL)
                RESET  FLTR_INTERVAL 0
                START  FLTR_INTERVAL  (BWASH_INTERVAL.VAL)
                DO  CASE  NEXT_FILTER
                    CASE  FILTER1
                        CALL  FLTR_ACTION  (FLTR1_INLET,  FLTR1_OUTLET,  FLTR1_BWASH_IND,  \
                            FLTR1_SERVICE_IND,  BACKWASH,  BWASH_DURATION.VAL)
                        CALL  FLTR_ACTION  (FLTR1_INLET,  FLTR1_OUTLET,  FLTR1_BWASH_IND,  \
                            FLTR1_SERVICE_IND,  FILTERING,  RUNNING)
                        NEXT_FILTER  =  2
                    CASE  FILTER2
                        CALL  FLTR_ACTION  (FLTR2_INLET,  FLTR2_OUTLET,  FLTR2_BWASH_IND,  \
                            FLTR2_SERVICE_IND,  BACKWASH,  BWASH_DURATION.VAL)
                        CALL  FLTR_ACTION  (FLTR2_INLET,  FLTR2_OUTLET,  FLTR2_BWASH_IND,  \
                            FLTR2_SERVICE_IND,  FILTERING,  RUNNING)
                        NEXT_FILTER  =  3
                    CASE  FILTER3
                        CALL  FLTR_ACTION  (FLTR3_INLET,  FLTR3_OUTLET,  FLTR3_BWASH_IND,  \
                            FLTR3_SERVICE_IND,  BACKWASH,  BWASH_DURATION.VAL)
                        CALL  FLTR_ACTION  (FLTR3_INLET,  FLTR3_OUTLET,  FLTR3_BWASH_IND,  \
                            FLTR3_SERVICE_IND,  FILTERING,  RUNNING)
                        NEXT_FILTER  =  1
```

Figure A-7. Example UDF Program (Page 3 of 4)

```
            END CASE
            WAIT UNTIL (FLTR_INTERVAL.ALM = TRUE)
        ENDWHILE
        NEXT STATE SYSTEM_SHUTDOWN
    FAULT LOGIC
        WAIT UNTIL (SYSTEM_STATUS = GOOD)
        NEXT STATE SYSTEM_SHUTDOWN
ENDSUBR SYSTEM_OPERATION

STATE SUBR SYSTEM_SHUTDOWN
    DECLARATIONS
        VAR SHUTDOWN_INTERVAL
