Foxboro®

by Schneider Electric

I/A Series® System

High Level Batch Language (HLBL) User's Guide

B0400DF

Rev J
December 16, 2014

Invensys is now part of Schneider Electric.

SOFTWARE LICENSE AND COPYRIGHT INFORMATION

Before using the Invensys Systems, Inc. supplied software supported by this documentation, you should read and understand the following information concerning copyrighted software.

1. The license provisions in the software license for your system govern your obligations and usage rights to the software described in this documentation. If any portion of those license provisions is violated, Invensys Systems, Inc. will no longer provide you with support services and assumes no further responsibilities for your system or its operation.
2. All software issued by Invensys Systems, Inc. and copies of the software that you are specifically permitted to make, are protected in accordance with Federal copyright laws. It is illegal to make copies of any software media provided to you by Invensys Systems, Inc. for any purpose other than those purposes mentioned in the software license.

Contents

Figures

Tables

Preface

High Level Batch Language (HLBL) is a PASCAL type language that has been designed to implement process control strategies. HLBL is used to build sequence control strategies and structured text items that are required for sequential process operations.

Purpose

This document describes the High Level Batch Language used to build sequence blocks and Structured text (ST) items. It describes the HLBL language, HLBL language preprocessor, sequence language and complier, and sequence language error codes. Sequence language is a synonym for HLBL.

Contents

The document is divided into the following chapters:

- Chapter 1 “Sequence Logic”
- Chapter 2 “Sequence Language”
- Chapter 3 “HLBL Preprocessor and Complier”
- Chapter 4 “HLBL Statements”
- Appendix A “Sequence Control Error Messages”.

Audience

This document is intended for anyone who needs to use HLBL to build a process control system to control their process. In particular this document describes the language required to build sequence blocks. You should be familiar with process control engineering functions and Integrated Control blocks.

Revision Information

For this revision of the document (B0400DF-J), the following change was made:

- Updated information in the section “Sequence Compiler Limits” on page 53.

Reference Documents

These documents present additional and related information:

- *Control Processor 270 (CP270) and Field Control Processor 280 (CP280) Integrated Control Software Concepts* (B0700AG)
- *Integrated Control Block Descriptions* (B0193AX)
- *Integrated Control Block Descriptions for FOUNDATION fieldbus Specific Control Blocks* (B0700EC)

Most of these documents are available on the Foxboro Evo Electronic Documentation media (K0174MA). The latest revisions of each document are also available through our Invensys® Global Customer Support at *https://support.ips.invensys.com*.

1. Sequence Logic

This chapter covers sequence logic control: sequential control block types and their various states, sequence processing, and SBX programming.

Sequence logic control complements continuous and ladder logic control with regulatory feedback applications at the equipment control level. For example, the sequential control software can be used to supervise a sequence of activities such as filling a tank, blending its contents, and draining the tank.

Whereas most continuous control blocks have fixed algorithms, sequence control blocks have user-defined algorithms. Sequential control software enables you to:

- Define a sequence of events
- Monitor process conditions, taking corrective action when required
- Time events
- Manipulate any compound or block parameter or any shared variable
- Output messages to any logical device or to the Historian.

To introduce sequential control to a control strategy, you must define sequential control blocks and add them to compounds.

Sequence logic is created with the Sequence Language, a subset of the High Level Batch Language (HLBL). The Sequence Language is a high level programming language resembling Pascal, but specifically geared toward creating process control strategies.

Sequential Control Blocks

Sequential control is performed at the compound level through sequential control blocks of the following types:

- Sequence (IND, DEP, and EXC)
- Monitor (MON)
- Timer (TIM).

Sequence Block

There are three Sequence block types:

- Independent (IND)
- Dependent (DEP)
- Exception (EXC).

IND and EXC blocks run independently of other Sequence blocks in the same compound. DEP blocks pause when an EXC block in the same compound is active.

This relationship between DEP and EXC blocks allows you to separate a sequence algorithm for handling normal conditions from a sequence algorithm for handling alarm conditions. For example, if a Monitor (MON) block detects an alarm condition and activates an EXC block to take

corrective action, the DEP block pauses until the corrective action is complete. When the EXC block is done, the DEP block can finish executing its sequence algorithm.

A Sequence block contains a user-defined sequence algorithm. You can use a Sequence block to:

- Manipulate parameters and shared variables
- Change the flow of execution based on the state of parameters and shared variables
- Activate other Sequence and Monitor blocks
- Measure time
- Report to the Historian
- Send information to logical devices, such as printers
- Call a subroutine and pass arguments, if any
- Make calculations
- Simulate a process for testing purposes.

A Sequence block is composed of:

- Standard Parameters
- Block Type Identification
- Symbolic Constants
- Local Block Variables
- User-Labels
- Include Files
- Subroutines (variables and statements)
- Standard Block Exception Handlers
- Block Statements, grouped into Steps.
  Each statement, whether in the block's main section, in its subroutines, or in its standard block exception handlers, may optionally have a label.

Standard parameters show block operation details and allow you to control block operation and connect the block in a control strategy that includes continuous blocks, ladder logic blocks, and other sequence blocks.

Block Type identification is a small block of information at the start of the sequence language file where you provide data such as block name, type, creator, revision level and date.

Symbolic constants are identifiers which represent constant values. They are used to indicate or illustrate the meaning of such values. The constants are an aid in compiling sequence blocks.

Changing the value of a constant in an include file does not affect currently running or already compiled sequence blocks automatically. After such a change, the blocks containing source code in which the constants are used have to be recompiled to effect the change.

Block variables are local and are not accessible from outside the block. You define their number and their size. There are no user-labels for local block variables. Refer to them by their declared names. You can use them in any HLBL expression and you can assign them to each other, to user (array) parameters, and to external references.

Local block variables can be any of the following types:

- Boolean and Boolean array

- Long integer and long integer array
- Real and real array
- String and string array.

  For local variables there are three string lengths: short (6 characters), medium (12 characters), or long (80 characters).

All arrays in the local block variables (main section) and the local subroutine variables may be multi-dimensional, with a maximum of 256 dimensions.

When specifying the types of main section or subroutine local variables, you may use a comma-separated list of variables before the type specification.

Example:
```
THIS_VAR, THAT_VAR, OTHER_VAR: R;
STR_VAR1, STR_VAR2, OTHER_STR: S12;
MY_BOOLS, YOUR_BOOLS: B [5];
```

The type specifications for local block variables are:

B	=	Boolean
I	=	Long Integer
R	=	Real
S	=	String of 80 Characters
S6	=	String of 6 Characters
S12	=	String of 12 Characters

User-labeled parameters can be referenced by the Sequence block's user-defined algorithm. There are a fixed number of each of the following types: real, long integer, Boolean, and string. All types except string and the data store arrays can be linked with parameters in other blocks and compounds and shared variables. The strings and data store arrays are settable but not connectable.

The standard parameters, all of which may be user-labelled, are:

BI0001 – BI0024:	Boolean Inputs
II0001 – II0008:	Long Integer Inputs
RI0001 – RI0015:	Real Inputs
SN0001 – SN0010:	Strings (always up to 80 characters) *
BO0001 – BO0016:	Boolean Outputs
IO0001 – IO0005:	Long Integer Outputs *
RO0001 – RO0015:	Real Outputs *
BA0001[16]:	Boolean Array Data Store *
BA0002[16]:	Boolean Array Data Store *
BA0003[16]:	Boolean Array Data Store *
BA0004[16]:	Boolean Array Data Store *
IA0001[16]:	Long Integer Array Data Store *

RA0001[16]:	Real Array Data Store *
RA0002[16]:	Real Array Data Store *

* Not available in the Monitor (MON) block (see page 4).

Standard and user-labeled parameters are described in the *Integrated Control Block Descriptions* (B0193AX) document.

An include file can be any set of HLBL statements. Use include files to define specific constructs such as sophisticated WAIT loops or complicated expressions of a set of parameters, or to define objects with a global scope such as symbolic constants, subroutines, or standard block exception handlers. You cannot compile include files separately.

The subroutine allows you to specify a general piece of control logic just once and apply it as often as required in the block algorithm. A subroutine is a sequence of HLBL user-defined statements that can be called from the sequence block's main code or from another subroutine. Subroutines can use any HLBL statement except for standard block exception handlers.

You can use a user-defined number of arguments to parameterize a subroutine. The data types of these arguments must be one of the data types supported in HLBL for block parameters and local variables.

You cannot install a sequence subroutine in a station as an independent entity and you cannot access it from outside the sequence block in which it is installed.

An Independent, Dependent, or Exception block may have subroutines. Monitor and Timer blocks do not support subroutines.

A Standard Block Exception Handler (SBX) is a user-specified section of HLBL statements that allows the sequence block to react to an operational error during automatic execution or to an outside interruption during normal block operation.

There are five events for which SBXs can be specified. Two are error handling SBXs:

- User-errors (OP_ERR between 2000 and 3000)
- System errors (all other errors).

The other three are state change SBXs:

- Switch to Inactive
- Switch to Manual
- Switch to Paused.

Sequence language statements define the sequential control algorithm, as specified by the user.

Monitor Block

A Monitor (MON) block contains up to 16 user-defined Boolean expressions called cases. The result of the evaluation of a monitor case is stored in the associated Boolean output parameter. When one of the cases evaluates to true, the MON activates a sequence block (EXC, DEP, IND, or MON). In this way, up to 16 blocks can be activated from the MON block.

A Monitor block is composed of:

- Standard Parameters
- Blocktype Identification
- Symbolic Constants

- User Labels
- Monitor Cases (up to 16).

User-labeled parameters can be referenced by the Monitor block's user-defined algorithm. There are a fixed number of each of the following types: real, long integer, and Boolean. All types can be linked with parameters in other blocks and compounds and shared variables.

Standard and user-labeled parameters are listed and described in the *Integrated Control Block Descriptions* (B0193AX) document.

A monitor case consists of a monitor condition and an optional activation request that is performed when the condition is true.

Example:

```
0001 WHEN level_hi DO :TANK_1:HI_LEVEL_EXC
```

- The case number is 0001.
- The condition is "level_hi", where level_hi is a user-labeled parameter.
- The request is "DO :TANK_1:HI_LEVEL_EXC". "HI_LEVEL_EXC" is the block name for a sequence block in the compound "TANK_1". This block will be activated when the condition "WHEN level_hi" is evaluated true.
- You see TRIPPD on the faceplate.
- The block activates again if the case is still true upon completion.
- BOnnnn output is set to correspond to the active case.
- BOnnnn and TRIPPD stay set as long as activated block is active, even if the block goes to manual.
- ACTPAT parameter determines the active patterns.

Timer Block

A Timer (TIM) block keeps track of time while control strategies are executed. It is composed of standard parameters and four timers. TIM blocks do not contain any Sequence language statements.

Standard parameters show block operation details and allow you to control block operation.

A timer is composed of a real and a Boolean parameter. The Boolean parameter value determines whether the real parameter is updated or not when the block is processed. When the Boolean value is true, the real parameter is updated. When the Boolean value is false, the real parameter is not updated.

The TIM block is processed when the compound in which it resides is On and the block is in Auto. When a TIM block is processed, timers that have been started are updated every scheduled Basic Processing Cycle (BPC). Timers are started by an external source, such as a statement in a Sequence block.

Sequential Control Block States

Block states describe the operational behavior of a block. All blocks have the Application states Manual and Auto.

In addition to the Application states, the sequential control blocks have the following Sequence states: Active, Inactive, Paused, and Tripped. Tripped applies only to the Monitor block. The set of operational modes for sequential blocks include:

Table 1-1. Operational Modes

Application States	Sequence States	Transition
Auto	Active	To_Manual
Semi-Auto (Step Mode)	Inactive	To_Inactive
Manual	Paused	To_Paused
Subr-Trace	Tripped	
SBX-Trace	Suspended on SENDCONF	

The Sequence states and the Application states control sequential control block algorithm execution and the operational state of block outputs.

Application States

The Application states, Auto, Semi-Auto, and Manual, control the operational state of a block's outputs. In conjunction with the Sequence states, they also control sequential control block algorithm execution.

The Application state is determined by the value of the block's MA parameter. When MA is true, the block is in the Auto state. When MA is false, the block is in the Manual state.

Another block parameter, RSTMA, controls the value of the MA parameter when the compound changes from Off to On. When RSTMA is 0, MA becomes false; when RSTMA is 1, MA becomes true; when RSTMA is 2, MA does not change upon the compound switch. You set the value of RSTMA during block configuration.

Auto State

In the Auto state, a block's output parameters are secured. This means that the block algorithm is responsible for updating the output parameters. External sources (other blocks and applications) cannot write values to block output parameters.

Sequential control block algorithms are processed as follows in the Auto state:

- TIM block timers that have been started are updated once every scheduled BPC. A timer is started with a START_TIMER statement in an IND, DEP, or EXC block.
- MON block cases are evaluated each scheduled BPC. If a case trips, it may lead to activation of an EXC block. If the EXC block activated is remote, tripping and untripping may require several BPCs to complete.
- IND, DEP, and EXC blocks process the number of statements specified by the block's BPCSTM parameter each scheduled BPC. When a statement requiring suspension such as WAIT or WAIT UNTIL executes, fewer statements may be processed than the number specified by BPCSTM.

Since Sequence block algorithms vary in length, a block may execute completely in one BPC or it may require several BPCs to execute completely.

Once all statements have been executed, the Sequence block is no longer processed unless a statement in the user-algorithm causes it to repeat.

If the sequence block contains state change logic, that logic will be executed if the block switches from the Active/Auto mode to the Inactive, Manual or Paused state. The logic for the state changes are user-defined in SBXs 3, 4, and 5.

The order of statement execution can be altered while in the Auto state. An operator, at a user-defined or default display, can redirect statement execution to a new start location by writing the desired statement number to the STMRQ parameter.

Semi-Auto State

In Semi-Auto (or Step mode), the sequence block executes only the HLBL statements that belong to a particular step. Statement execution stops when a step boundary is passed. Steps can be requested in any order, at a user-defined or default display, by writing the desired step number to the STEPRQ parameter. The block is divided into steps by means of the step labels in HLBL.

If the sequence block contains state change logic, the corresponding logic will be executed if the block switches from the Active/Step mode to the Inactive, Manual, or Paused state.

The logic for the state changes are user-defined in SBXs 3, 4, and 5.

Manual State

In the Manual state, output parameters are not secured. This means that external sources (other blocks and applications) can write values to the block's output parameters. Unlike continuous control blocks, sequential control blocks may have their own statements write to their own output parameters while the block is in Manual.

Sequential control block algorithms are processed as follows in the Manual state:

- TIM blocks are not processed.
- MON block cases are executed one at a time by user-request. You select a case for execution, from a user-defined or default display, by writing the desired case number to the CASERQ parameter.

If a requested case trips (i.e., the evaluated condition is true), a block activation request is executed. After the case has been processed completely (tripped and untripped), the standard parameter CASENO is set to indicate the number of the next case. The next case is not evaluated unless requested. The TRPCHG parameter is incremented each time a case changes to or from the tripped state.

The processing of EXC blocks already activated by tripped cases in the MON block are not affected by other case evaluation requests to the MON block.

- IND, DEP, and EXC block statements are executed one at a time by request. You can select a statement for execution from a user-defined or a default display by:
  - Writing to the parameter STEPRQ the step number which begins with the requested statement.
  - Writing to the parameter STMRQ the number of the requested statement.
  - Setting NXTSTM to true.

A statement requiring several BPCs to execute, such as a WAIT statement, need only be requested once to initiate execution.

Statement execution can be cancelled by requesting that another statement be executed. The standard parameter STMNO indicates the number of the statement currently executing. When the statement finishes execution, STMNO is set to the number of the next statement dictated by execution flow. That statement is not executed unless requested by:

- Writing its number to the parameter STMRQ.
- Setting NXTSTM to true.

When the requested statement calls a subroutine, all the HLBL statements of that subroutine (and any nested subroutine) are executed. The parameter SUBRNO indicates in which subroutine, if any, the currently executed statements reside. The parameter STMNO indicates the statement number within that subroutine.

The Subr-Trace and SBX-Trace modes enable you to single step through statements of subroutines and SBXs. You can switch the block into one of the Trace modes only when the block is in the Active/Manual state.

Subr-Trace is a substate of the Manual state that enables you to single-step through a subroutine. You enter this substate by selecting the "SUBR TRACE" button in the ALL CODE display. This enters the integer value "1" into the TRACRQ parameter which, in turn, sets the block into the Subr-Trace mode when the block is in Manual. After granting the request, the block resets TRACRQ to 0.

Once in the Subr-Trace mode, you "select" a subroutine by requesting a call-subroutine statement in the block's main section. The block is then idle before the first statement in the requested subroutine. You can then single-step through the subroutine statements by toggling the NXTSTM parameter. STEPRQ and STMRQ cause the execution of a single statement in the block's main section.

When you switch into the SBX-Trace mode, the block environment (i.e., step, subroutine, statement number) is saved. The block returns to this environment when you exit the SBX-Trace mode.

Once in the SBX-Trace mode, you "select" an SBX by setting the SBXRQ parameter to a value of 1 to 4. SBX5 (a switch to Paused) applies only to the DEP block (the block ignores out of range values). When you select an SBX, the block idles at the first statement within that SBX. You can then single-step through the SBX statements by toggling the NXTSTM parameter. The block ignores step- and statement-requests while it is in the SBX-Trace mode.

In the Manual, Subr-Trace, and SBX-Trace modes, the block does not secure its output parameters. External sources (other blocks and applications) can write values to the block's outputs. While the block is in one of these modes, the EXC, DEP, or IND block algorithm can update its output parameters after a step-, statement-, or next-statement request.

To exit from the Trace mode, select the "TRACE" field in the faceplate.

Auto/Manual Transitions

You can change the block Auto/Manual state from external sources such as user-defined and default displays, other blocks, and applications.

If a statement is in execution when you request a state change, the statement's execution is completed as if it had begun in the requested state. Then any following statements are executed as appropriate for the requested state.

NOTE

If one or more cases are making a transition from Active to Tripped (for example, the blocks to be activated are remote blocks) when you change a MON block from Automatic to Manual, then the block activation is completed but the cases do not trip.