    CONTINUOUS
        SET BWASH_TIME_REMAINING.VAL = 999
        SET TIME_TO_NEXT_BWASH.VAL = 999
    NORMAL LOGIC
        SHUTDOWN_INTERVAL = SYSTEM_STARTUP_DURATION.VAL / NUMBER_OF_FLTRS

        SET OPERATOR_MSG.OSTR = "FILTER #1 SHUTDOWN INITIATED"
        CALL FLTR_ACTION (FLTR1_INLET, FLTR1_OUTLET, FLTR1_BWASH_IND, \
            FLTR1_SERVICE_IND, SHUTDOWN, SHUTDOWN_INTERVAL)

        SET OPERATOR_MSG.OSTR = "FILTER #2 START-UP INITIATED"
        CALL FLTR_ACTION (FLTR2_INLET, FLTR2_OUTLET, FLTR2_BWASH_IND, \
            FLTR2_SERVICE_IND, SHUTDOWN, SHUTDOWN_INTERVAL)

        SET OPERATOR_MSG.OSTR = "FILTER #3 START-UP COMPLETE"
        CALL FLTR_ACTION (FLTR3_INLET, FLTR3_OUTLET, FLTR3_BWASH_IND, \
            FLTR3_SERVICE_IND, SHUTDOWN, SHUTDOWN_INTERVAL)

        SET OPERATOR_MSG.OSTR = "FILTER SYSTEM SHUTDOWN COMPLETED"

        WAIT UNTIL (RUN_STOP_PB.VAL = RUNNING)
        NEXT STATE STARTUP_SYSTEM

ENDSUBR SYSTEM_SHUTDOWN
```

Figure A-7. Example UDF Program (Page 4 of 4)

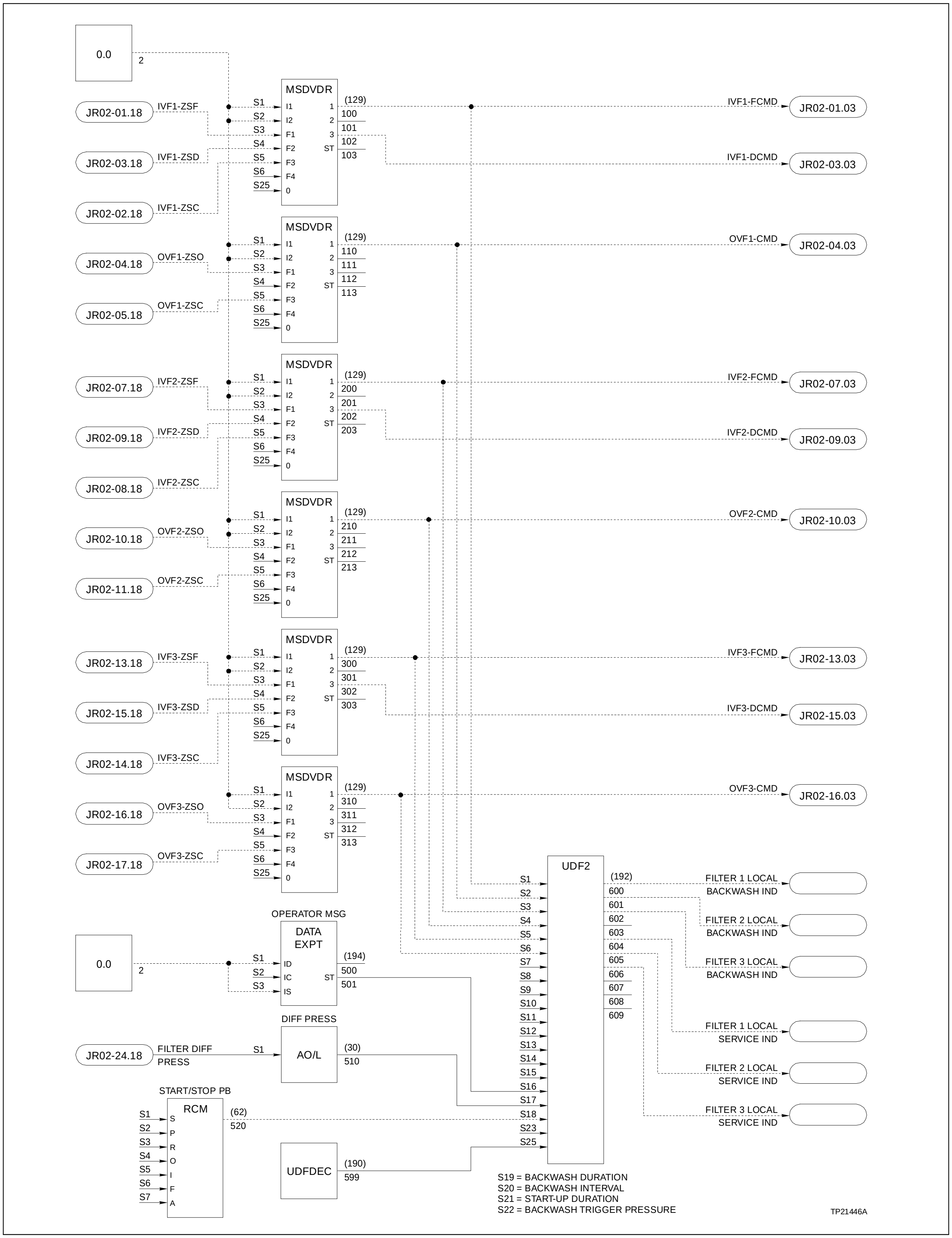


Figure A-8. Example UDF Function Block Configuration

APPENDIX B - STATE TRANSITION DIAGRAM

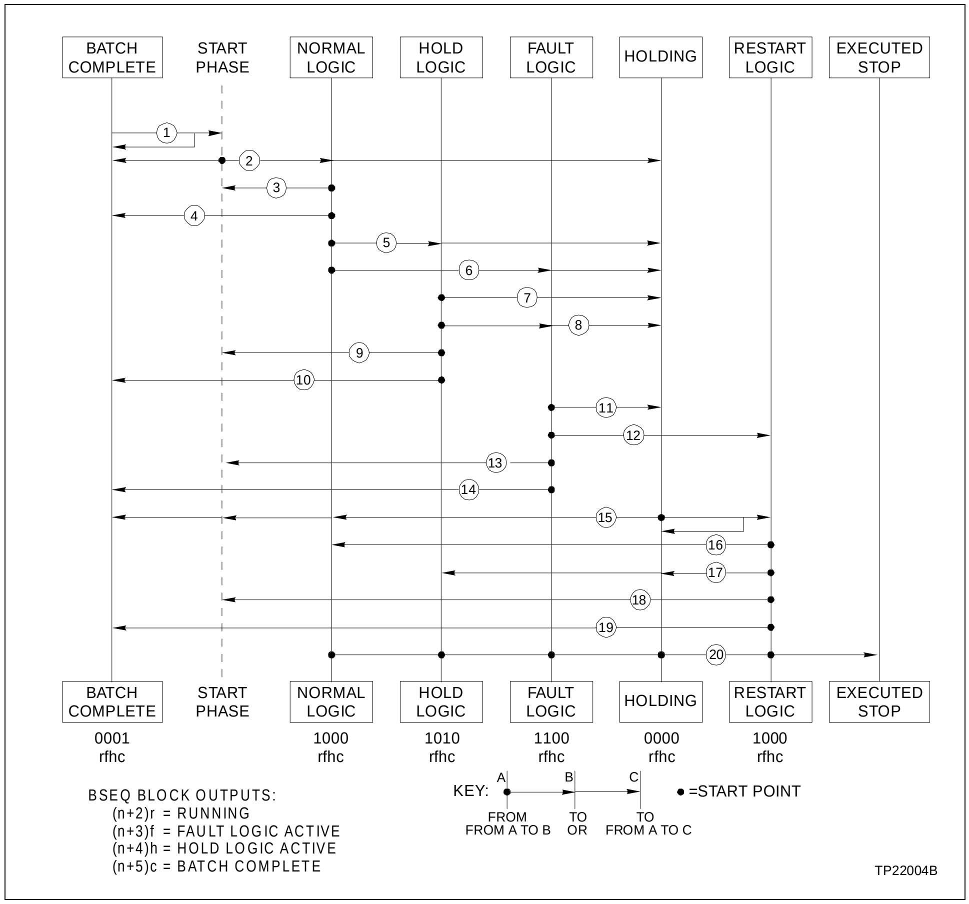


Figure B-1. Example State Transition Diagram

BATCH SEQUENCE STATES

Batch_complete:
Previous batch sequence completed.
All sequential and continuous activity halted.
Program_ID (S9) may be changed.
Object file may be changed.
Recipe_ID input (S1) may be changed.
Unit recipe file may be modified or deleted.
Operation number input (S2) may be changed.

Normal_logic:
For each function block execution cycle:
Execute global monitor subroutines and active data structures (that is, timers, integrators).
For each parallel phase subroutine:

Execute local monitor subroutines and active data structures (that is, timers, integrators).
Execute continuous section.
Execute first/next segment of normal logic.

Hold_logic:
For each function block execution cycle:
Execute global monitor subroutines and active data structures (that is, timers, integrators).
For each parallel phase subroutine:
Execute local monitor subroutines and active data structures (that is, timers, integrators).
Execute continuous section.
Execute first/next segment of hold logic.

Fault_logic:
For each function block execution cycle:
Execute global monitor subroutines and active data structures (that is, timers, integrators).
For each parallel phase subroutine:
Execute local monitor subroutines and active data structures (that is, timers, integrators).
Execute continuous section.
Execute first/next segment of fault logic.

Holding:
For each function block execution cycle:
Execute global monitor subroutines and active data structures (that is, timers, integrators).
Program_ID (S9) may be changed.
Object file may be modified (changes are restricted).
Recipe_ID input (S1) may be changed.
Unit recipe file may be modified.
Operation number input (S2) may be changed.

Restart_logic:
For each function block execution cycle:
Execute global monitor subroutines and active data structures (that is, timers, integrators).
For each parallel phase subroutine:
Execute local monitor subroutines and active data structures (that is, timers, integrators).
Execute continuous section.
Execute first/next segment of restart logic.

Emer_stop:
This is a state that contains all regular states (normal, fault, hold, restart). These states and transitions behave normally except for:
The operation number is 0.
The run input is ignored except in the handling state (that is, the program runs whether the input is 1 or 0).
The following conditions must exist before control can be transferred to a normal operation.

The executed_stop input must be 0.
The run input must be 0 to 1 transition.
Any attempt to transfer to a new operation (end of normal logic, new operation, etc.) without meeting these conditions will cause a transfer to hold_logic (in operation 0).

BATCH SEQUENCE TRANSITIONS

Batch_complete:
(1) Run input (0 or 1) or CSEQ start command.
If executed_stop input is 1:
    set fault code (-13)
    goto--> Batch_complete
If operation number input less than 1:
    set fault code (-15)
    goto--> Batch_complete
Read program object file.
    If error:
        set fault code (-4, -5, -6, -7, -9 or -18)
        goto--> Batch_complete
Check function block references.
    If error:
        set fault code (-16)
        goto--> Batch_complete
Lock unit recipe file and check for compatibility with Batch program.
    If error:
        set fault code (-8, -9, -10, -12, -19, -20, -25)
        goto--> Batch_complete
    If waiting for historian:
        set fault code (-23)
        when historian is ready goto (start operation)---> (2)
Goto (start operation)---> (2)

Start_operation (transition):
(2) If end of unit recipe (operation not in unit procedure):
    goto--> Batch_complete
Read unit recipe data for this operation.
    If error:
        set fault code (-10)
        goto---> Holding
Initialize local data (variables, timers, etc.) for all parallel phase subroutines.
Set up for first statement of normal logic.
Goto---> Normal_logic

Normal_logic:
(3.1) End of normal logic -or- NEXT OPERATION statement.
Increment operation number.
Goto (start new operation).--------------------------------------> (2)
(3.2) New operation statement.

Set operation number to specified value.
Goto (start new operation)---------------------------------------> (2)

Normal_logic:
(4) Done statement.
Unlock unit recipe file.
Goto---> Batch_complete

Normal_logic:
(5) (Run input (0) -or- CSEQ hold command) -and- hold enabled.
Set fault code (-1).
If hold logic is defined:
    set up for first statement of hold logic
    goto---> Hold_logic
else
    unlock unit recipe file
    goto--> Holding

Normal_logic:
(6) Fault statement.
If fault logic is defined:
    set up for first statement of fault logic
    goto---> Fault_logic
else
    unlock unit recipe file
    goto--> Holding

Hold_logic:
(7) End of hold logic.
Unlock unit recipe file.
Goto--> Holding

Hold_Logic:
(8) Fault statement.
If fault logic is defined:
    set up for first statement of fault logic
    goto---> Fault_logic
else
    unlock unit recipe file
    goto--> Holding

Hold_logic:
(9.1) NEXT OPERATION statement.
Increment operation number.
Goto (start new operation)---------------------------------------> (2)
(9.2) New operation statement.
Set operation number to specified value.
Goto (start new operation)---------------------------------------> (2)

Hold_logic:
(10) Done statement.
Unlock unit recipe file.
Goto---> Batch_complete

Fault_Logic:
(11) End of fault Logic.
Unlock unit recipe file.
Goto---> Holding

Fault_Logic:
(12) Restart statement.
Set up for first statement of Restart logic.
Disable fault statement.
Goto---> Restart_logic

Fault_logic:
(13.1) NEXT OPERATION statement.
Increment operation number.
Goto (start new operation)---------------------------------------> (2)
(13.2) New operation statement.
Set operation number to specified value.
Goto (start new operation)---------------------------------------> (2)

Fault_logic:
(14) Done statement.
Unlock unit recipe file.
Goto---> Batch_complete

Holding:
(15) Run input (0-1) or CSEQ start command.
Read program object file.
    If error:
        set fault code (-4, -5, -6, -7, -9 or -18)
        goto--> Holding
If Batch data has changed:
    set fault code (-21)
    goto---> Batch_complete
Check function block declarations in Batch data.
    If error:
        set fault code (-16)
        goto--> Holding
If recipe_ID input has changed:
    goto (cold restart)--> (2)
Lock unit recipe file and check for compatibility with Batch program.
    If error:
        set fault code (-8, -9, -10, -12, -19, -20, -25)
        goto--> Holding
If operation number input has changed,
    goto (cold restart)--> (2)
Read unit recipe information for this operation.