Sequence States

The Sequence states, Active, Inactive, Paused, and Tripped, in conjunction with the Application states, control sequential control block algorithm execution. The Sequence states are determined by the values of the block's ACTIVE, PAUSED, and TRIPPD parameters. When ACTIVE is true, the block is in the Active state. When ACTIVE is false, block is in the Inactive state.

Another block parameter, RSTACT, controls the value of the ACTIVE parameter when the compound in which it resides changes from Off to On, or when the control processor in which it resides undergoes a restart operation, as follows:

- RSTACT = 0: ACTIVE is false.
- RSTACT = 1: ACTIVE is true.
- RSTACT = 2: ACTIVE retains the value from the checkpoint file when the Control Processor is restarted, or remains 0 when the compound makes a transition from Off to On.

When a DEP block is in the ACTIVE state, it may also be in the PAUSED state. A DEP block is Paused when the PAUSED parameter is true.

When a MON block is in the Active state, it may also be in the Tripped state. A MON block is tripped when the TRIPPD parameter is true.

Table 1-2. Sequence States

State	Description
Inactive	An IND, DEP, or EXC block is not executing any statements or a MON block is not evaluating conditions.
Active	An IND, DEP, or EXC block is executing statements or a MON block is evaluating conditions.
Paused	A DEP block's execution is suspended because one or more EXC blocks in the same compound are Active. The DEP block remains suspended until all such EXC blocks are done executing.
Tripped	A condition evaluated by a MON block causes it to activate other blocks. The MON block remains tripped until all activated blocks are done executing.

Active State

In the Active state, an IND, DEP, EXC, or MON block is processed. (The TIM block does not have an Active state. It is processed when the compound is On and the Application state is Auto.) How statements are executed depends upon the Application states Auto, Step, and Manual.

Inactive State

In the Inactive state, an IND, DEP, EXC, or MON block is not processed. (TIM blocks do not have an Inactive state. TIM blocks are not processed when the Application state is Manual.)

Active/Inactive Transitions

You can change the block Active/Inactive state from external sources such as user-defined and default displays, other blocks, and applications.

When a linkage to the ACTIVE parameter exists, it is secured. This means that you cannot access the parameter directly. To activate or deactivate the block, you can:

- Access the ACTIVE parameter through the source of the linkage
- Write the number of a non-existing statement to STMRQ.

Writing the number of a non-existing statement to STMRQ directs statement execution to the end of the algorithm. Although the block is in effect deactivated, the ACTIVE parameter remains true until it has been released. When released, it is automatically set to false.

Paused State

In the Paused state, DEP block statement execution is suspended due to active EXC blocks in the same compound. The PAUSED parameter indicates whether a DEP block is in the Paused state. When PAUSED is true, the block is in the Paused state.

Tripped State

In the Tripped state, a MON block has one or more cases tripped. A case trips when it is evaluated as true and activates another block. The TRIPPD parameter indicates whether a MON block is in the Tripped state. As long as at least one case is tripped, the TRIPPD parameter is true; otherwise, TRIPPD is false.

Transition States

The Transition states, To_Inactive, To_Manual, and To_Paused, are intermediate states that the Sequence Control block assumes while the block is executing one of the three standard block exception handlers (SBXs 3, 4, and 5) that are provided for state change handling.

To_Inactive State

The To_Inactive state is an intermediate state that an IND, DEP, or EXC block assumes while SBX 3 is executing. SBX 3 is the user-defined, user-enabled response to an externally initiated change of block state from the Active/Auto (or Active/Step) state to the Inactive state.

The MON and TIM blocks do not have a To_Inactive state since they do not contain SBXs.

To_Manual State

The To_Manual state is an intermediate state that an IND, DEP, or EXC block assumes while SBX 4 is executing. SBX 4 is the user-defined, user-enabled response to an externally initiated change of block state from the Active/Auto (or Active/Step) state to the Manual state.

The MON and TIM blocks do not have a To_Manual state since they do not contain SBXs.

To_Paused State

The To_Paused state is an intermediate state that a DEP block assumes while SBX 5 is executing. SBX 5 is the user-defined, user-enabled response to an externally initiated change of block state from the Active/Auto (or Active/Step) state to the Paused state.

The IND and EXC blocks do not have a To_Paused state since they do not have a Paused state. The MON and TIM blocks do not have a To_Paused state since they do not contain SBXs.

Compound Sequence State

The collective operational state of all sequential control blocks in a compound is represented by the compound parameter SSTATE. SSTATE can be one of three values:

- SSTATE = 0 (Inactive)
- SSTATE = 1 (Active)
- SSTATE = 2 (Exception).

See Table 1-3 for further definition.

Table 1-3. Compound Sequence State

SSTATE Value	Description
Inactive (SSTATE = 0)	All MON blocks and all Sequence blocks (IND, DEP, or EXC) in the compound are Inactive.
Active (SSTATE = 1)	One or more MON blocks and/or one or more Sequence blocks (IND and DEP) in the compound are Active. No EXC blocks are active.
Exception (SSTATE = 2)	One or more EXC blocks in the compound are Active. IND blocks may be active; TIM blocks may be running.

When a compound is switched OFF, all the sequence blocks in that compound go to the Manual state, thereby releasing their output parameters.

Sequence Processing

Sequential control blocks are processed every scheduled Basic Processing Cycle (BPC) as defined for the Control Processor in which they operate. The following figure shows the processing order within a scheduled BPC.

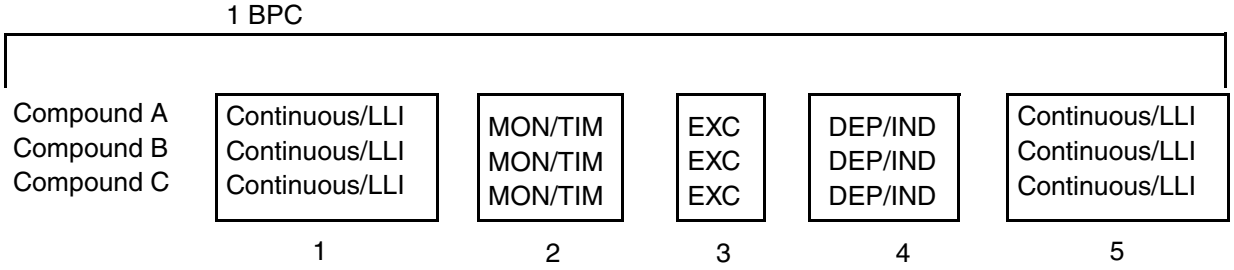

Figure 1-1. Processing Order Within a Scheduled BPC

The scheduled BPC is determined by the block parameters PHASE and PERIOD. The PHASE parameter specifies when a block should be executed relative to the other blocks in that PERIOD. The PERIOD parameter specifies how frequently a block should be executed.

When a sequential control block is activated, it begins executing its block algorithm in the next scheduled BPC as defined by the block parameters PERIOD and PHASE.

When a MON or TIM block is processed, its entire algorithm is executed each scheduled BPC until deactivated.

When an IND, DEP, or EXC block is processed, a specified number of statements in the algorithm are executed each BPC. The number of statements processed is determined by the value of the block's BPCSTM parameter. When the last statement is executed, the block automatically deactivates itself.

Sequence language statements are executed in the order programmed by you. Statement execution continues in a given BPC until:

- The number of statements specified by the BPCSTM parameter are executed in Auto, or in Step mode
- The last statement of a step is executed in Step mode
- One statement is executed by request in Manual for an IND, DEP, or EXC block.

The number of statements executed in automatic can be less than the number specified by the BPCSTM parameter if an executed statement requires more than one BPC to complete. For example:

- A statement makes a full pathname request to access a block parameter in a remote Control Processor.
- A SENDCONF is executed.
- A WAIT or WAIT UNTIL statement has a wait condition exceeding the BPC.

Sequence Processing Overrun

If the block processor cannot process all scheduled blocks in a BPC, there is BPC overrun. In a serious system overload, sequence (and continuous) blocks are always processed, even if it delays the next BPC for the remaining blocks not yet processed.

Useful Hints on SBX Programming

Operational Error SBXs

The sequence Interpreter has comprehensive error protection. In case of any operational error, the parameter OP_ERR assumes an integer number that indicates the kind of failure.

If OP_ERR has a value larger than 3000, a serious error has occurred. When a serious error has occurred, you must be careful when defining SBX (exception logic) TO_SYS_ERROR. An attempt to continue to run or retry the same statement may cause memory violation (for CP30, CP40, or CP60), memory corruption, or unpredictable control algorithm behavior. The nature of this kind of error is usually a software or hardware problem. Therefore, you should use basic logic (no subroutine calls and external references) to write OP_ERR, SBRNO, SBXNO, BLOCK_STMNO, and STMNO, then stop the block, collect any related information, and report the information to your local customer service representative.

All other error conditions, including user-errors (SBX TO_USR_ERROR: 2000 < OP_ER < 3000), allow retrying the same statement. (If you do this, you should be careful to avoid infinite loops.) For most of the cases the “retry” does not clear the error.

Example:

OP_ERR 2314 “string type expected in string expression” does not go away after retry. The important thing to remember is that if you decide to continue you may end up with unexpected results.

```
LOC_VAR, STR1 : S;
STATUS        : B;
LOC_VAR    := ":CONTROLLER_1.BI0001";
STATUS     := :'LOC_VAR';
STR1       := ":MAIN_LOOP.BO0001";
LOC_VAR    :=  STRING  :'STR1';
:'LOC_VAR' := TRUE;
```

STR1 is not a FPN (full path name) for a string type parameter. In this case LOC_VAR keeps its value unchanged and CONTROLLER_1 BI0001 parameter is set to “TRUE”.

SBX RETRY Logic

If you want the same number of retries on any HLBL statement which caused an operational error, you must build the logic to properly reset the retry counter. It is important to remember that each HLBL statement can produce more then one error. Therefore, you should choose a proper limit for retries since the RETRY instruction repeats the entire statement not just the faulty element. An example of the proper implementation of the “retry” is as follows:

```
RETRY_CNT   : I;  {* to count retries on individual statements *}
SAVED_STMNO : I;
SAVED_STMNO := -1;  {* initialization to guarantee miscompare with
   BLOCK_STMNO *}

  BLOCK_EXCEPTION <TO_SYS_ERROR or TO_USR_ERROR>_SBX
IF OP_ERR <> ERR1 AND OP_ERR <> ERR2 AND OP_ERR <> ERR3 THEN
        GOTO FINAL;
{* ERRx are errors to retry. For TO_SYS_ERROR SBX the statement
   must be replaced  with IF OP_ERR > 3000 THEN GOTO FINAL ENDIF;
*}
ENDIF
IF BLOCK_STMNO <> SAVED_STMNO   THEN
   RETRY_CNT    := 0;
   SAVED_STMNO  := BLOCK_STMNO;
ENDIF;
IF RETRY_CNT < LIMIT THEN
   RETRY_CNT := RETRY_CNT + 1;
 {* In some cases, you may want to add the “WAIT” statement at this
point.   For example, OP_ERR ‘-1’ or ‘-45’
 *}
RETRY;
ENDIF;
  <<FINAL>>
SENDMSG (“ERROR! op_err=”,OP_ERR,” stm=”, BLOCK_STMNO,
    “ SBXNO=”,SBXNO) to SN0001;    {* or to MSGGRx *}
```

```
EXIT;
  ENDEXCEPTION
```

The above SBX design assumes that SAVED_STMNO must be set to '-1' (initialized):

1. In the first statement of the:
   - Main code section
   - Each subroutine
   - State change SBX (TO_INACTIVE, TO_MANUAL, TO_PAUSED).
2. In the last statement of the state change SBX unless the logic terminates the block.
3. In the first statement after any label which is used as the "GOTO" statement argument to change execution flow backwards or to jump from any type of SBX to the block main section.

Operational Error Conditions Inside Subroutines

Presently in an SBX there is no information about the subroutine number and statement number of a subroutine which has triggered an SBX execution. BLOCK_STMNO always refers to the main section. In the SBX the STMNO refers to the current SBX statement, SUBRNO is 0 unless the SBX itself calls a subroutine. To make the retry logic work properly, SAVED_STMNO must be reset before each subroutine statement which may trigger the SBX retry logic. In most cases, this results in a significant overhead. An alternative is to use a more elaborate approach which is primarily based on an assumption that the retries make sense only on statements with external references. Related OP_ERRs which may be fixed with retries are:

- All negative values. They are related only to external references outside the CP or to targets which do not exist.
- "3": Attempt to write a secured parameter (for example, an output parameter while the block is in AUTO).
- "18": Locked access (for example, the OWNER parameter of a block).
- "2401": Attempt to set the ACTIVE parameter to a value that is already assumed.

The idea of the approach is to reset SAVED_STMNO inside subroutines before each statement with:

- External references (including ACTCASES, ABORT and ACTIVATE statements) to a remote station database.
- Any of external sets including ABORT, and ACTIVATE statements which may produce OP_ERR 3, 18, or 2401 if retry is expected.

If speed and memory are not major issues, instead of resetting the SAVED_STMNO multiple times for the kind of statement listed above, you may program HLBL logic to avoid usage of this statement more than once inside of each subroutine. To remove multiple external references from the subroutines, you can create a set of SET_VALx and GET_VALx subroutines to be called when needed.

Example:

For SET_VALx() (where 'x' stands for R,I,B,S data type)

```
SUBROUTINE SET_VALR(IN FPN : S; INOUT VAR : R)
STATEMENTS
 SAVED_STMNO := -1;
 VAR := :'FPN';
```

```
ENDSUBROUTINE

SUBROUTINE SET_VALI(IN FPN : S; INOUT VAR : I)
STATEMENTS
 SAVED_STMNO := -1;
 VAR := :'FPN';
ENDSUBROUTINE

SUBROUTINE SET_VALB(IN FPN : S; INOUT VAR : B)
STATEMENTS
 SAVED_STMNO := -1;
 VAR := :'FPN';
ENDSUBROUTINE

SUBROUTINE SET_VALS(IN FPN : S; INOUT VAR : S)
STATEMENTS
 SAVED_STMNO := -1;
 VAR := STRING :'FPN';
ENDSUBROUTINE
```

NOTE

To continue after unsuccessful retries, you can secure the results by writing the default value into VAR. Example:

```
SUBROUTINE SET_VALB(IN FPN  : S;
 INOUT VAR : B;
 IN DFLT : B )
STATEMENTS
 SAVED_STMNO := -1;
 VAR := DFLT;
 VAR := :'FPN';
ENDSUBROUTINE
```

Operational Error Condition Inside State Change SBX

Presently in an operational error SBX there is no information about the state change SBX which has triggered an operational error SBX execution. The situation is exactly the same as described for subroutines. All recommendations for the retry logic which were given before are suitable for the state change SBXs.

Complex Statements

You should avoid complex HLBL statements such as the following:

```
SAVED_STMNO := -1;   {* the initialization is not needed for the main
code section.  *}
IF :COMP1:BLOCK1.BO0001 <> :COMP2:BLOCK2.BO0002 THEN
  ...........
ENDIF
```

This statement can produce OP_ERR “-1” twice and RETRY_CNT is not reset for the second one. It is preferable to avoid usage of two or more external references in the single statement. You should use extra assignment statement instead:

```
SAVED_STMNO := -1;
TMP_BOOL := :COMP2:BLOCK2.BO0002;
SAVED_STMNO := -1;
IF :COMP1:BLOCK1.BO0001 <> TMP_BOOL THEN
  ............
ENDIF
```

You may also double the number of retries by setting a proper value for LIMIT (the variable LIMIT was used in the above example for SBX programming).

Usage of the SUBR_LEVEL Standard Parameter

Some applications use SBX logic to decide to “retry” or to “not retry” a particular HLBL statement on a particular OP_ERR. Those applications may need more precise identification for subroutines which triggered the SBX. While the subroutine statement identification is the user’s responsibility, there are ways to identify the subroutine.

The subroutine can be identified in the SBXs by the BLOCK_STMNO and SUBR_LEVEL standard parameters in cases when:

- HLBL logic does not have subroutine calls from other subroutines (the maximum nesting level is one).
- All subroutines have not more than one call of another subroutine (no limits on the nesting level).

Remember that if the SBX does the identification inside a subroutine (the SBX calls another subroutine), the SUBR_LEVEL is incremented by one.

If subroutines are called from the state change SBXs and the block main section, you must set the special flag at the beginning of the each state change SBX to be used in the operational error SBX to ensure the correct subroutine identification. The flag must be cleared at the end of the state change SBX.

HLBL and SFC Platform Compatibility

Refer to the “HLBL Migration Considerations” and “SFC Migration Considerations” tables in the latest version of the Foxboro Evo Control Core Services or I/A Series software release notes for an overview of the issues you may encounter when migrating High Level Batch Language (HLBL) and Sequential Function Chart (SFC) files between platforms and configurators, as well as general considerations for migrating sequence block code. You may also want to refer to the “HLBL Code” and “SFC Code” sections in these release notes for issues with HLBL and SFC code that may appear when migrating code between configurators and operating systems.

2. Sequence Language

This chapter contains information on Sequence language including block algorithm structure, sequence language data types, declaration of user-labeled parameters, and sequence language character set, vocabulary, and expressions. Also included in this chapter is information on data type checking, reference connection checking, operational errors, the bad status attribute, preprocessor directives, sequence compiler limits, and the sequence compiler limits table.

The Sequence language is a high level language similar to the Pascal programming language, but designed to implement process control strategies. Sequence language is a synonym for High Level Batch Language (HLBL).

The Sequence language is used to write Sequence (IND, DEP, and EXC) and Monitor (MON) block algorithms. The three Sequence blocks support all HLBL statements. Timer (TIM) blocks do not have Sequence language code.

Algorithm Structure

The block algorithm is composed of:

- The heading
- The subroutines section
- The standard block exceptions section
- A main statement section.

Heading

The heading section contains:

- Block type identification
- Definition of constants
- Definition of block local variables
- User-labeled parameter declarations.

The "Block Type Identification" is the identifier at the first line (`Dependent_sequence`, `Independent_sequence`, or `Exception_sequence`).

Constants, variables, and user-labels can take any name normally used for standard block parameters, for example, ACTIVE, PAUSED, BLKSTA, and so forth, except for the following five names:

- OP_ERR
- STMNO
- SUBRNO
- SBXNO
- STEPNO.

The “Definition of Constants” section of the sequence block definition enables you to introduce names for numerical values. A constant is an identifier which acts as a synonym for a value. The definition of constants is the introduction of such synonyms.

Constants are known throughout the whole block: in the statements of all subroutines, standard exception handlers and the block’s statement section.

Constants are an aid in compiling Sequence blocks.

Changing the value of a constant in an include file does not automatically affect a previously compiled sequence block. To effect the value change, you must recompile the block.

The values of the block local variables can be scalars, or arrays, of data type boolean, real, integer, or string. Strings can be 6, 12, or 80 characters. See Table 3-3 for local variable related limits.

Variables provide local data storage for values, or intermediate results, which are only accessible from the block in which they are declared. Block local variables are known throughout the whole block: in the statements of all subroutines, standard exception handlers and the block’s statement section. Local variables should be initialized by the programmer because they are not initialized by the run-time environment.

The “User-Labeled Parameters” section of the sequence block definition enables you to attach a specific name to a “usable parameter.” A “usable parameter” is a Sequence control block parameter that is available to you. See Table 3-3 for the maximum length of a user-parameter label.

The assigning of a user-label to a usable parameter is called a “parameter declaration.” User- labels are optional, and the labeled parameters can be referenced by either the user-label or the parameter name.

User-labels cannot be used as labels of labeled statements or steps, to which you can jump by means of a GOTO statement.

Subroutines

Subroutines are supported in Independent, Dependent, and Exception blocks. The Monitor and Timer blocks do not have subroutines.

A Sequence subroutine is a sequence of user-programmed HLBL statements that can be called from the Sequence block’s main code section, or from another subroutine. Recursive subroutines, however, are not supported. Subroutines cannot call themselves, directly or indirectly. See Table 3-3 for subroutine related limits.

Unlike control blocks, Sequence subroutines:

- Are not control objects
- Cannot be installed in a station as an independent identity
- Do not have parameters that can be linked, connected, or accessed from the outside in any way.

Each subroutine name uses a number of bytes equal to one plus the number of characters in the name (for example, SUBROUTINE TIMER uses six bytes).

The compiler numbers the subroutines in the order of their definition within the Sequence block.

The structure of a subroutine consists of:

- A heading
- The local subroutine variables
- A body of HLBL statements.

The heading specifies the subroutine's name and formal arguments, if any.

A subroutine without arguments can be used for different actions upon the same set of parameters or block variables.

A subroutine with arguments can be used for repeating the same action on different sets of parameters.

Subroutine local variables are optional.

The body contains the HLBL statements that define the algorithm.

The subroutine allows you to specify a general piece of coded control logic, only once within a block, and apply it repeatedly by simply referring to the subroutine name. This saves programming effort and gains reliability and consistency.

Subroutines can be used in equipment-oriented applications by using string type arguments. The subroutine can then be called with the name of an object that represents or controls a piece of equipment. For instance, a subroutine that contains the logic for opening a valve in a particular way can be called for INLETVALVE1, for INLETVALVE2, and for INLETVALVE3, where these names refer to I/O blocks that reside in the same compound as the Sequence block that calls the subroutine label.

You can use subroutines to calculate long integer values for the Monitor block's ACTPAT using the new rules for this 32-bit value.

You can save defined objects, including subroutines, as include files, and make them available to all configured blocks.

Subroutine Heading and Formal Arguments

The heading specifies the name of the subroutine and the formal arguments (if any).

Formal arguments are defined at subroutine definition. The data types of the formal arguments can be boolean, long, real, or string (scalars and/or arrays). Whole arrays can be passed as arguments to subroutines.

Formal arguments are known and accessible within the subroutine's statements only. References to formal arguments can only appear within the subroutine for which they are defined.

There are two categories of formal arguments: IN and INOUT. If no category is specified upon definition, the default category, INOUT, is assigned. The IN argument provides the subroutine with a value which is used inside the subroutine. The value can be changed locally within the subroutine but it is not made known outside the subroutine. Use the INOUT argument to return values to the subroutine caller.

When a formal argument is used as the left value in an assignment, the resulting value is only known to the caller if that argument is an INOUT argument.

When you define a formal argument, either IN or INOUT, to be an array type, its actual counterpart in the CALL statement:

- Must be a reference to an array item (the formal argument is a destination array when applying the array assignment rules)
- Cannot be an expression
- Can be an external reference (for IN arguments only).

Usable parameters and block local variables are known within the subroutine. They can also be passed to the subroutine by means of arguments.

Subroutine arguments can take any name normally used for standard block parameters, for example, ACTIVE, PAUSED, BLKSTA, and so forth, except for the following five names:

- OP_ERR
- STMNO
- SUBRNO
- SBXNO
- STEPNO.

Actual arguments are provided with the CALL statement. Refer to “Calling the Subroutine” on page 21 for greater detail on actual arguments.

Subroutine Local Variables

This section of the subroutine definition enables you to define the subroutine’s local variables. Subroutine local variables are optional. They are only known and accessible within the subroutine in which they are defined. They cannot be accessed in the block’s statements section, or from another subroutine, or by means of GETVAL or SETVAL from outside the block.

Subroutine local variables provide local data storage for values, or intermediate results. When a subroutine is called, the initial values of its local variables are undetermined. Local variables should be initialized by the programmer because they are not initialized by the run-time environment.

Subroutine local variables can be multi-dimensional and their number and size are user-definable. Permitted data types of subroutine local variables are the same as for block local variables: boolean, long, real, or string (scalar or array).

Subroutine local variables can use all names normally used for standard block parameters, for example, ACTIVE, PAUSED, BLKSTA, and so forth, except for the following five names:

- OP_ERR
- STMNO
- SUBRNO
- SBXNO
- STEPNO.

Local subroutine variables are referenced by their declared names. There are no user-label names for subroutine local variables.

Subroutine variables can take the same name as block variables. When they do, the subroutine does not recognize the block variable.

Like the local Block variable, the local subroutine variable can be used in any expression or assignment within the logic of the subroutine. You can transport the value outside the subroutine by assigning it to a user-parameter, a local variable, or an INOUT argument.

Subroutine Statements

All HLBL statements can be used in the subroutine body. Their syntax and semantics are the same as if they were used within the block code. Control flow statements such as GOTO and EXIT-LOOP are restricted to the environment of the subroutine, that is, “You cannot jump out of a subroutine.”

Within a subroutine, you can reference user-input, user-output, and user-array parameters by their standard names or by their user-labeled names.

A subroutine has a single exit point. Multiple return statements within a subroutine are not possible. The only way to leave the subroutine code is the ENDSUBROUTINE statement.

HLBL statements are numbered from one for each subroutine, for each SBX, and in the block's main code section. You can execute the subroutine statements one at a time when the block is in the Subr-Trace mode.

Calling the Subroutine

The CALL statement serves to execute the subroutine denoted by the subroutine name. The CALL statement can contain a list of actual arguments which are substituted in place of their corresponding formal arguments defined in the subroutine definition. The correspondence is established by the position of the arguments in the list of formal and actual arguments, respectively.

When called, a subroutine executes its statements sequentially while its calling statement waits for the subroutine to complete.

Actual Arguments

When called, subroutines are supplied with a list of actual arguments which are substituted for the corresponding formal arguments. The data types of the actual arguments should match the formal ones.

The number of elements of an actual argument of type array (parameter or local variable) should match the number of elements of the corresponding formal argument exactly. The corresponding formal argument represents this actual argument during the entire execution of the subroutine.

When the CALL statement includes array references for formal arguments of the IN category, all elements of the actual array are physically copied into the subroutine's data space. If the reference is for a formal argument of the INOUT category, only the address of the actual array is copied.

When a subroutine is defined as having a formal INOUT argument, the actual argument included in the subroutine call must be an identifier referring to:

- A block user-parameter (input, output, string, array)
- A local block variable
- In the case of nested subroutines, a formal argument
- In the case of nested subroutines, a local subroutine variable.

An actual INOUT argument cannot be:

- An item of data type string with a subset specification
- An external reference
- A constant
- A literal
- An expression.

Upon call, the actual counterpart of an IN argument can be:

- A constant
- A name of:
  - A user-parameter (input, output, string, array)

- A local block variable
- In the case of nested subroutines, a formal argument
- In the case of nested subroutines, a local subroutine variable

- An external reference
- An expression that combines the categories listed above.

Any arithmetic expression is allowed for actual IN arguments, including external references.

Full blown string expressions are not allowed. Only string literals can be specified as IN arguments. When a string expression is used as an IN argument, the comma denoting a concatenation of string elements cannot be distinguished from a comma which separates the arguments of a subroutine.

The corresponding formal argument represents a local variable of the called subroutine, and the current value of the expression is initially assigned to this variable.

If an actual argument is an array element, the index expressions are evaluated when the subroutine is called.

Subroutine Examples

Subroutine with No Arguments

From the subroutine definition part of a Sequence block listing:

```
SUBROUTINE DATA_IN ()
{************************************************************
* Purpose: Retrieve data from an identified compound       *
* Arguments: - none                                        *
* Global variables used in the Subroutine                  *
*     COMP_ID - String containing the compound name for    *
*        the data to be retrieved - pre-assigned by the    *
*        calling program.                                  *
*                                                          *
* Global variables modified by the Subroutine              *
*               CF - Compound Flow                         *
*        Valve-Out - Compound valve output                 *
*              LAI - Low Absolute Flow Alarm               *
*         Dev_Shut - Deviation Shutdown required           *
************************************************************}
STATEMENTS
  CF:= :'COMP_ID':FLOW_BLK.RO01;
  Valve_Out:= :'COMP_ID':PID.OUT;
  LAI:= :'COMP_ID':INFLOW.LAI;
  Dev_Shut:= :'COMP_ID':PID.MEASHI OR :'COMP_ID':PID.MEASLI;
ENDSUBROUTINE
The Main STATEMENTS Section includes the Subroutine Call statement/s
for the DATA_IN subroutine.
.......
COMP_ID:= “BLEND1”;
CALL DATA_IN ();
.......
.......
COMP_ID:= “BLEND2”;
CALL DATA_IN ();
.......
```

```
.......
ENDSEQUENCE
```

Subroutine with Arguments

NOTE

The argument flow is, by default, an INOUT argument. The arguments Comp_Id and Valve_No are here designated as IN and INOUT, respectively.

From the subroutine definition part of a Sequence block listing:

```
SUBROUTINE RINSE_FLOW (IN  Comp_Id : S12
                       INOUT Valve_No: I
                       Flow : R )
{************************************************************
* Purpose: Retrieve flow-rate from an identified valve     *
* Arguments: Comp_Id - Compound name                        *
*            Valve_No - Valve number within compound        *
*                Flow - Flow Rate through the valve         *
*                                                           *
* Global variables used in the Subroutine                  *
*    Comp_Proc - TRUE when compounds are being processed   *
*                                                           *
* Global variables modified by the Subroutine - none       *
************************************************************}
STATEMENTS
    IF Comp_Proc AND :'Comp_Id'.ON THEN
        IF Valve_No = 1 THEN
            Flow:= :'Comp_Id':FLOW1_PID.OUT;
        ELSEIF Valve_No = 2 THEN
            Flow:= :'Comp_Id':FLOW2_PID.OUT;
        ELSEIF Valve_No = 3 THEN
            Flow:= :'Comp_Id':FLOW3_PID.OUT;
        ENDIF;
    ELSE
        Flow:= -1;
    ENDIF;
ENDSUBROUTINE
The Main STATEMENTS Section includes the Subroutine Call statement/s
for the RINSE_FLOW subroutine.
......
CALL RINSE_FLOW (“RINSE_1”, 3, 0);
.......
.......
CALL RINSE_FLOW (ProcName, 2, 0);
.......
ENDSEQUENCE
```

In the first call, the formal argument, Comp_Id, takes the literal string, RINSE_1. In the second call, Comp_Id takes the literal assigned to the string variable, ProcName. If the actual argument, ProcName, has the value RINSE_1 when the subroutine is called, the two CALL instructions perform identical operations.

String expressions such as PlantName,“RINSE_1” or PlantName, ProcName cannot be used because the comma separating the entities is interpreted as a delimiter between actual arguments.

Subroutine Trace Mode

In the Subr-Trace mode, you can single-step individual statements of subroutines. You enter the Subr-Trace mode by writing the value 1 to the parameter TRACRQ. The mode is indicated by the parameter TRACMD.

When a block is in Subr-Trace mode, and a CALL statement is requested, statement execution is redirected just before the first statement within a subroutine. You can then single-step through the subroutine statements by toggling the parameter NXTSTM. If a subroutine contains a CALL statement then the statements of the nested subroutine can be single-stepped.

The parameter STMRQ always refers to statements within the block’s main code section. This enables you to redirect statement execution from a (nested) subroutine statement to a block statement by setting STMRQ.

When the block parameter SUBRNO is set to denote the currently called subroutine, the parameter STMNO shows the number of the current HLBL statement within the subroutine:

- During execution of the subroutine statements
- During suspension of a subroutine statement
- When a subroutine statement yields an operational error.

The statements of each subroutine are numbered starting from one. Upon a switch to Auto, block execution proceeds starting at the statement that was reached in Subr-Trace mode.

Standard Block Exception Handlers (SBXs)

Five new sections of the sequence block definition consist of standard block exception handlers, SBXs. In an SBX, logic can be specified for execution after detection of a specific event. Such an event can be an operation error (user-errors and system errors) or a change of the block’s state to Inactive, to Manual, or to Paused.

Main Statement Section

The main statement section contains HLBL statements that specify the sequence block’s actions.

You must declare, in the heading section, any user-labeled parameters addressed in the statement section. You do not need to declare standard parameters and shared variables from other blocks and applications (refer to “References” on page 29).

Statement labels enable the program to redirect to a specific statement by referencing the statement label in a GOTO statement, or in a GOTO clause in statements like SENDCONF, WAITUNTIL, or ACTIVATE. A statement label is an identifier enclosed by << and >>, and placed before the HLBL statement. You can also use statement labels within subroutines and SBXs. See Table 3-3 for label related limits.

Example:

```
<<FLOW_MEAS>> RI0001 := Flow;
```

enables the program, with the statement GOTO FLOW_MEAS, to redirect to the HLBL statement, `RI0001 := Flow;`.

Steps and Step Labels

The block's main code section is divided into steps. Steps cannot overlap and they are contiguous. All the HLBL statements within the block's main code belong to a step.

By default, the block's main section is divided into two steps: INIT_STEP and END_STEP. INIT_STEP, starting at the first statement, contains all the statements of the block's main code section up to the first statement of the next step. END_STEP consists of the last statement of the main code section, which is always the ENDSEQUENCE statement. You need not define the step labels INIT_STEP and END_STEP before referencing them.

You can subdivide the block's main code section into more steps by labeling HLBL statements with step labels, making labeled statement the first statement in the new step. The step ends with the next step label.

Subroutines and SBXs cannot be subdivided into steps.

Step labels enable the program to redirect to a specific group of statements by referencing the step label in a GOTO statement, or in a GOTO clause in statements like SENDCONF, WAITUNTIL, or ACTIVATE. A step label is an identifier enclosed by $$ and $$, and placed before the first HLBL statement in the step.

Example:

```
$$Draining$$ RI0001 := Flow;
```

enables the program, with the statement GOTO DRAINING, to redirect to the HLBL statement, `RI0001 := Flow;`.

Step labels can be used in GOTO commands in the same way as statement labels. The block parameters STEPMD, STEPRQ, and STEPNO refer to the steps defined by these step labels.

Step labels also define the group of HLBL statements to be executed when block operates in the Semi-Auto (or Step) mode.

Data Types

The Sequence language uses the following data types: real, long integer, integer, Boolean, string, and enumerated data type. All data manipulated in the Sequence language must be of one of these data types. The data type determines the type of values that an object can be as well as the operations that can be performed on the value.

Real data can assume values that are approximations of the real numbers. Real values have an accuracy of seven digits and they are limited to the range -3.402824^{38}...$+3.402824^{38}$.

Integer data can have values that express whole numbers. Operations that can be performed are addition, subtraction, multiplication, division, and testing on relational ordering of values. Integer values are limited to the range (-2^{31})... $(+2^{31}-1)$.

There is data conversion between real numbers and integers. Real values can be written to long, short, or integer parameters. Conversely, integer values can be written to real parameters.

Boolean data can assume two values: TRUE and FALSE. These values can be manipulated by the operators AND, OR, and NOT. They can be compared for equality with other Boolean values.

String data type variables and parameters can be used in string expressions and assignments. There are, in addition to the standard 80 character string, the short (6-character) and medium (12-character) length string data type.

Predefined data is enumerated data associated with a standard parameter. The enumerated data has a range of symbolic values. Enumerated data can be compared for equality. A standard parameter of the predefined data type is assigned symbolic values in the predefined range of integer values that correspond to the symbolic values in accordance with an ordinal numbering scheme (see the ORD function in "Transfer Functions" on page 38).

Note that each predefined type is treated like a data type of its own; it is not considered a variation of the integer data type.

The compound parameter SSTATE is the only enumerated data type supported in HLBL. SSTATE has the following three values:

Symbolic Value	Ordinal Value
INACTIVE	0
ACTIVE	1
EXCEPTION	2

In a Sequence language statement, you can use either the symbolic or the corresponding ordinal value. For example, the expression of an IF statement, looking for the condition where the compound SSTATE is active, can be either of the following:

```
IF (:COMPNAM.SSTATE = ACTIVE)
IF 1 = ORD(:COMPNAM.SSTATE);
```

The sequence compiler and sequence processor recognize the packed Boolean and packed long Boolean data types as valid for assignments and subroutine call arguments. Packed Boolean data types are treated as long integer data types.

Declaration of User-Labeled Parameters

The syntax is as follows:

```
user parm decl =
user_label_name  ':'  user_parm_name |
empty .
user label name =
identifier10 .


user parm =
user_arithm_parm_name |
user_string_parm_name |
user_array_parm_name .
user arithm parm name =
BIOOnn  (where nn = 01 -- 24) |
IIOOOn  (where n  =  1 --  8) |
RIOOnn  (where nn = 01 -- 15) |
BOOOnn  (where nn = 01 -- 16) |
IOOOOn  (where n  =  1 --  5) |
ROOOnn  (where nn = 01 -- 15) .
user string parm name =
SNOOnn  (where nn = 01 -- 10) .
user array parm name =
BAOOOn  (where n  =  1 --  4) |
```

```
IA0001                          |
RA000n  (where n  =  1 --  2) .
```

A user-label cannot be one of the names of the standard parameters, of the block local variables, or of the constants.