If phase subroutine name has changed:
    goto (cold restart)--> (2)
If active phase subroutine data declarations have changed:
    set fault code (-21)
    goto---> Holding
If restart logic is defined:
    disable fault statement

set up for first statement of restart logic
goto--> Restart_logic
else
set up for first statement of normal logic
goto--> Normal_logic

Restart_Logic:
(16) Resume statement.
Enable fault statement.
Set up for specified restart point.
Goto---> Normal_logic

Restart_logic:
(17) (Run input (0) -or- CSEQ hold command) -and- hold enabled
Enable fault statement.
Set fault code (-1).
If hold logic is defined:
set up for first statement of hold logic
goto--> Hold_logic
else
unlock unit procedure file
goto-->Holding

Restart_logic:
(18.1) NEXT OPERATION statement.
Enable fault statement.
Increment operation number.
Goto (start new operation)--------------------------------------> (2)
(18.2) New operation statement.
Enable fault statement.
Set operation number to specified value.
Goto (start new operation)--------------------------------------> (2)

Restart_logic:
(19) Done statement.
Unlock unit recipe file.
Goto--> Batch_complete

Any state (except Batch_complete):
(20) Executed_stop input (1).
Set operation = 0.
Read unit recipe data for this operation.
If error
set fault code (-10)
goto--> Holding
Initialize local data (variables, timers, etc.). (operation 0)
Set up for first statement of normal logic.
Goto--> Normal_logic
---> (operation 0)
(6) If fault.
If fault logic defined.
set up for statement of fault logic
goto---> Fault_logic

else
goto---> Holding
(4) Done statement or last line of normal logic.
Unlock unit recipe file.
Set fault code (-13).
goto---> Batch_complete

Index

Index (continued)

Index (continued)

V

W

For prompt, personal attention to your instrumentation and control needs or a full listing of Bailey representatives in principal cities around the world, contact the Bailey location nearest you.

Australia
Elsag Bailey Pty. Limited
Regents Park, NSW
Phone: 61-2-645-3322
Telefax: 61-2-645-2212

Brazil
Bailey do Brasil
São Paulo
Phone: 55-11-548-4122
Telefax: 55-11-547-0315

Canada
Elsag Bailey (Canada), Inc.
Burlington, Ontario
Phone: 1-905-639-8840
Telefax: 1-905-639-8639

France
Elsag Bailey S.A.
Massy
Phone: 33-1-64-47-2000
Telefax: 33-1-64-47-2016

Germany
Bailey-F & P Automation GmbH
Overath
Phone: 49-220-473-90
Telefax: 49-220-473-979

Italy
Elsag Bailey
Genoa
Phone: 39-10-658-1
Telefax: 39-10-658-2941

Japan
Bailey Japan Company, Ltd.
Tagata-Gun, Shizuoka-Ken
Phone: 81-559-49-3311
Telefax: 81-559-49-1114

Jordan
Bailey Controls Jordan
Amman
Phone: 962-6-788-116
Telefax: 962-6-756-908

Mexico
Bailey Mexico S.A. de C.V.
Naucalpan
Phone: 52-5-557-6100
Telefax: 52-5-557-7022

Norway
Bailey Norge A.S.
Bergen
Phone: 47-55-222-000
Telefax: 47-55-222-010

People's Republic of China
Bailey Beijing Controls
Beijing
Phone: 86-10-401-0651
Telefax: 86-10-401-1643

Singapore
Elsag Bailey Pte. Ltd.
Singapore
Phone: 65-442-3200
Telefax: 65-442-2700

United Kingdom
Bailey Automation plc
Telford, Shropshire
Phone: 44-1952-670-477
Telefax: 44-1952-670-455

United States
Bailey Controls Company
Wickliffe, Ohio
Phone: 1-216-585-8500
Telefax: 1-216-585-8756

Venezuela
Bailey de Venezuela SA
Valencia
Phone: 58-41-329-196
Telefax: 58-41-327-632

Form WBPEEUI330252A0 Litho in U.S.A. 396

Elsag Bailey
Process Automation

29801 Euclid Avenue • Wickliffe, Ohio 44092 • (216) 585-8500
Telex: 980621 • Telefax: (216) 585-8756 or (216) 943-4609