Character Set

The basic character set of the Sequence language consists of following letters, numbers, and special symbols:

- Letters A through Z, a through z, and underscore
- Numbers 0 through 9
- Special Symbols `+ - * / = { } ( ) < > . , ; : “`

Vocabulary

Sequence language statements are composed of the following lexical units:

Special Symbols	Keywords
References	Literals
Labels	Comments
Operator Remarks	

You should avoid the use of the following specific lexical units:

CONSTANTS	DISABLE	ENDSUBROUTINE	EXCEPTION
IN	INOUT	SUBROUTINE	to_inactive
to_manual	to_paused	to_sys_err	to_usr_err
VARIABLES	BAD	ON	E
B	I	R	S
S80	S12	S6	

Special Symbols

Special symbols serve as operators and delimiters in Sequence language statements (refer to Table 2-1 below). Some operators and delimiters are created by combining several special symbols.

Table 2-1. Special Symbols

Symbol	Description
`{ ... }`	Encloses comments. Comments are used to document the algorithm.
`(* ... *)`	Encloses operator remarks. Remarks describe the actions performed by statement execution. Remarks can be seen on the default displays.
“...”	Encloses a string data type.
`-->`	Means “to.” Assign the results of a Boolean expression to a Boolean output parameter in a Monitor (MON) block MONITOR CASES statement.
`:=`	Assigns values to parameters.

Table 2-1. Special Symbols (Continued)

Symbol	Description
;	Ends a Sequence language statement.
:	Begins an external reference to a block, compound or block parameter, or shared variable. It also separates compound and block names in a block or block parameter reference.
<<...>>	Indicates a statement label. Labels can be referenced from other statements such as the GOTO statement.
$$...$$	Step label.
(...)	Establishes precedence when evaluating arithmetic and Boolean expressions. It delimits messages sent with the SENDMSG statement. It delimits activation requests in an ACTCASES statement. It delimits TIMER statements.
*	The arithmetic operator for multiplication.
/	The arithmetic operator for real division.
+	The arithmetic operator for addition; it is also a unary operator for identity.
-	The arithmetic operator for subtraction; it is also a unary operator for sign inversion.
=	The relational operator meaning equality.
<>	The relational operator meaning inequality.
<	The relational operator meaning less than.
>	The relational operator meaning greater than.
<=	The relational operator meaning less than or equal to.
>=	The relational operator meaning greater than or equal to.
.	Separates a compound from a parameter in a path or a block from a parameter in a parameter reference.
,	Separates a block reference from case activation requests in the ACTCASES statement. Also, it separates lexical units that make up a message in the SENDMSG statement – string concatenation operator.
/*...*/	Comments in preprocessor directives

Keywords

Keywords are reserved to specify actions carried out by statements when the block is processed. Keywords are always expressed in uppercase characters. Other lexical units, such as literals and identifiers, cannot be given the same name as a keyword. The keywords are:

ABORT	DO	INDEPENDENT_SEQUENCE	NOT	THEN
ACTCASES	DOWNTO	ENDWHILE	ON	TIMER
ACTIVATE	E	EXIT	OR	TO
AFTER	ELSE	EXITLOOP	REPEAT	TRUE
ALREADY	ELSEIF	FALSE	SECURED	UNTIL
AND	ENDFOR	FOR	SENDMSG	WAIT
BAD	ENDIF	GOTO	START_TIMER	WHEN

CASES	ENDMONITOR	IF	STATEMENTS	WHILE
DIV	ENDSEQUENCE	MOD	STOP_TIMER	
DEPENDENT _SEQUENCE	EXCEPTION _SEQUENCE	MONITOR		

Additional keywords include:

ABS	ACTIVE	BIT_PATTERN	BLOCK_EXCEPTION
BLOCK_NAME	BLOCK_STMNO	CALL	COMPOUND_NAME
CONSTANTS	DISABLE	ENDEXCEPTION	ENDSUBROUTINE
EXCEPTION	INACTIVE	MULT_ARRAY	OP_ERR
ORD	RETRY	ROUND	SBXNO
SENDMSG	SENDCONF	SET_ARRAY	SET_SBXS
SQRT	STEPNO	STMNO	STRING
SUBRNO	SUBROUTINE	SUBR_LEVEL	SUM_ARRAY
TRUNC	USER_LABELS	VARIABLES	

References

A Sequence language algorithm can access data by specifying a path to the data in a Sequence language statement. This data is called a reference. Any of the following types of data can serve as a reference in a Sequence language statement:

- Blocks
- Parameter status attributes (ON SECURED BAD)
- Compound parameters
- Block parameters (standard and user-labeled)
- Shared variables
- Timers.

Before referencing user-labeled parameters in the statement section, you must declare the parameters in the heading section. You do not need to declare in the heading section the standard Compound:Block parameters (SSTATE, MA, and so forth).

Internal References

An internal reference is a reference to a standard block parameter, a user-labeled parameter, or one of the status attributes (ON, SECURED, or BAD) associated with a user-labeled parameter, that belongs to the block. In an internal reference, a user-labeled parameter must be referred to by the user-label assigned to the parameter in the heading section.

Examples:

A user-labeled parameter declaration in the heading of block1:

```
ERR_FLG : BI0001;
```

A reference to the user-labeled parameter in the statement section of block1:

```
ERR_FLG := TRUE;
```

A reference to a standard block parameter in the statement section of block1:

```
BI0001 := TRUE;
```

Internal reference formats are described in “Reference Format” on page 30.

External References

An external reference is a reference to a standard parameter, shared variable, timer, or user-labeled parameter that is specified by pathname. (It may or may not belong to the block.)

When you make an external reference to a user-labeled parameter, use the path to the parameter and the parameter name (not the user-label assigned in the heading).

When you use a string type external reference in an expression, precede it with the keyword STRING. HLBL assumes all external references without the keyword STRING are arithmetic data types (real, integer, or Boolean).

You can use relative specification in external references. You can omit the compound name when referring to blocks (and/or their parameters) within the compound. The compound name is prepended to the path name.

Examples:

A reference to a Boolean input parameter in BLOCK1 from the statement section in another block in the same compound:

```
::BLOCK1.BI0001 := TRUE;
```

A reference to a Boolean input parameter in COMPOUND1, BLOCK1 from the statement section in the same compound or in another compound:

```
:COMPOUND1:BLOCK1.BI0001 := TRUE;
```

External reference formats are described in the paragraph below.

Reference Format

The path specified in a reference may contain spaces or comments, such as in the following example: (`::C O MP : BLOCK . PARM:=1;`).

External references are case sensitive. All compound names, block names, and parameters must be entered in uppercase.

Internal Reference Formats:

PARAM
PARAM.STATF

where:

. = Separates parameter and status field names

PARAM = User-labeled parameter name (one to ten characters).

STATF = ON, SECURED, BAD parameter status field.

These status extensions must be used only on the right side of the :=assignment. If used on the left, a syntax error will result.

Examples:

```
FILLFLAG
HEAT_F
LEVEL_F.SECURED
```

External Reference Formats:

:CNAM.CPARAM

:CNAM:BNAM

:CNAM:BNAM.BPARAM

:CNAM:BNAM.TIM

::BNAM

::BNAM.BPARAM

::BNAM.TIM

:SVAR

where:

: = Precedes external references; separates compound and block names.

. = Separates compound and block names from parameter and timer names.

CNAM = Compound name (one to 12 characters) – not required if referenced block is in same compound

CPARAM = Compound parameter name (one to six characters).

BNAM = Block name (one to 12 characters).

BPARAM = Block parameter name (one to six characters).

TIM = Timer name (TIMR1, TIMR2, TIMR3, TIMR4).

SVAR = Shared variable name (one to 12 characters).

— NOTE

::.BPARAM and **::.TIM** are invalid. In both cases, **BNAM** must be specified.
There are only two cases where **BNAM** may be excluded, namely
:CNAM.CPARAM (a compound parameter reference)
:SVAR (a shared variable reference)

Examples:

```
:REACT_FILL.ON
:REACT_FILL:LEVER
:REACT_FILL:DRAIN.MA
:REACT_FILL:DRAIN.BI0001
:REACT_FILL:FILL_TIME.TIMR4
::CLEAN.MA
::CLEAN.RI0004
::CLEAN_TIME.TIMR2
:TANK_LEVEL
:.ON
```

Literals

Literals are values that are interpreted as they appear literally in a statement. A literal can be a real, integer, Boolean, string, or enumerated value.

Examples:

unsigned integer literal:	325
unsigned real literal:	200, 233.2, 5E4, 23E-12, .5E+6
Boolean literal:	TRUE, FALSE
enumerated literal:	Active, Inactive, Exception
string literal:	“Heat Phase is beginning”

— NOTE

1. No spacing or passage to a new line can occur within a literal (except the string literal which can have spaces).
2. Two consecutive double quotation marks ("") in a string literal are interpreted as one double quotation mark that does not terminate the string literal.
3. String literals contain up to 80 characters (not including the enclosing quotation marks), where two consecutive double quotation marks count as one.
4. An enumerated literal only appears as the single right operand of the **=** or **<>** operators in an expression. The left operand must be a parameter of the same pre-defined enumerated type (for example: SSTATE = Active). Expressions are described in "Expressions" on page 34.

Labels

A label identifies a statement or a step. A labeled statement, or step, can be executed by request from another statement, such as a GOTO statement in the same coded programming entity (block, subroutine or SBX).

Labels are only recognized in the block in which they are defined. At run time, they cannot be referred to from sources outside the block.

A label must not be the last line in a subroutine or program. Add a semicolon ";" following the label if this positioning is necessary in the routine.

You can refer to a statement label, or step label, within the block's main code section, only from a statement within that same main code section, or, from a statement in one of the block's SBXs; that is, you can use a label reference to jump out from an SBX into the main code section.

The HLBL compiler generates an error message if the program tries to use a label reference to:

- Jump into a subroutine
- Jump out from a subroutine
- Jump into a standard block exception handler.

Format:

<<X>>

where:

<< >> = special symbol that encloses the label.

x = one to 12 characters (letters, digits, or underscores).

Examples:

```
<<fill_vat2>>
ACTIVATE :REACT_LOGIC:FILL;
.
.
.
GOTO fill_vat2;
```

The **fill_vat2** label is not enclosed in **<< >>** symbols when referred to in a statement. Only the actual label must be so enclosed.

```
<<repeat>>
ACTIVATE :COMP_REACT:DRAIN;
```

This is accepted by the HLBL compiler. If the label uses uppercase lettering, the compiler rejects the label because **REPEAT** in uppercase characters is a keyword.

Comments

Comments, enclosed by the special symbols **{** and **}**, describe algorithm actions; they do not affect the algorithm's flow of control or any data operations. You should use comments to document your algorithm.

Comments can appear anywhere in the algorithm between identifiers, numbers, and special symbols. A comment can occupy more than one line.

Format:

{x}

where:

{ } = Special characters that enclose the comment

x = One or more characters (letters, digits, or underscores); there is no limit to the number of characters.

Example:

```
{Set alarm block used to activate annunciator light.}
```

⚠ CAUTION

The backslash character ("\") must not be added in comments sections of HLBL code, or IACC will throw an error during validation.

Operator Remarks

An operator remark tells the operator what the program is doing while executing. It is not used to tell the operator to perform an action.

Operator remarks describe block algorithm procedures as do comments; however, operator remarks are included in the Operator Representation Module and therefore are accessible to the operator for guidance from displays during operation.

Remarks can appear at statement boundaries only. They can occupy more than one line including carriage returns and the special symbols (* and *) used as delimiters. See Table 3-3 for the maximum buffer size for each operator remark.

For MON blocks, operator remarks must be placed between the case number and the case statement.

Format:

(*x*)

where:

(* *) = Special characters that enclose the remark.

x = characters (letters, digits, carriage returns, or underscores).

Example:

```
(* Fill tank to 50% *)
```

— NOTE

If the next line is a label, you must end the operator remark line with a semicolon.

Expressions

Expressions combine operands and operators according to rules of computation to obtain new values.

There are two kinds of expressions in HLBL: string expressions and arithmetic expressions.

String Expressions

A string expression constructs a string from any mixture of:

- String literals
- Results of arithmetic expressions (including non-string external references)
- Internal references to a string item, optional with a subset specification
- External references of data type string.

The comma (**,**) is the concatenation operator.

The conversion of arithmetic values to ASCII character string expressions are automatically performed as follows:

1. A Boolean value is converted into one of the strings TRUE or FALSE.
2. An integer value is converted into a string of digits, if positive, or into a minus sign followed by a string of digits, if negative.
3. A real value converts into a string of fixed length. The string contains:

   For the mantissa: a sign character (**+** or **-**), one digit, the decimal point, six digits, and the character **e**. For the exponent: a sign character (**+** or **-**), and two digits.

   Examples: +8.800497e+01, -7.423075e-13

A string expression can occupy up to 80 characters. Characters in excess of 80 are ignored without notification; that is, no operational error is raised.

Trailing blanks of the elements that make up the expression are *not* removed. Trailing blanks of the string expression are also **not** removed.

An external reference is interpreted as data type string if, and only if, it is preceded by the keyword STRING. Without that keyword, arithmetic values retrieved from the external reference are converted to ASCII as described above.

Examples of string expressions include:

1. `7 * 10 + 6`
2. `“7”, “6”`
3. `IO0003`

   If `IO0003` = 76, then Examples 1, 2, and 3 all yield: `76`.
4. `STRING :.NAME , “biphenyl_temp =” , RO0002`

   If the compound name = `COMP_64` and `RO0002` = 27.4, then Example 4 yields:

   ```
   COMP_64 biphenyl_temp = +2.740000e+01
   ```
5. `“string” , SN0010 , 3 * i , :COMP:BLK.IO0003`

The result of Example 5 consists of the characters “string”, contents of **SN0010** (without trailing blanks), the ASCII representation of the value of **3*i**, and the ASCII representation of the value stored in COMP:BLK.IO0003.

Arithmetic Expressions

Arithmetic expressions include integer, real, and Boolean expressions.

Integer arithmetic expressions are those in which the operators operate on integer operands only, and the results are integers.

In real arithmetic expressions, the operators operate on all real operands or a combination of integer and real operands. In the latter case the integer value is changed to real before the operator is applied. The results are real.

In Boolean arithmetic expressions, the Boolean result is derived by applying the logical operator **not** to a Boolean value, or by applying the relational operators (=, <>, <=, >=, <, or >) to compatible operands, either arithmetic or string.

Examples of Boolean expressions include:

```
BI0001 := NOT BI0002;
BI0003 := cycle_no = 6;
```

This assigns a value of TRUE (=1) to the parameter, BI0003, when the integer variable, cycle_no, has a value of 6; any other value of cycle_no sets BI0003 to FALSE (=0).

```
BI0004 := cycle_no <> last_cyc;
```

This assigns a value of TRUE (=1) to the parameter, BI0004, when the integer variable, cycle_no, is not equal to the integer variable, last_cyc; if the two variables are equal, BI0004 is set to FALSE (=0).

Boolean expressions (and IF conditional clauses) can also use string comparisons to determine Boolean values.

The following string items can be compared:

- String constants
- Local string scalars (block or subroutine variables, or subroutine arguments)
- Subsets of local string scalars
- Elements of local string arrays (block or subroutine variables, or subroutine arguments).

Any external string parameter to be used in a comparison must appear on the left hand side of the (in)equality sign and must be preceded by the keyword STRING.

String expressions **cannot** be compared with each other because the concatenation operator (**,**) cannot be used.

When entire strings are being compared, trailing blanks of the strings (on either the left or right hand side of the comparison) are stripped away. As a result, the following Boolean expressions all yield the same result:

```
IF STRING :A:B.SN0001 = “COOL    ” THEN ...
IF STRING :A:B.SN0001 = “COOL” THEN ...
IF SN0001 = “COOL    ” THEN ...
IF “COOL” = SN0001 THEN ...
```

When the expression is comparing string subsets, the trailing blanks are not stripped. For instance, the following two statements do not yield the same result.

```
BI0001 := SN0002 ( 1 .. 5 ) = “COOL ”;
```

This assigns a value of TRUE (=1) to the parameter, BI0001, when the first five characters of the string variable are C, O, O, L, and space. If any of the first characters differ, BI0001 is set to FALSE (=0).

```
BI0001 := SN0002 ( 1 .. 4 ) = “COOL ”;
```

In this example, BI0001 can never be true since the subset does not specify enough characters, and since it is a subset that is being compared, the trailing space in COOL is not stripped away.

Other examples of valid string comparisons include:

1. `BO0001 := LoopNamesArr[14] = SN0005 ;`
2. `IF SN0003 ( 61 .. 72 ) = “TempLoop1988” THEN ...`
3. `BI0001 := STRING :COMP4711:BLK0011.SN0010 = “COMP002” ;`
4.
```
SUBROUTINE PQR ( IN arg1 : S80; INOUT arg2 : S12[2] )
VARIABLES
STATEMENTS
II0001 := 5 ;
BI0001 := (arg1[ II0001 .. 9 ] = arg2[1] );
ENDSUBROUTINE
.
.
.
strarr[1] := “FC101”
CALL PQR ( “ FC101”, strarr ); {this sets BI0001 True}
```

Operands

An operand is an internal or external reference that is evaluated according to the operators in an expression. The following references can be used as operands: a compound parameter, block parameter, block status attribute (ON, SECURED, or BAD), shared variable, or literal. References are described in more detail in “References” on page 29. Reference formats are described in “Reference Format” on page 30.

When evaluating an expression, if the results of addition (adding operators: + and -) or multiplication (multiplying operators: *, /, MOD, and DIV) cause an overflow, underflow or divide-by-zero condition, the appropriate minimum or maximum value is substituted for the result. During translation the HLBL compiler detects division by zero in constant expressions.

Operators

Operators combine operands to obtain a single value. The data type of operands and operators must be consistent. The table below lists each operator and its appropriate operand data types.

Operator	Operation	Type of Operand	Type of Result
Unary			
+	Identity	Integer or real	Same as operand
-	Sign inversion	Integer or real	Same as operand
NOT	Logical negation	Boolean	Boolean

Operator	Operation	Type of Operand	Type of Result
Arithmetic Dyadic			
+	Addition	Integer or real	Integer/Real
-	Subtraction	Integer or real	Integer/Real
*	Multiplication	Integer or real	Integer/Real
/	Real division	Integer or real	Real
DIV	Integer division	Integer	Integer
MOD	Modulus	Integer	Integer
Relational			
=	Equality	Both boolean or both compatible	Boolean
<>	Inequality	Both boolean or both compatible	Boolean
<=	Less than or equal	Compatible	Boolean
>=	Greater than or equal	Compatible	Boolean
<	Less than	Compatible	Boolean
>	Greater than	Compatible	Boolean
Logical Dyadic			
AND	Logical and	Boolean	Boolean
OR	Logical or	Boolean	Boolean
String Dyadic			
,	Concatenation	B, I, R, S, S12, S6	String

Where: Integer/Real = Integer if both operands are integers; real otherwise.

Compatible means the operands must be of the same data type or one operand must be integer and the other must be real. In the latter case, the integer value is changed to real before the operator is applied.

The **MOD** operator divides the first integer operand by the second integer operand and returns only the remainder.

The **DIV** operator divides the first integer operand by the second integer operand and drops any remainder.

For example, consider the following code:

RO0001 := II0001 / II0002 ;

RO0002 := II0001 DIV II0002 ;

RO0003 := II0001 MOD II0002 ;

If II0001 = 27 and II0002 = 3

RO0001 = 27/3 = 9

RO0002 = 27 DIV 3 = 9

RO0003 = 27 MOD 3 = 0

If II0001 = 27 and II0002 = 4

RO0001 = 27/4 = 6.75

RO0002 = 27 DIV 4 = 6

RO0003 = 27 MOD 4 = 3

If II0001 = 27 and II0002 = 5

RO0001 = 27/5 = 5.4

RO0002 = 27 DIV 5 = 5

RO0003 = 27 MOD 5 = 2

Functions

The Sequence language provides a set of functions that include transfer functions, arithmetic functions, and array functions shown in the following table.

Transfer Functions	Arithmetic Functions	Array Functions
TRUNC	ABS	SET_ARRAY
ROUND	SQRT	SUM_ARRAY
ORD		MULT_ARRAY

Transfer functions convert values of one data type into another data type. They have one parameter and deliver one result. A transfer function can be a simple expression by itself or an operand in a more complex expression. HLBL transfer functions include TRUNC, ROUND, and ORD.

Arithmetic functions perform an algebraic operation on an integer or real value. An arithmetic function can be a simple expression by itself or an operand in a more complex expression. HLBL arithmetic functions include ABS and SQRT.

Array functions allow you to manipulate all elements of an array with a single statement. An array function can only appear in the right hand side of an assignment statement. HLBL array functions include SET_ARRAY, SUM_ARRAY, and MULT_ARRAY.

Transfer Functions

Transfer functions convert values of one data type into another data type. They have one parameter and deliver one result. A transfer function can be a simple expression by itself or an operand in a more complex expression. HLBL transfer functions include TRUNC, ROUND, and ORD.

TRUNC

The TRUNC(r) function changes a real value into an integer value by truncating any places to the right of the decimal point.

Examples:

```
TRUNC(1.83 + 1.56) = 3
TRUNC(-3.7) = -3
```

The value r must be an integer or a real value. If r is an integer, TRUNC(r) yields r.

If r is not within the range -3.1E10 to 3.1E10 (floating-point notation), TRUNC(r) assumes the minimum value 80000001 hexadecimal (-231) or the maximum value 7FFFFFFF hexadecimal (+231-1).

ROUND

The ROUND(r) function changes a real data value to an integer data value by rounding to the nearest integer. Specifically, if r Š 0 then ROUND(r) = TRUNC(r + 0.5), or if r < 0 then ROUND(r) = TRUNC(r - 0.5), where TRUNC truncates any places to the right of the decimal point.

Examples:

```
ROUND(3.2 + 1.3) = 5
ROUND(-4.5) = -5
```

The value r must be an integer or real. If r is an integer, then ROUND(r) yields r.

If r is not within the range -3.1E10 to 3.1E10 (floating-point notation), ROUND(r) assumes the minimum value 80000001 hexadecimal (-231) or the maximum value 7FFFFFFF hexadecimal (+231 - 1).

ORD

The ORD(v) function converts any enumerated type value into an integer value. The integer value reflects the ordinal number in a list of symbolic values that defines the enumerated type. The value **v** must yield a Boolean, integer, or enumerated type value.

If **v** is boolean, then **ORD**(false) = 0 and **ORD**(true) = 1.

If **v** is an integer, then **ORD**(v) yields the value of **v**.

If **v** is enumerated, then **ORD**(v) yields the integer ordinal number in the range of values for that enumerated type. See “Data Types” on page 25, for the enumerated data types supported in HLBL.

In a Sequence language statement, either the symbolic value or the corresponding ordinal value can be used. For example, the expression of an IF statement looking for the condition where the compound SSTATE is active can be either of the following:

```
IF (:COMPNAM.SSTATE = ACTIVE) THEN ...
IF (1 = ORD (:COMPNAM.SSTATE)) THEN ...
```

The restrictions to the data types of the values r and v are enforced during compilation when the data type is known. If r and v contain external references, data type checking is done at run time and a violation yields an operational error.

The Bad Status attribute associated with the operand of a transfer function is propagated to the result.

Arithmetic Functions

Arithmetic functions perform an algebraic operation on an integer or real value. An arithmetic function can be a simple expression by itself or an operand in a more complex expression. HLBL arithmetic functions include ABS and SQRT.

ABS

The ABS(v) function yields the absolute value of the real or integer value v. The data type of the result is that of v.

Example:

```
ABS(-26.37) = +26.37
```

SQRT

The SQRT(v) computes the real, square root of the integer or real value v. The data type of the result is real.

Example:

```
SQRT(4) = 2.0
```

The HLBL compiler generates an error message when the operand, v, of the SQRT function assumes a value that, during compilation, is known to be negative. If v assumes a negative value at run time, the function yields the square root of the absolute value without any warning or error message.

The Bad Status attribute associated with the operand of an arithmetic function is propagated to the result.

Array Functions

Array functions allow you to manipulate all elements of an array with a single statement. HLBL array functions include SET_ARRAY, SUM_ARRAY, and MULT_ARRAY.

An array function can only appear in the right-hand side of an assignment statement. Only one array function can appear in a statement, that is, an array function cannot be a factor within a more complicated expression.

The left-hand side of the statement is an internal reference to:

- An array item
- A user-array parameter
- A local-array variable
- An array IN argument
- An array INOUT argument.

An array function cannot manipulate the elements of an external array.

SET_ARRAY

The SET_ARRAY(expr) function assigns the value of the expression to all elements of an array. The data type of the expression, and of the destination array, can be either string or arithmetic.

Examples:

```
RA0001 := SET_ARRAY (0) ;
BA0001 := SET_ARRAY (II0008 < 3 OR BI0024) ;
STR_A2 := SET_ARRAY (“Initialized String ”) ;
```

SUM_ARRAY

The **SUM_ARRAY**(expr) function adds the value of the expression to all elements of an array. The data type of the expression, and of the destination array, must be arithmetic. The HLBL compiler generates an error if the destination array is data type string.

Example:

```
RA0002 := SUM_ARRAY ( -130.5 * Min_Level ) ;
```

MULT_ARRAY

The MULT_ARRAY(expr) function multiplies all elements of an array by the value of the expression. The data type of the expression, and of the destination array, must be arithmetic. The HLBL compiler generates an error if the destination array is data type string.

Example:

```
IA0002 := MULT_ARRAY ( 1 / AcidTempsArr [5] )
```

MULT DIMENSIONAL ARRAY MANIPULATION

Sequence logic can manipulate a two dimensional array as follows.

In the VARIABLES section, an array is defined as follows:

```
array_name : I[i, j];
```

where `: I` is the type

`i, j` are the number of rows and columns respectively

In the STATEMENTS section, a specific row x and column y can be assigned a value z as follows:

```
array_name[xx,y] := z;
```

In this example, the STATEMENTS section can include a specific row x and column y to be monitored by a writeable integer input on the Sequence block's detail display as follows:

```
II0001 := array[x,y] ;
```

TRANSFERRING ARRAYS

The following external reference will transfer the contents of the local array "xarray" to another block's RA0001 array:

```
:CMPD:BLK.RA0001 := xarray;
```

Operator Order of Precedence

To avoid ambiguity when evaluating an expression, operators are evaluated in the following order:

1. Unary operators (`+`, `-`, `NOT`) and functions
2. Multiplying Operators (`*`, `/`, `DIV`, `MOD`)
3. Arithmetic Operators (`+`, `-`)
4. Relational Operators (`=`, `<>`, `<=`, `>=`, `<`, `>`)
5. Logical Operators (`AND`, `OR`).

Operators in parentheses are evaluated prior to other operators regardless of the order of precedence.

Data Type Checking

The data type of operands and operators in an expression must be consistent. Data consistency is checked either during compilation or run time, depending on the type of operand.

If an operand contains an internal reference (parameter or parameter status attribute that belongs to the block), its data type is known by the sequence block compiler and it can be checked for consistency during compilation.

If an operand contains an external reference (a parameter, shared variable, or timer that belongs to another block or application), its data type is not known by the compiler; therefore, consistency cannot be checked until run time.

When data type consistency is checked at run time, inconsistencies are handled as follows:

- If an integer value appears where a real value is expected, the integer value is changed to real.
- If any other type inconsistency occurs, it causes a run time operational error. Refer to Appendix A "Sequence Control Error Messages" in this document for lists of operational errors.

Reference Connection Checking

References used in Sequence language statements have connection properties associated with them. A reference can be connectable, settable, both, or neither.

Any parameter linked to another parameter or a shared variable, either to receive data or to send data, must be a connectable parameter. If the parameter is to send data to the linked parameter or shared variable, it can be either a connectable output or a connectable input parameter. If it is to receive data from the linked parameter or shared variable, it must be a connectable input parameter.

Some Sequence language statements, such as ABORT, ACTCASES, ASSIGNMENT, ACTIVATE, START_TIMER, and STOP_TIMER set the value of a reference (block, parameter, timer, or shared variable). When using one of these statements, the reference must be settable.

Internal references are settable with HLBL statements and are checked during compilation.

The settability of external references cannot be checked until run time.

In sequence blocks, as in all other blocks, output parameters are secured when a block is in Auto. Input parameters are secured when they act as a data-sink in a linkage or connection.

Operational Errors

An operational error occurs at run time when a sequential control block encounters an error that it cannot recover from by itself. Operational error codes are stored in the block's OP_ERR parameter and can be displayed during operation. Here are some conditions that can cause operational errors:

- Data type inconsistencies in an external reference are discovered.
- A statement tries to set the value of a secured parameter. For example, an ACTCASES statement cannot address a MON block if the block's ACTPAT parameter is secured.
- An external reference is defined to a compound, block, or parameter that is not defined in uppercase.
- A statement references a parameter or block that does not exist. For example, a START_TIMER statement cannot address a TIM block that does not exist.
- A statement cannot address a parameter or block due to communication errors. For example, the ABORT statement fails to deactivate a block.

SBX1, for errors (OP_ERR = 2000 ... 3000), enables sequence blocks to suppress error messages and execute error handling logic. This logic can either re-execute the erroneous statement, or move to the statement following the erroneous statement. You can also define SBX1 to count errors and/or to output a user-specific message. SBX2 provides the same capability for system errors (all errors outside the 2000 to 3000 range).

Bad Status Attribute

If a communications error occurs, the parameter's Bad status bit is set to indicate that the value is invalid. The Bad status bit can be set because a point, channel, or Fieldbus Module fails, or because there is a mismatch between an Equipment Control Block and a Fieldbus Module.

When a Sequence language expression is evaluated, if any operand (internal or external reference) has the Bad status attribute set the result of the expression has the Bad status attribute set.

However, if the Bad status attribute is set as a result of evaluating an expression in a FOR, IF, REPEAT, WAIT, or WHILE statement, it is ignored and the value is interpreted as if it were not set.

In a Monitor (MON) block, if the Bad status attribute is set as a result of evaluating an expression in the Monitor Case statement, the expression value is interpreted as false. The value of Bad status attribute is sent to the Boolean output parameter associated with the expression.

3. HLBL Preprocessor and Complier

The HLBL preprocessor is a macro processor that transforms the source text before checking and also before compilation.

Preprocessor Commands

Preprocessor commands are lines of source code which start with "#".

The "#" is followed by an identifier which is the command name. For example, "#define" is the command which defines a macro.

Space characters are allowed before and after the "#".

There is a fixed set of valid preprocessor command names (see Table 3-1). You cannot define new preprocessor commands.

Some of the command names require arguments. These make up the rest of the command line and must be separated from the command name by space character.

For example, "#define" must be followed by a macro name and the intended expansion of the macro.

A preprocessor command cannot be more than one line but may be split with the backslash "\" character to force a new line for the sake of readability without changing the semantics of the command.

Table 3-1 lists the commands that the preprocessor supports.

Table 3-1. Preprocessor Commands

Command	Description
#define	Define a preprocessor macro.
#undef	Remove or cancel a preprocessor macro definition.
#include	Insert text from another source file.
#if	Conditionally include some text, based on the value of a constant expression.
#ifdef	Conditionally include some text, based on whether a macro name is defined.
#ifndef	Conditionally include some text, with the sense of the test opposite that of #ifdef.
#else	Alternatively include some text, if the previous #if, #ifdef, or #ifndef, test failed.
#endif	Terminate conditional text.

The /* and */ comment markers can also be used. Note that this comment is stripped by the preprocessor and, therefore, it does **not** appear in the error and listing file nor in the operator representation module.

— NOTE

The { and } are invalid to enclose comments for preprocessor commands. The /* and */ must be used for preprocessor commands.

All preprocessor commands are processed before the actual compilation. As a result those directives do **not** appear in the error and listing file nor in the operator representation module.

Macros

Macros are a sort of abbreviation used for the substitution of code parts. They are definable with or without arguments.

Simple Macro Definitions

A simple macro definition is a kind of abbreviation. It is a name that stands for a fragment of code, which is then expanded by the preprocessor wherever the macro definition appears in the code.

The macro must be defined with the “#define” command. It is followed by the name of the macro and then the code it should be an abbreviation for.

For example,

```
#define BUFFER_SIZE 1000
```

defines a macro named “BUFFER_SIZE” as an abbreviation for the text “1000”.

The following HLBL code referring to the “BUFFER_SIZE” macro,

```
my_size := BUFFER_SIZE * 2;
```

is expanded by the preprocessor to

```
my_size := 1000 * 2;
```

Any macro definition must be a single line; however it can be split up to more than one line by the backslash “\” character as described in “Preprocessor Commands” on page 45.

You can place (even multiple line) comments within a macro without any impact on the macro.

The syntax of the “#define” does not require an assignment operator or any special delimiter token like the “;”. You need not balance parentheses, but a quoted (delimited by “'”) string is extended to include the matching quote character.

Although the body need not resemble valid code, it should provide valid code for the source code context it is used in. If not, further compilation results in errors.

The macro body can contain calls to other macros.

Example:

```
#define BUFSIZE            1000
#define TABLESIZE          BUFSIZE
```

The name “TABLESIZE” then when used in source code would go through two stages of expansion, at last resulting in “1000”.

The command name cannot derive from a macro expansion.

Thus the following example does not work:

```
#define               my_define  define
#my_define            FLAG       1     (* error *)
```

Also the preprocessor does not recognize macros which are defined within another macro like intended in the following example:

```
#define DO_MY_DEFINITION #define FLAG 2
:
:
DO_MY_DEFINITION;   (* error *)
```

Macros with Arguments

Arguments are fragments of code which must be supplied each time the macro is used. These fragments are then included in the expansion of the macro according to the instructions of the macro definition.

Definition Rules

1. A macro with arguments is defined with the "#define" command followed by a list of argument names in parentheses.
2. The arguments can be any valid identifiers, separated by commas and optional space characters.
3. The open-parenthesis "(" must immediately follow the macro name, with no space in between as shown in the example to calculate the distance between two points.

   Example:

   ```
   #define  DIST(X1, Y1, X2, Y2)  (SQRT ((X1-X2)*(X1-X2)+ \
                                        (Y1-Y2)*(Y1-Y2)))
   ```

4. To use the macro that expects arguments, the name of the macro is written followed by a list of actual arguments in parentheses, separated by commas. The number of actual arguments must match the number of arguments the macro expects.

   Example:

   ```
   dist := DIST (10.1, 12.5, 5.3, 3.4);
   ```

   or

   ```
   dist := DIST (x + 10.0, y + 3.0, 40.0, 10.0);
   ```

5. The expansion text of the macro depends on the used arguments. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument.

   Example:

   ```
   dist := DIST (10.1, 12.5, 5.3, 3.4);
   ```

   expands to

   ```
   SQRT ((10.1-5.3)*(10.1-5.3)+(12.5-3.4)*(12.5-3.4)))
   ```

   and

   ```
   dist := DIST (x + 10.0, y + 3.0, 40.0, 10.0);
   ```

   expands to

   ```
   SQRT((x+10.0-40.0)*(x+10.0-40.0)+(y+3.0-10.0)*(y+3.0-10.0)))
   ```

6. IACC expands Structured Text (ST) templates.

Argument Rules

1. Parentheses in the actual arguments must balance.
2. A comma within parentheses does not end an argument.
3. Brackets as used in arrays need not be balanced and thus do not prevent a comma from separating arguments.

   Example:

   ```
   macro (array[x := y, x + 1])
   ```

   The arguments separated by the comma passed to the macro are:

   ```
   “array[x := y” and “x + 1]”.
   ```

   To supply an appropriate argument the following syntax with parentheses must be used:

   ```
   macro (array[(x := y, x + 1)])
   ```

4. The actual arguments of a macro can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros.

   To supply an empty argument to macro “MAC(arg1)” it must be called with at least one space character between the parentheses, like this: “MAC( )”.

   Calling “MAC()” is providing no arguments, which is an error as MAC expects an argument.

   But if a macro is defined like 'MAC0()' the correct way to call this macro to take zero arguments is 'MAC0()'.

5. Using the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), causes the preprocessor to leave the source code as is.

Undefining Macros

To undefine a macro means to cancel its definition. This is done with the “#undef” command. “#undef” is followed by the macro name to be undefined.

Example:

```
#define MAC       4

x := MAC;

#undef MAC

x := MAC;
```

expands into

```
x := 4;

x := MAC;        (* error if MAC isn’t also a variable *)
```

The same form of "#undef" command will cancel definitions with arguments or definitions that do not expect arguments.

The "#undef" command has no effect when used on a name not currently defined as a macro.

Redefining Macros

Redefining a macro means defining (with "#define") a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one.

That can happen automatically when a source file containing macro definitions is included more than once (see "Include Files" on page 49).

Nonidentical redefinition provokes a warning message from the preprocessor. As it is sometimes useful to change the definition of a macro in mid-compilation, a warning can be inhibited by undefining the macro with "#undef" before the second definition.

For a redefinition to be identical, the new definition must exactly match the one already in effect, with two possible exceptions:

- Space character can be added or deleted at the beginning or the end.
- Space character can be changed in the middle (but not inside strings). However, it can not be eliminated entirely, and it can not be added where there was no space character.

Comments do not have any impact on redefinition.

Include Files

An include file is a file containing declarations, macro definitions (see "Macros" on page 46) or common source code to be shared between several source files.

The file inclusion statement is a directive to the HLBL compiler to include the characters of the named file into the source file of the block that is being translated. To request the use of an include file the preprocessor command "#include" is used together with a sequence of printable characters.

The include statement can appear anywhere in an HLBL source. The characters in the include file need not be a correct compilation unit. The source file, resulting from the combination of the include file(s) and the source file, is checked by the HLBL compiler for syntax and semantic errors.

The requested filename is identified through the configurator's database.

Syntax:

```
"#include" ['"']{<any printable character>}['"']
```

Examples:

Table 3-2. Include File Examples

`#include "default_types.dec"`	May include a file containing default data type declarations
`#include MY_DEFAULT_TYPES`	Includes the file represented by the macro MY_DEFAULT_TYPES. The expanded macro is then enclosed by "double quote" characters.

Uses of Include Files

Include files serve different kinds of purposes:

- Provide common declarations and macro definitions to be used in the actual source code. Therefore, the related declarations and definitions appear only in one place. Any modifications to those declarations and definitions can be done only in one place, and the code parts which include those files will automatically use the modified version when next recompiled.
- Provide specific source code to fit right into the place where they are being used within the actual source code.

#include Command

The "#include" command has two variants:

- "#include" "filename"

  The requested filename is identified within the configurator's database. If it does not represent an identifiable filename, the preprocessor generates a warning.

- "#include" macro_text

  macro_text is checked for being a valid macro call and expanded to the above version of "#include" command. This allows you to define a macro that controls the file name to be used at a later point in the program.

The following directory is used to search for the file:

- /usr/fox/ciocfg/sequeninclude (for ICC - VENIX)
- /opt/fox/ciocfg/sequeninclude (for ICC - Solaris)
- IACC puts them in the library for text objects.

The "#include" command works by directing the preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that generates from the text after the "#include" command.

Included files are not limited to declarations and macro definitions; those are merely the typical uses. You can include any fragment of a code program. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the containing file.

However, a comment or a string may not start in the included file and finish in the including file. An unterminated comment or string constant in an included file is considered to end (with an error message) at the end of the file.

The line following the "#include" command is always treated as a separate line by the preprocessor even if the included file lacks a final new line.

Example of a sequence source file using preprocessor directives:

```
INDEPENDENT_SEQUENCE
#define TESTPHASE                    2
#define ON                           TRUE
#define OFF                          FALSE
#define Switch_Motor2(s)           :COMP_12:MTR_101.MA:=s
CONSTANTS
#include "constant.defs.2"
```

```
VARIABLES
#include “vars_2.decl.org”
USER LABELS
Nylon_Temp : RI0001 ;
#include “nylon2.subrs”
#if TESTPHASE == 2
#include “dral.sbxs.tst”              { test version }
#else
#include “dral.sbxs.fin               { final version }
#endif
STATEMENTS
#include “polym.main”               {main polymerization logic }
#ifdef TESTPHASE                        { not in final version }
SENDMSG (“Executing Motor switch logic”) TO MSGGR2 ;
#endif
Switch_Motor2 ( ON );                   { <- parameterized macro }
WAIT 30;
Switch_Motor2 ( OFF );
ENDSEQUENCE
```

Once-Only Include Files

“#include” files can include other “#include” files. The nesting level is only limited by the resources as determined by the environment. However, this can easily result in certain includes being done more than once, which may lead to errors.

A standard way to prevent this is to enclose the entire real contents of the “#include” file in a conditional macro, like this:

```
#ifndef __MY_INC_FILE__
#define __MY_INC_FILE__
```

The entire “`#include`” file

```
#endif /* __MY_INC_FILE__ */
```

The macro __MY_INC_FILE__ indicates that the file has been included once already.

Its name should begin with “__” to avoid any conflicts with other names used and should include the name of the file with additional text.

Conditional Commands

Conditional commands allow parts of the source code to be included or ignored during compilation. The condition can be tested based on the value of a constant expression or on whether a macro name is defined.

#if Command

The “#if” command in its simplest form consists of:

#if “expression”

controlled text

```
#endif
```

The “expression” part is an integer-type expression that is restricted to containing:

- Integer constants, which are all regarded as long or unsigned long.
- Character constants, which are interpreted according to the character set.
- Arithmetic operators for addition, subtraction, multiplication, division, “AND”, “OR” and comparisons.
- Identifiers which are not macros. (These are all treated as zero).
- Macro calls. All macro calls in the expression are expanded before any actual computation of the expression's value.

The controlled source text inside of a conditional command can again include preprocessor commands. The text can also contain other conditional groups.

The “#if” and “#endif” commands must balance.

#else Command

The “#else” command can be added to a conditional command to provide alternative text to be used if the condition is false.

Example:

```
#if expression
text-if-true
#else
text-if-false
#endif
```

If expression is nonzero, and thus the text-if-true is active, then “#else” acts like a failing conditional and the text-if-false is ignored and vice versa.

#ifdef and #ifndef Commands

These are conditional commands to test if macro names are defined or not.

Example:

```
#ifdef  macro name
text-if-defined
#ifndef macro name
text-if-not-defined
#endif
```

Sequence Compiler Limits

The maximum size of a sequence block allowed by CP30, CP40, CP60, CP270, and FCP280 and equivalent platforms is 32000 bytes (including block header, parameters, and interpretive code).

You can configure the size limit of a sequence block to a maximum of 32000 bytes (including block header, parameters, and interpretive code) solely for the purpose of ensuring contiguous memory allocation for sequence blocks in the CP (not for changing the limits imposed by the compiler).

Block parameter CSIZE gives the current block size. Parameter CSPACE allows you to configure the space to be reserved.

Table 3-3 lists the sequence compiler limits.

Table 3-3. Sequence Compiler Limits

Limit	Description
32000	Maximum size (in bytes) of a sequence block including header, parameters, and interpretive code
30000	Maximum size (in bytes) of a sequence block in the CP30, CP40, or CP60 including, parameters, and interpretive code
31200	Maximum size (in bytes) of a sequence block in a CP270 or FCP280 including, parameters, and interpretive code
16[1]	Maximum length of a constant identifier
16[1]	Maximum length of a variable name
16[1]	Maximum length of an argument name
16[1]	Maximum length of a user-parameter label
10	Maximum length of a subroutine name
16[1]	Maximum length of a step label (for example, $$step$$) excluding the enclosing $$ and $$
16[1]	Maximum length of a statement label (for example, **<<stmt>>**) excluding the enclosing **<<** and **>>**
128	Maximum number of block local variables in the main code
128	Maximum length of a line of source code
50	Maximum combined number of subroutine local variables and subroutine arguments in each subroutine

Table 3-3. Sequence Compiler Limits (Continued)

Limit	Description
20	Maximum number of comma separated arguments before a data type specification
256	Maximum number of dimensions in an array-type block local variable or subroutine local variable
8	Maximum levels of nested array index expressions (for example, a[1,b[6,c[8,2]]] has three levels)
10	Maximum number of nested subroutine CALLs (Note that a subroutine cannot call itself, directly or indirectly.)
255	Maximum number of subroutines in each dependent, independent, or exception block
255	Maximum number of external references a) in each subroutine or b) in the main code and all SBXs (standard block exception handlers)
255	Maximum number of message strings a) in each subroutine or b) in the main code and all SBXs
255	Maximum number of steps in the main code including INIT_STEP and END_STEP
255	Maximum combined number of statement labels and step labels a) in each subroutine or b) in the main code and all SBXs (Note that only the main code can have step labels and INIT_STEP and END_STEP are counted as two.)
20	Maximum levels of nested control constructs (for example if, for, repeat, while, . . .)
4096	Maximum buffer size for all subroutine names in each dependent, independent, or exception block
4096	Maximum buffer size for all external references a) in each subroutine or b) in the main code and all SBKs
4096	Maximum buffer size for all message strings a) in each subroutine or b) in the main code and all SBKs
1025	Maximum buffer size for each operator remark

[1.] These strings may exceed 16 characters. However, their first 16 characters MUST be unique, as only these 16 characters are used (valid).

4. HLBL Statements

This chapter contains information on HLBL statements including statement syntax and the statements ABORT, ACTCASES, ACTIVATE, ASSIGNMENT, SCALAR, ASSIGNMENT, ARRAY, BIT_PATTERN, EXIT, EXITLOOP, FOR, GOTO, IF, MONITOR CASE, REPEAT, RETRY, SENDCONF, SENDMSG, SET_SBXS, START_TIMER, STOP_TIMER, WAIT, and WHILE.

Sequence Block Statements

Sequence block statements can be divided into two groups: statements that alter the flow of control, and statements that perform operations on data.

The logic flow control statements are:

- EXIT
- EXITLOOP
- FOR
- GOTO
- IF
- MONITOR_CASE
- REPEAT
- RETRY
- WAIT
- WHILE.

The execution of statements causes actions to be performed. The elements of a statement can be distributed over any number of lines of text.

Statements can be prefixed by a label. There are two types of labels:

statement_label – a name that serves to identify a single statement

step_label – a label that identifies a group of statements

Both statement and step labels can be referenced by GOTO statements or by GOTO clauses in statements such as WAITUNTIL, SENDCONF, or ACTIVATE. A GOTO statement (or GOTO clause) can redirect the flow of control to a specific statement by naming the label that identifies the target statement.

Statements that are put in sequential textual order are executed in sequential time order at run time.

Statement Syntax

The executable HLBL algorithm code in the IND, DEP, EXC, and MON blocks consists of statements and statement_sequences defined below. The code enclosed by square brackets **[ ]** is optional.

Executable algorithm syntax:

```
statement_sequence [ ; statement_sequence ; .... ]
algorithm_terminator
```

Statement_sequence syntax:

```
labelled_statement [ ; labelled_statement ; .... ]
```

Labelled_statement syntax:

```
[ stmnt_label_def ] [ opr_remark ] statement
```

or

```
[ step_label_def ] [ opr_remark ] statement
```

Statement syntax:

```
assignment_statement
```

or

```
logic_flow_control_statement
```

or

```
procedural_statement
```

or

```
subr_call_statement
```

or

```
empty
```

Algorithm_terminator syntax:

```
ENDSEQUENCE
```

or

```
ENDSUBROUTINE
```

or

```
ENDEXCEPTION
```

A statement can also be empty.

The Monitor (MON) block has only one type of statement, the Monitor Cases statement. It is used to define up to 16 cases that can be monitored by the block.

Following the information on statement syntax and format conventions, statements are listed in alphabetical order.

The semicolon separates statements in an algorithm. Additional algorithm terminators are: ENDFOR, ENDWHILE, ENDIF, ENDSEQUENCE, ELSE, ELSEIF, and UNTIL.

When constructing a statement, adjacent lexical units must be separated by spaces, tabs, or lines.

Except for comments, operator remarks, and string literals, lexical units must not contain spaces or tabs.

Lexical units, except for comments and operator remarks, must fit on a single line. Otherwise, spacing is optional and used to make statements legible.

Statement Format Conventions

In the following alphabetized list of statements, the statement format is described using these conventions:

[ ]	Square brackets enclose optional lexical units.
...	Ellipses indicate that a lexical unit or series of lexical units can be repeated.
UPPERCASE	Indicates keywords that must be typed exactly as they appear.
lowercase	Indicates lexical units that are variable.

Abort

Format

```
ABORT :compoundname:blockname
```

Description

The ABORT statement deactivates an active Sequence (IND, DEP, or EXC) or Monitor (MON) block by writing a false value to the ACTIVE parameter of the addressed block. If another request is pending, the ABORT statement is retried once every scheduled basic processing cycle (BPC) until no other request is pending.

An attempt to deactivate a block that is already deactivated is treated as a successful attempt.

An unsuccessful attempt to deactivate the block is treated as an operational error.

Example:

```
ABORT :REACT_LEVEL:FILL
```

Actcases

Format

```
ACTCASES (compoundname:blockname,“activate pattern”)
```

Description

The ACTCASES statement can set the activity status of individual cases in the Monitor Case statement of the specified Monitor (MON) block.

The activation pattern represents the new activity states of the cases belonging to the MON block specified in the statement.

Each character in the activation pattern corresponds to a case in that block. The characters allowed in the pattern are A, I, and -. The character A stands for Active, I for Inactive, and - indicates no change in activity state.

The first character stands for the first case in the Monitor Case statement, the second character for the second case, and so on. Any cases not specified in the pattern are left as they are.

Examples:

```
ACTCASES (:REACT_LEVEL:FILL,“AAAAAIAIAIAIAAAA”)
ACTCASES (:REACT_LEVEL:DRAIN,“A-AIIIA-I---A”)
```

Activate

Format

```
ACTIVATE :compoundname:blockname [WHEN ALREADY GOTO label]
```

Description

The activation statement activates a Sequence (IND, DEP, or EXC) or Monitor (MON) block by writing a true value to the ACTIVE parameter of the addressed block. If another request is pending, the ACTIVATE statement is retried once every scheduled BPC until no other request is pending.

An attempt to activate a block that is already active is treated as a successful attempt. An optional WHEN ALREADY GOTO clause allows you to branch to a statement specified by a label when the block is already active. The label can be either a statement label or a step label.

An unsuccessful attempt to activate the block is treated as an operational error.

Examples:

```
ACTIVATE :REACT_LOGIC:FILL;
ACTIVATE :REACT_LOGIC:LEVEL WHEN ALREADY GOTO drain;
.
.
.
<<drain>>
ACTIVATE :REACT_LOGIC:DRAIN
```

Assignment, Scalar

Format 1

```
internal reference := expression
```

Format 2

```
external reference := expression
```

Description

The assignment statement replaces the current value of an internal or external reference (on the left hand side) with the value that results from evaluating an expression (on the right hand side). The value of the expression’s Bad status bit is transferred to the reference (refer to “Bad Status Attribute” on page 43).

The two categories of scalar assignments are the arithmetic assignment statement and the string assignment statement. The data type of the left hand side of the assignment statement determines the category of any given statement.

If the left hand side is a reference to a string type internal reference, the assignment is a string assignment. In all other cases involving internal references the assignment is arithmetic.

If the left hand side is an external reference, the right hand side, if it is an internal reference, determines the assignment category.

If both left and right sides are external references, the assignment is arithmetic unless the keyword STRING precedes the external reference in the right hand side. See the following table.

WHERE AE = Arithmetic Expression CE = Compiler Error Message SE = String Expression LEFT HAND SIDE OF ASSIGNMENT	RIGHT HAND SIDE OF ASSIGNMENT Non-String Internal Reference	 String-Type Internal Reference	 External Reference	 'STRING' External Reference	
Non-String Internal Reference	AE	CE	AE	CE	External References are specified by a FPN_construct. FPN = Full Pathname
String-Type Internal Reference	SE	SE	SE	SE	
External Reference	AE	SE	AE	SE	

Scalar String Assignment

Example:

```
SN0007 := "Hello"
```

Scalar Arithmetic Assignment

In an arithmetic assignment, the left hand value can be:

- An external reference
- A user parameter (labeled or not)
- A local block variable
- A local subroutine variable
- A formal subroutine argument.

External references can be any compound or block parameter or shared variable including standard parameters of the block that is defined. An operational error occurs if the assignment violates the access restrictions of the external reference.

Internal references are governed by the following rules:

- The left value can be a user Output and Input parameter.
- If the left value is an Input parameter that is secured, the value is not assigned and the block raises an operational error 2317.

- Standard parameters of the Sequence block cannot be the left value when addressed using an internal reference.
- Connection status fields (SECURED, ON, and BAD) can only be read in HLBL statements, that is, they can only appear on the right hand side of an assignment. They cannot be set to some value by means of an assignment statement.

The data types of the reference and the value yielded by evaluating the expression must be compatible as follows:

- If the right value is integer, the left value must be either real or integer and the right value is coerced appropriately.
- If the right value is real, the left value must be real. The expression can be programmed to deliver an integer using the TRUNC or the ROUND function.
- If the right value is Boolean, the left value must be Boolean.

When the data type is not known at compilation time, a check is performed at run time. At that time, if the right value appears to be a real and the left value is an integer, the right value is coerced automatically as if the ROUND function were specified. Any other data-type mismatch at run time raises an operational error.

Any failure, not described above, that occurs during execution of the assignment statement, is treated as an operational error.

If the conversion of a real value causes overflow, because the converted value does not fit into a long integer, an appropriate minimum or maximum value is substituted for the result.

Examples:

```
:REACT_LOGIC:HEAT.BI0001 := TRUE
opr_res := -1
```

Assignment, Array

Format

`array reference := array reference`

where an `array reference` can be: an external reference, a user array parameter, or a local variable array name.

Description

While scalar assignment statements can manipulate the individual elements of arrays, array assignment statements can assign whole arrays, arithmetic or string, to each other with one statement.

The right hand side of the statement should be an internal or an external reference; that is, the right side cannot be an expression of more than one operand or operator.

Elements of external arrays cannot be accessed individually; that is, you cannot index within an external array.

If the right hand side is an external array parameter, an operational error occurs at run time because a whole array cannot be pushed onto the internally used evaluation stack. As a result, elements of two external arrays cannot be moved from one to the other with a single array assignment statement.

If the right hand side is a scalar and the left hand side is an array, no operational error occurs, but only the first element of that array is assigned a new value.

The data types of the left and right hand side have the same compatibility requirements as Scalar Assignment statements.

If the source and destination internal references do not have the same number of elements, the compiler issues an error message. If the number of elements are equal and the dimensioning is different, the compiler issues a warning. For instance, two arrays dimensioned [3, 2, 4] and [12, 2] can be assigned to each other. Note that arrays are stored in memory with the last index moving fastest.

If either side is an external reference, the compiler cannot check its size. If the number of elements differ, then:

- If the destination array contains fewer elements than the source array, an operational error is raised and all the elements in the destination array remain unchanged.

  or

- If the destination array contains more elements than source array, only destination elements in excess remain unchanged.

As with the scalar assignment, the array assignment is in one of two categories: arithmetic and string.

An external reference cannot be a string array. Therefore, the compiler does not allow a string-type array assignment to have external references. If both sides of an array assignment are external references, the compiler translates the assignment as an arithmetic array assignment.

Examples:

```
BA0003 := BOOL_VALUES
:COMP:BLK.RA0002 := IA0001
```

The number of array dimensions can be up to 256. There is **no** limit on the number of elements you can define for an array.

The following are examples for assignments to sequence logic array:

```
BA0001[3] := FALSE;
USER_BA := SET_ARRAY(0);
USER_BA[2, 2, 2] := BA0001[3];
USER_BA[2, 3, 4] := TRUE;
IA0001[16] := 100;
```

Bit_pattern

Format

```
destination := BIT_PATTERN ( source )
```

Description

The BIT_PATTERN statement enables:

- The packing of a set of Boolean values into a bit pattern
- The unpacking of a bit pattern into a set of Boolean values

- The bit pattern that can be stored in a single user-integer parameter (II000x or IO000x) or in an integer variable, either block or subroutine.

The set of Boolean values can consist of:

- All Boolean user-input parameters
- All Boolean user-output parameters
- One of the four Boolean user-array parameters
- A Boolean local-array variable.

A Boolean formal argument:

- The Boolean parameter (or variable or argument) determines how many bits are used in the user integer parameter. When a Boolean local array is packed into a user integer parameter, up to 32 elements are packed. Excess elements are ignored without an error being raised.
- The data type on the left side of the := sign in the BIT_PATTERN statement determines whether the bits are packed or unpacked. If the left side is a user integer parameter, a set of Boolean values are packed into the integer.

Examples:

```
IO0004 := BIT_PATTERN ( BA0003 )
```

{ All 16 elements of `BA0003` are packed as a bit pattern into `IO0004`}

```
IO0008 := BIT_PATTERN ( BO0001 )
```

{All 16 boolean output parameters are packed as a bit pattern into `IO0008`}

```
II0001 := BIT_PATTERN ( Loc_Bool_Arr )
```

{Up to 32 elements of `Loc_Bool_Arr` are packed as a bit pattern into `II0001`}

```
BI0001 := BIT_PATTERN ( II0002 )
```

{24 bits of `II0002` are unpacked to give all boolean input parameters a boolean value}

```
BO0001 := BIT_PATTERN ( Loc_Int_Var )
```

{16 bits of the `Loc_Int_Var` are unpacked to give all boolean output parameters a boolean value}

Exit

Format

```
EXIT
```

Description

When executed, the EXIT statement terminates the block algorithm. Any statements following EXIT are not executed. You can use the EXIT statement in subroutines, in SBXs, and/or the main code section.

Example:

```
STATEMENTS
<<fill>>
SENDMSG (“Beginning New Batch”) TO status_stg;
ACTIVATE :REACT_LOGIC:HEAT;
```

```
WAIT 1.0;
WAIT UNTIL fill_act = FALSE AND fill_flg = FALSE
AFTER 600 GOTO fill_fail;
EXIT;
<<fill_fail>>
SENDMSG (“Fill time too long 1:Retry 2:Abort 3:Continue”) TO
status_stg;
.
.
.
ENDSEQUENCE
```

NOTE

In this example, the statements after EXIT are not executed when conditions are normal. However, if conditions specified in the AFTER clause of the WAIT UNTIL statement are met, statement execution skips the EXIT statement and goes to the statement labeled fill_fail. Then statement execution terminates at ENDSEQUENCE.

Exitloop

Format

```
EXITLOOP
```

Description

The EXITLOOP statement directs control flow to the point right after the nearest enclosing loop. It can only occur within a loop.

Example:

```
FOR batchno := 1 TO 5 DO
SENDMSG (“Beginning batch number,” batchno) TO status_stg;
    IF a = 1 THEN
        GOTO startcycle;
    ELSE
        EXITLOOP;
    ENDIF;
ENDFOR
```

NOTE

The EXITLOOP statement terminates the FOR loop, not the ELSE clause.

For

Format

```
FOR control variable := expression TO expression DO
    statement...
    statement...
```

```
    statement...
ENDFOR;
FOR control variable := expression DOWNTO expression DO
    statement...
    statement...
ENDFOR
```

Description

The FOR statement executes a group of statements a specified number of times. The control variable determines the number of repetitions.

The control variable is an integer. It can be a user output parameter (labeled or not) or a local variable (either block or subroutine) known by the sequence block, or an arithmetic user parameter name (II000n or IO000n).

The first expression in the statement is the initial value of the control variable, the second expression is the final value of the control variable for which the DO statement is performed. Both expressions should yield an integer value. If they do not, a compiler error is generated in case of internal references or a run-time error is generated in case of external references.

Statements in the FOR loop are executed until the control variable reaches its final value. After each repetition, the control variable is incremented or decremented by one until the final value is reached. The keyword TO increments the value. The keyword DOWNTO decrements the value.

The control variable value is tested before statements in the loop are executed. If the initial value of the control variable is equal to (or greater than) the final value, when the keyword, TO, is used, the statements are not executed. Likewise, when the keyword, DOWNTO, is used, the statements are not executed if the initial value is equal to (or less than) the final value.

For a positive increment, the FOR statement terminates when the control variable exceeds the second expression. Likewise, for a negative increment, the FOR statement terminates when the control variable is less than the second expression. These conditions are tested before executing the DO statements. Therefore, if either of these conditions exist initially, the DO statements are not executed. The value of the control variable, after leaving the FOR statement, is undefined.

If evaluating one of the expressions sets the Bad Status Attribute, the Bad Status is ignored and the statement uses the value as if it were not marked Bad. This can occur if:

- One of the operands of the expression is an input parameter that belongs to the block and is currently serving as a data sink of a linkage.
- One of the operands of the expression is an external reference to an input, or output, parameter that has its Bad status attribute set.
- If an operand is an external reference and its value cannot be retrieved at run time, an operational error is raised.

Examples:

```
FOR batchno := :REACT_COMP:DRAIN.IO0001 TO end_cntr DO
    SENDMSG (“Beginning batch number,” batchno) TO MSGGR2;
ENDFOR
```

The following FOR statement,

```
FOR i := 42 TO 50 DO
    asciinr := i;
    :'dcomp':V0'asciinr'.AUTOPN := FALSE;
ENDFOR
```

can replace the following nine commands:

```
:'dcomp':V042.AUTOPN:=FALSE;
:'dcomp':V043.AUTOPN:=FALSE;
:'dcomp':V044.AUTOPN:=FALSE;
:'dcomp':V045.AUTOPN:=FALSE;
:'dcomp':V046.AUTOPN:=FALSE;
:'dcomp':V047.AUTOPN:=FALSE;
:'dcomp':V048.AUTOPN:=FALSE;
:'dcomp':V049.AUTOPN:=FALSE;
:'dcomp':V050.AUTOPN:=FALSE
```

Goto

Format

```
GOTO label
```

Description

The GOTO statement provides transfer of the flow of control to any labeled statement within the same coded programming entity (block main section, subroutine, or SBX).

Flow of control cannot be transferred to a statement inside another block.

You can refer to a statement label, or a step label, within the block's main code section only from a statement within that same main code section or from a statement in one of the block's SBXs; that is, you can use the GOTO statement to jump out from an SBX into the main code section.

The HLBL compiler generates an error message if the program tries to use the GOTO statement to:

- Jump into a subroutine
- Jump out from a subroutine
- Jump into a standard block exception handler.

Example:

```
IF amount < REACT_COMP2:MIXER.RI0001
    THEN GOTO drain_fail;
ENDIF;
<<drain_fail>>
SENDMSG (“Draining too slowly 1:Retry 2:Abort 3:Continue”) TO
status_stg
```

If

Format

```
IF expression THEN statement;
[ELSEIF expression THEN statement...;]
```

```
[ELSE statement;]
ENDIF
```

Description

The IF statement provides conditional statement execution. Statements are executed according to the evaluation of a conditional expression. The ELSEIF and ELSE clauses are optional. When an expression in the IF clause or an optional ELSEIF clause in the statement is true, the THEN statement following the expression is executed. In an IF statement that contains an optional ELSE clause, or if the IF clause expression and all optional ELSEIF clauses are false, the statement following the ELSE clause is executed.

An IF statement can have multiple ELSEIF clauses and/or one ELSE clause.

If evaluating the expression sets the Bad status attribute, the block ignores it and interprets the value as if it were not marked Bad. The Bad status attribute can be set when:

- One of the operands of the expression is an input parameter that belongs to the block itself and the parameter serves as a data sink of a linkage.
- One of the operands of the expression is an external reference to an input or output parameter that has its Bad status attribute set.

An operational error is raised if one of the operands is an external reference and its value cannot be retrieved at run time.

Examples:

```
IF a <1 OR a >3 THEN
    GOTO error_msg;
ENDIF;
IF a = 1 THEN
    GOTO startcycle;
ELSEIF a = 2 THEN
    GOTO stopcycle;
ELSE
    GOTO error_msg;
ENDIF;
IF a = 1 THEN
    GOTO startcycle;
ELSE
    GOTO error_msg;
ENDIF
```

When a CALL subroutine statement is the last statement of an IF/THEN clause, and is executed, the STEPNO parameter indicates the step number of the CALL subroutine statement. The parameter, STMNO, indicates the statement being executed, or to be executed next, within the current subroutine.

Monitor Case

Format

```
MONITOR
mb_heading;
casenumber boolean expression --> output parameter label;
casenumber WHEN boolean expression [ --> output parameter label] DO
```

```
:compoundname:blockname;
ENDMONITOR
```

Description

The Monitor Case statement only applies to the Monitor (MON) block. Each MON block contains one Monitor Case statement which can be assigned up to 16 cases for evaluation. All expressions in the Monitor Case statement must yield a Boolean value when evaluated. The data type is checked at compilation time. No external references are allowed in monitor expressions.

Each time a Monitor Case is evaluated the Boolean result is stored in the corresponding Boolean output parameter. When a compound:block name is specified it can activate any sequence block (EXC, DEP, IND, or MON).

The user-labeled Boolean output parameter is not declared in the heading section. It is named in the case where it is used to save a Boolean value. That user-label is only introduced to convey the Boolean result of a particular Monitor Case onto another one. See Table 3-3 for the maximum length of a user parameter.

The Monitor block heading has two forms:

- Only user-label declarations without any keywords
- Constants definition and the user-label declaration preceded by keywords.

Before you have added any cases, the HLBL source of a monitor looks like this:

```
MONITOR
0001 ;
0002 ;
0003 ;
0004 ;
0005 ;
0006 ;
0007 ;
0008 ;
0009 ;
0010 ;
0011 ;
0012 ;
0013 ;
0014 ;
0015 ;
0016 ;
ENDMONITOR
```

Example:

```
MONITOR
    level_hi : BI0001;
    level_lo : BI0002;
    level_rec1 : BI0001;
CASES
0001 WHEN level_hi DO :REACT_LOGIC:HI_LVL_EXC; {only run block}
0002 level_hi --> level_rec1; {only set bit}
.
.
.
0016 WHEN level_lo--->level_rec1 DO :REACT_LOGIC:LO_LVL; {set bit and
```

```
run block}
ENDMONITOR
```

NOTE

1. Cases can be added after each of the case numbers listed.
2. Case numbers need not have a case assigned to them, but all case numbers must be listed.
3. A case statement can be preceded by an operator remark that is following a case number.
4. If the Bad status attribute is set as a result of evaluating the expression, the expression is interpreted as being false. The value of Bad status attribute is sent to the associated Boolean output parameter.
5. Only active patterns, as determined by the parameter ACTPAT, are evaluated.

Repeat

Format

```
REPEAT statement... UNTIL expression
```

Description

The REPEAT statement executes a series of statements repeatedly until the value of a conditional expression is true. Statements in the REPEAT statement are executed at least once.

Example:

```
time: = 0;
REPEAT
IF opr_res < 1 OR opr__res > 3 THEN
    GOTO drain_fail;
ENDIF;
time = time + 1;
WAIT 1;
UNTIL time = 50;
```

If evaluating the expression sets the bad status attribute, the block ignores it and interprets the value as if it were not marked Bad. The Bad status attribute can be set when:

- One of the operands of the expression is an input parameter that belongs to the block itself and the parameter serves as a data sink of a linkage.
- One of the operands of the expression is an external reference to an input or output parameter that has its Bad status attribute set.

An operational error is raised if one of the operands is an external reference and its value cannot be retrieved at run time.

Retry

Format

```
RETRY
```

Description

The RETRY statement is an alternate exit point from the two error handling SBXs. You can use RETRY only with the two error handling SBXs. Any other use produces a compiler error.

When an error handling SBX is terminated with ENDEXCEPTION, statement execution resumes with the statement following the one that caused the SBX to be executed. The RETRY statement directs the logic to re-execute the erroneous statement that caused the SBX to be executed.

— NOTE

Use the RETRY statement with care. The execution of an error handling SBX suppresses the generation of an alarm, and the switching of the block state to Manual. The RETRY statement permits the repeated execution of the erroneous statement, with no check, or indicator, of the number of retries. It is advised that you build a counting mechanism (see Example) and/or a security check into the SBX.

Example:

```
    BLOCK_EXCEPTION      TO_SYS_ERROR
    #if PHASE == FINAL APPLICATION
    { use this retry counting after all tests have passed }
    IF Retry_Cnt < 3 THEN
    Retry_Cnt := Retry_Cnt + 1;
    RETRY ;
ELSE
    SENDCONF (“Should”, COMPOUND103,”;”,BLOCK14,” do another 3
retries?”)
    TO MSGGR1 AFTER 60 GOTO no_answer;
    Retry_Cnt := 0;
    RETRY ;
ENDIF
<<no_answer>>
SENDMSG (“No confirmation on retry, “, BLOCK14, “ de-activated”) TO
MSGGR1;
EXIT
```

Sendconf

Format

```
SENDCONF ([literal, literal..], [expression, expression..]) TO
parameter AFTER expression GOTO label
```

Description

The SENDCONF statement sends messages interactively to logical devices or objects that act like logical devices (such as historians and printers).

Upon executing the SENDCONF statement, the standard block parameter SUSPND (suspended on message) is set true and statement execution is suspended until SUSPND is set to false again from outside the block, or until an optional time-out value expires.

The message must be less than 80 characters.

The message is a string composed of any combination of string literals and expressions.

Commas separate individual strings and expressions in the message.

The label can be either a statement label or a step label.

Examples:

```
SENDCONF ( “Operational Error is”, OP_ERR ) TO MSGGR3;
SENDCONF ( “Suspension lasts for”, II0001, “sec.” ) TO MSGGR2
AFTER II0001 GOTO DRAINING
```

Sendmsg

Format

```
SENDMSG ([string literal, string literal...], [expression, expres-
sion...]) TO parameter
```

Description

The SENDMSG statement sends messages to logical devices or objects that act like logical devices (such as historians and printers) and user-labeled string parameters.

- The message is a string composed of any combination of string literals and expressions.
- Commas separate individual strings and expressions in the message.
- The message must be less than 80 characters.
- When an expression that is part of a message, is evaluated, the results are converted into strings as follows:
  - A real value is converted into a string of fixed length. The string format is:

    `sx.xxxxxxesyy`

    where:

`s`	is a sign character (`+` or `-`).
`x`	is a digit. One digit before the decimal point and up to six digits after the decimal point maximum.
`e`	is the symbol for exponentiation.
`yy`	is the exponent, a maximum of two digits.

    For example: `+0.000000e+00, -1.234567e-12`

- A boolean value is converted into the string FALSE or TRUE.
- An integer value is converted into a string of digits. A negative integer value is preceded by a minus sign. The length of the string produced depends on the size of the value.
- An enumerated value is converted using the **ORD** function and then treated like an integer. The symbolic value is not converted into string format.

There are four message groups in a sequence block: namely, MSGGR1, MSGGR2, MSGGR3, and MSGGR4. You can assign each of these parameters to one of eight message groups.

The Station block has eight alarm groups. These groups are available to all blocks in any compound in that station as message group numbers 1 to 8.

Each control block has its own compound groups available as message groups 1 to 4.

You can send a message to any message group.

To send a message to message group 5:

1. Configure the MSGGRx block parameter value to 5 (where x = 1 to 4).
2. Use the following HLBL statement (where x = 1 to 4):

   ```
   SENDMSG ("user's message") TO MSGGRx
   ```

   In this example, the message is sent to all logical devices assigned to group 5.

   Up to eight (8) logical devices can be assigned to each message group in a compound, via the GRxDVy parameters (x = 1 to 3, and y = 1 to 8). Each of these parameters can be configured to contain the logical name of the device. A logical device belongs to only one message group.

   Up to 16 logical devices can be assigned to each message group in the Station block, via the GRx and DVnn parameters (x = 1 to 8, and y = 1 to 16). The DVnn parameters can be configured to contain the logical names of the devices. The GRx parameters are packed Boolean bit maps that can be configured to specify which of the DVnn devices belong to message groups x. The same logical device can belong to more than one group.

   In the above example, to send the message to logical devices 1, 3, 9, and 15 in message group 5, you should configure the following Station block parameters:

DV1	**<logical_name>**
DV3	**<logical_name>**
DV9	**<logical_name>**
DV15	**<logical_name>**
and	
GR5	**0xA082**

   where:

0x	specifies hexadecimal notation
A =	1010 in binary for devices 1 and 3

`0 =` 0000 in binary for no devices

`8 =` 1000 in binary for device 9

`2 =` 0010 in binary for device 15

You can also send a message to a user-labeled string parameter that belongs to the block. The Operator Message Interface can then retrieve the message for operator display. The standard parameter MSGNO belonging to the block is incremented by one each time a new message is assigned to any of the user-labeled string parameters.

NOTE

1. Using an out-of-range value for x in the MSGGRx entry clamps the value at 1 or 8, whichever is nearest.
2. When sent to a message group parameter, the message is appended to a list of fixed message fields and sent to all logical devices associated with the message group.
3. To send messages to the Historian, you should assign the message to a MSGGRx parameter that has the Historian in its group of associated devices.
4. The compiler raises an error if the parameter name in the TO clause is not one of MSGGRx parameters or a user-string parameter.

Examples:

```
status_flg : SN0001 ;
.
.
.
SENDMSG (“Beginning New Batch”) TO status_flg;
SENDMSG (“Beginning Batch Number = ”, batchno) TO MSGGR2
```

Set_sbxs

Format

```
SET_SBXS (availability pattern)
```

Description

The SET_SBXS statement uses its availability pattern to manipulate the five block parameters DISBX1, DISBX2, DISBX3, DISBX4, DISBX5. When an SBX is “Enabled” (DISBXx = False) the SBX is available, and the block executes the statements specified in the SBX when the corresponding event occurs.

The availability pattern is a string literal whose only three valid characters are:

`E` which enables the corresponding SBX

`D` which disables the corresponding SBX

`-` which disregards the corresponding SBX

As long as an SBX is set “Disabled” (DISBXx = True) the SBX is unavailable and its logic is not executed.

Two standard block exception handlers (SBXs) are supported for error handling and three are supported for state change handling.

SBX Number	SBX Function
1	System error handler, OP ERR = no user error
2	User error handler, OP ERR = 2000 .. 3000
3	Transition logic for switch to Inactive
4	Transition logic for switch to Manual
5	Transition logic for switch to Paused

Each character in the availability pattern corresponds to a standard block exception handler. The first character refers to SBX1, the second character to SBX2, and so forth.

If the number of characters in the availability pattern is less than maximum number of SBXs, the availability states of the unnamed SBXs are left unchanged.

Example:

```
SET_SBXS ("E-EDD")
```

This statement:

- Enables the "TO_SYS_ERROR" SBX (DISBX1 = False)
- Leaves the "TO_USR_ERROR" SBX as it was (DISBX2 is not changed)
- Enables the "TO_INACTIVE" SBX (DISBX3 = False)
- Disables the "TO_MANUAL" SBX (DISBX4 = True)
- Disables the "TO_PAUSED" SBX (DISBX5 = True).

Start_timer

Format

```
START_TIMER (:compoundname:blockname.timername [, expression])
```

Description

The START_TIMER statement turns on a specified timer in a Timer (TIM) block. START_TIMER optionally initializes a timer to the number of seconds specified in the expression. The timer is then incremented by the value of the block's period each time the block is scheduled to be executed. Timers are updated as long as the TIMER block is in Auto and the compound is on.

Specify the timer name as TIMR1, TIMR2, TIMR3, or TIMR4.

If the addressed Timer block does not exist, an operational error is raised at run time.

NOTE

After starting a timer, you must suspend (for example, use a WAIT statement) before testing the timer. Otherwise, the timer block does run and you receive an invalid timer status.

Examples:

```
START_TIMER (:REACT_LOGIC:FILL_TIME.TIMR1)
WAIT 0.5 ;
                {or}
START_TIMER (:REACT_LOGIC:FILL_TIME.TIMR1, 0.0)
WAIT 0.5 ;
```

Stop_timer

Format

```
STOP_TIMER (:compoundname:blockname.timername)
```

Description

The STOP_TIMER statement turns off a specified timer in a Timer (TIM) block.

Specify the timer names as: TIMR1, TIMR2, TIMR3, or TIMR4.

If the addressed Timer block does not exist, an operational error is raised at run time.

Example:

```
STOP_TIMER (:REACT_LOGIC:FILL_TIME.TIMR1)
```

Wait/Wait Until

Format 1

```
WAIT time_period
```

Format 2

```
WAIT UNTIL controlling_expression AFTER time_period GOTO label
```

Description

Statement execution can be delayed with a WAIT or WAIT UNTIL statement.

WAIT delays execution a specified number of seconds.

WAIT UNTIL delays execution until the result of the controlling_expression becomes true. The AFTER – GOTO clause in the WAIT UNTIL statement is optional.

The optional AFTER – GOTO clause allows you to specify a time limit on the WAIT period, and an alternate action, if that time limit is exceeded. Specifically, if the controlling expression does not go TRUE within the specified number of seconds, the statement specified in the GOTO clause is executed. If the controlling expression does go true within the time limit, the statement following the WAIT UNTIL statement is executed.

In both the WAIT and WAIT UNTIL statements, time_period is in seconds, and can be a constant or an expression yielding an integer or real value.

For either statement, block execution is suspended until the next regular cycle (per the block's period) that equals or follows time_period. If time_period is less than 0.1, block execution is always suspended by one period, regardless of the value of the period. This means that a WAIT 0.0; statement will still wait for one block period.

Measurement of time is independent of the state (Active, Paused, Inactive) of the block in which the statement appears and the frequency at which the block is processed.

Examples:

```
WAIT 1.0;
WAIT UNTIL fill_actv = FALSE AFTER 600 GOTO fill_fail;
.
.
<<fill_fail>>
SENDMSG (“Filling too slowly 1:Retry 2:Abort 3:Continue”) TO
status_stg
```

While

Format

```
WHILE expression DO statement... ENDWHILE
```

Description

The WHILE statement executes a series of statements repeatedly while a given conditional expression is true. Statements in the WHILE loop are not executed at all if the specified condition is not true initially.

Example:

```
WHILE count <= 5 DO
    value := value * count;
    count := count + 1;
ENDWHILE
```

If evaluating the expression sets the Bad status attribute, the block ignores it and interprets the value as if it were not marked Bad. The Bad status attribute can be set when:

- One of the operands of the expression is an input parameter that belongs to the block itself and the parameter serves as a data sink of a linkage.
- One of the operands of the expression is an external reference to an input or output parameter that has its Bad status attribute set.

An operational error is raised if one of the operands is an external reference and its value cannot be retrieved at run time.

When a CALL subroutine statement is the last statement of a WHILE loop, and is executed, the STEPNO parameter indicates the step number of the CALL subroutine statement. The parameter, STMNO, indicates the statement being executed, or to be executed next, within the current subroutine.

Appendix A. Sequence Control Error Messages

Operational Errors occur at runtime when a sequential control block encounters an error that it cannot recover from by itself. Operational error codes are stored in the block's OP_ERR parameter and can be displayed during operation. Here are some conditions that can cause operational errors:

- Data type inconsistencies in an external reference are discovered.
- A statement tries to set the value of a secured parameter. For example, an ACTCASES statement cannot address a MON block if the block's ACTPAT parameter is secured.
- An external reference is defined to a compound, block, or parameter that is not defined in upper case.
- A statement references a parameter or block that does not exist. For example, a START_TIMER statement cannot address a TIM block that does not exist.
- A statement cannot address a parameter or block due to communication errors. For example, the ABORT statement fails to deactivate a block.

Operational Errors (OP_ERR) messages listed in the following sections are those which can be raised in a Monitor or a Sequence block. When this occurs, the OP_ERR parameter assumes an integer that indicates the type of failure.

The 2000 and 3000 series OP_ERR messages are detected by the sequence control blocks. The values of OP_ERR less than 2000 are simply conveyed by the sequence control blocks.

These error messages are classified as either priority one (High) or priority two (Low) messages. Priority one messages, requiring immediate attention, are those sent to the workstation processors and/or personal computers designated to report system alarms. This type is also sent to the historian and to a high-priority message printer. Priority two messages, those usually not requiring your immediate attention, are sent to the historian and to a low-priority message printer.

The following string text represents a typical message line that is printed via a logging device to a priority printer. The standard message line contains the following constants:

(mm-dd-yy)	month-day-year
(HH:MM:SS)	hour:minute:second
(PRI <subsystem>)	the primary subsystem (for example, PRI SYSMON=SYSMN1)
(IPC)	the subsystem text string file
(-01071)	the file code
"Invalid station address and/or letterbug"	the Message

Run-Time Data Conversion Errors

Table A-1. Run-Time Data Conversion Errors

Code	Description
2101	attempt to assign a Boolean value to a user parameter or variable with a data type of integer or real
2102	attempt to assign a non-Boolean value to a Boolean user parameter or variable
2103	attempt to write to an array value(s) which cannot be converted to integer or real
2116	more external array elements read than can be assigned to destination array

HLBL Semantics Violation Errors

Table A-2. HLBL Semantics Violation Errors

Code	Description
2301	in an arithmetic expression, the data type is unequal to BOOL, LONG, or FLOAT encountered, OR in a string expression, the data type is unequal to STRING encountered. (NOTE: External string parameters must be preceded by the keyword STRING in string expressions. For example, SN0005: = STRING : CMP1: BLK1.SN0001)
2302	only a Boolean data type is permitted in a logical expression
2303	a Boolean data type is not permitted in an arithmetic expression
2304	only an integer data type is permitted in a DIV or MOD function
2305	operands should be compatible for (in)equality
2306	operands should be integer or real for equality operator
2307	a Boolean data type is not permitted in a TRUNC, ROUND, ABS, or SQRT function
2308	a real data type is not permitted in an ORD function
2309	timer value is not treated as a real value
2310	array index out of bounds
2311	array index expression not of data type long
2312	first subset position out of bounds
2313	second subset position out of bounds
2314	string type expected (in string expressions)
2315	first subset position expression not of data type long
2316	second subset position expression not of data type long
2317	input parameter is not writable: an attempt was made to write to a linked input parameter; check reason for linkage and reconfigure, if necessary.

Environmental Interaction Errors

Table A-3. Environmental Interaction Errors

Code	Description
2401	Indicates an attempt to set the ACTIVE parameter to a value it presently assumes
2402	Monitor case cannot activate block because the ACTIVE parameter of the target block is the sink of the linkage or connection

Sequence Control Block System Errors

If the OP_ERR assumes one of the following values, a more serious error has occurred (that is, the block is incorrectly installed, a memory corruption has occurred, and so forth). The error condition encountered is one that is normally checked by the HLBL compiler or the run-time system. Carefully examine the possible cause of one of the following errors before any further block processing is performed.

Table A-4. Sequence Control Block System Errors

Code	Description
3001	bad SRC parameter detected by Monitor block
3002	deleted SRC block detected by Monitor block
3003	no input parameter located by Monitor block
3004	evaluation stack empty
3005	evaluation stack full
3006	invalid bit number
3007	invalid category number
3008	invalid constant type
3009	invalid function
3010	invalid message group index
3011	invalid message number
3012	invalid operator code This indicates a statement has been made that executes an ILLEGAL OPERATOR such as ADD, MUL, DIV, and so forth, with an ILLEGAL OPERATION such as a WAIT, JUMP, SEND MESSAGE, and so forth. This statement must be located and corrected. The Standard Block Exception Handler (SBX) can locate this statement, and is discussed in section “Useful Hints on SBX Programming” in *Control Processor 270 (CP270) and Field Control Processor 280 (CP280) Integrated Control Software Concepts* (B0700AG).
3013	invalid operator
3014	invalid parameter number
3015	invalid path
3016	invalid return from IC (interpretive code) processor

Table A-4. Sequence Control Block System Errors (Continued)

Code	Description
3017	statement request and IC-PTR null
3018	invalid data type on stack (stack may be corrupted)
3019	statement not implemented
3020	string corrupted
3021	(not being used)
3022	message not queued
3023	no message packet allocated
3024	error return from string pool function stripped
3025	invalid vector type
3026	invalid scalar type
3027	invalid argument type
3028	invalid return from SUBR processor
3029	invalid path indication
3030	no FPN (full path name) construct
3031	invalid return from SBX (standard block exception) processor
3032	HLBL statement not found
3033	FPN construct too long
3034	invalid array operation
3035	invalid array category
3036	invalid data type
3037	invalid length for CPGET
3038	invalid length for GETVAL
3039	statement not found
3040	STMNO out of bounds
3041	real value too big (real to ASCII conversion – conversion to ASCII failed)
3042	real value too small (real to ASCII conversion – conversion to ASCII failed)
3101	invalid parameter type
3102	data type of loop variable not integer
3103	no conversion to string
3201	ERH entry for ACTIVATION is not serviced
3202	invalid index of ERH-entry
3203	(not being used)
3204	not an ACTIVATION ERH-entry
3205	request type mismatch
3206	invalid data type CPSET
3207	invalid data type setconfirm
3208	invalid length CPSET
3209	invalid length setconfirm

Table A-4. Sequence Control Block System Errors (Continued)

Code	Description
3301	data type of monitor expression is not boolean

CPGET/CPSET Error Messages

The following codes are database access return codes generated by cpget(), cpset(), cpsecure(), and/or cprelease(). These codes are returned to OM software in response to user OPEN/CLOSE/GETVAL/SETVAL calls.

Table A-5. CPGET/CPSET Error Messages

Code	Description
2	INACTIVE – the parameter ACTIVE already has the value you are trying to write (this is a warning, not an error)
3	SECURED – attempt to set a parameter that is secured
4	NOT_SECURABLE – the point is not securable
5	NOT_SECURED – the point is not secured
6	NOT_RELEASABLE – the point is secured and cannot be released
7	(not being used)
8	(not being used)
9	TO_BOOL_CONV_ERROR – value cannot be converted into boolean
10	TO_STRING_CONV_ERROR – value cannot be converted into string
11	TO_INTEGER_CONV_ERROR – value cannot be converted into integer
12	TO_REAL_CONV_ERROR – value cannot be converted into real
13	TOO_BIG – attempt to write a real value greater than an integer parameter can hold
14	NOT_CONNECTABLE – the point is not connectable
15	NOT_SETTABLE – attempt to set a parameter that is not settable
16	TOO_MUCH – attempt to write too much data to parameter, or not enough room available to read value from parameter.
17	OUT_OF_ENG_RANGE – the point was set beyond the engineering range
18	LOCKED_ACCESS – locked access to the parameter
19	TO_CHAR_CONV_ERROR – to character conversion error
20	GETPTRP_ERROR – the string could not be retrieved from the string pool
21	PUTSTR_ERROR – the string could not be stored in the string pool
22	ONLY_LONG_FOR_ACTPAT – The ACTPAT parameter of the Sequence Monitor block requires a data type of long integer; no other data type is acceptable
23	TO_PACKED_CONV_ERROR – to packed Boolean conversion error
24	OUTPUTS_DISABLED – outputs are disabled
25	TOO_LITTLE – too little (for example, an attempt to write a large INTEGER value into SHORT INT, or an attempt to write a negative value into a parameter with the “non_negative” rule [refer to SPRATE of PIDA])

The following are Object Manager error/exception message codes.

Table A-6. Object Manager Error Codes

Code	Description
0	OM_SUCCESS – no error
-1	ENOTFOUND – top level name not found or internal time-out expired
-2	EBADTYPE – invalid data type argument
-3	ESECURE – attempt to write to a secured variable
-4	ENOTOPENED – invalid om_descriptor
-5	EREADERROR – inaccessible change-driven variable
-6	EREAD – om_descriptor has write-only access
-7	EWRITE – om_descriptor has read-only access
-8	EWRITEERROR – change-driven variable write
-9	ENOSPACE – memory allocation error or shared memory attach failure
-10	EDUPLICATE – entry is already in table
-11	ELOCAL – name is local; cannot import
-12	EBADNAME – invalid full pathname
-13	EBADLEN – all data not returned (receive buffer too small) for example, SN0001 := ; SN0001 can accommodate 80 characters, but a shared variable of the String type is good for 255 characters
-14	ESTRLEN – string length is invalid
-15	EOPENED – omopen list is already opened
-16	EBADLIST – bad variable list pointer
-17	EBADATBL – bad address table pointer
-18	ECONNBAD – omopen failure; no connections
-19	EQNOTEMPTY – change queue is not empty
-20	ENOVALUE – object type has no value record
-21	EBADVREC – bad value record
-22	EIMPORT – not enough space for import table
-23	EOBJDIR – not enough space for object directory
-24	EADDRTBL – not enough space for address table
-25	ESCANDB – not enough space for scanner database
-26	EIMPFULL – import list is full
-27	EBADQTBL – not enough space for dqchg queue table
-28	EBADQUE – error creating a queue
-29	EOMWSIZE – invalid omwrite list size
-30	EQEMPTY – dqchg queue is empty
-31	ENOADDTBL – optimized omopen without address table
-32	ENOQUE – unable to get a queue for omopen
-33	ENOSEND – unable to send omopen requests

Table A-6. Object Manager Error Codes (Continued)

Code	Description
-34	ENOADRSP – caller's address table is full
-35	ENOTACTIVE – caller not activated with IPC
-36	EBCONTBL – error creating connection table
-37	ENODISQ – error creating disconnect queue
-38	ENOOPNTBL – error creating open ID table
-39	EMAXOPNS – the omopen ID table is full
-40	ESCANFUL – the scanner database is full
-41	EIPCRET – error detected in IPC system
-42	EWAIT – remote scanner has not responded
-43	ESTATION – communication problem with remote station (remote station has not responded)
-44	EOBJPND – object with same name pending creation
-45	ENOCONFIRM – no setval confirmation
-46	ENOVENQ – unable to get omvenix queue
-47	ENOSCANTSK – unable to create scanner task
-48	ENOVENTSK – unable to create omvenix task
-49	ENOSRVTSK – unable to create server task
-50	EFABORT – a fabort was received from VENIX
-51	ESIZELST – invalid list size
-52	ENORECQ – error creating reconnect queue
-53	EBADRSZ – invalid users address table size
-54	ENORECTSK – error creating OM reconnect task
-55	OM_ECBSY – open failure; all connections busy
-56	EOMOSIZE – invalid open variables list size
-57	EBADINDEX – invalid open variables list index
-58	EBADMSK – bad status write mask
-59	EBADWKSTA – Workstation name length
-60	EBADPARM – parameter does not exist; for example, this error is generated when a full pathname refers to a block that exists in the control database, but the parameter with given name does not exists in the block

Index

H

I

M

O

P

R

S

T

U

W

Invensys Systems, Inc.
10900 Equity Drive
Houston, TX 77041
United States of America
http://www.invensys.com

Global Customer Support
Inside U.S.: 1-866-746-6477
Outside U.S.: 1-508-549-2424
Website: https://support.ips.invensys.com