## Comment Form
### RS3™ Manuals

## Rosemount Basic Language Manual

### Please give us your feedback to help improve this manual.

1. Do you actually use this manual when you are:
   - configuring  
   - making changes or enhancements  
   - operating the system  
   - troubleshooting  
   - other ________________  

<table>
<thead>
<tr>
<th>Never</th>
<th>Rarely</th>
<th>Sometimes</th>
<th>Usually</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Does this manual answer your questions?  

<table>
<thead>
<tr>
<th>Never</th>
<th>Rarely</th>
<th>Sometimes</th>
<th>Usually</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. What could be changed in this manual to make it more useful?

### Errors and Problems:  
Please note errors or problems in this manual, including chapter and page number, if applicable; or send a marked-up copy of the affected page(s).

May we contact you about your comments?  

- [ ] Yes  
- [ ] No

**Name**  
______________________________

**Company**  
______________________________

**Phone**  
______________________________

**Date**  
______________________________

Fisher-Rosemount FAX #: (512) 834-7200

Thank you!
About This Manual

Changes for This Release

- Removed references to SRU and Material History, which are now obsolete. (References to the SRU Data Folder remain, because they are still valid.)
- Added information about slowing while, for, and until loops with a sleep instruction to Chapter 1: Section 5 and Chapter 1: Section 6.
- Added a note to the description of the wait-bid and send-bid instructions in Chapter 1: Section 7.
- Added descriptions of status -2 and status -3 to the format of align_on and align_wait in Chapter 1: Section 7.
- Added return value information to the description of the input instruction in Chapter 1: Section 11.
- Added a note about the use of a blank line after sequences of indented lines to the descriptions of certain instructions.
- Changed Chapter 1: Section 13 to “Timestamp Functions” and removed the Material History information.

Revision Level for This Manual

<table>
<thead>
<tr>
<th>For This Software Version:</th>
<th>Refer to This Document:</th>
<th>Date</th>
<th>Part Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Title</td>
<td>P1</td>
<td>Rosemount Basic Language Manual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18R2</td>
<td>Rosemount Basic Language Manual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18R1</td>
<td>Rosemount Basic Language Manual</td>
</tr>
</tbody>
</table>
References to Other Manuals

References to other RS3 user manuals list the manual, chapter, and sometimes the section as shown below.

**Sample Entries:**
For ..., see CC: 3.  
For ..., see CC: 1-1.

**Manual Title**  **Chapter**  **Manual Title**  **Chapter-Section**

**Abbreviations of Manual Titles**
- **AL** = Alarm Messages
- **BA** = ABC Batch
- **CB** = ControlBlock Configuration
- **CC** = Console Configuration
- **DT** = Disk and Tape Functions
- **IO** = I/O Block Configuration
- **OP** = Operator’s Guide
- **OV** = System Overview and Glossary
- **PW** = PeerWay Interfaces
- **RB** = Rosemount Basic Language
- **RI** = RNI Release Notes and Installation Guide
- **RP** = RNI Programmer’s Reference Manual
- **SP** = Site Preparation and Installation
- **SV** = Service
Reference Documents

Prerequisite Documents

You should be familiar with the information in the following documents before using this manual:

- **System Overview Manual and Glossary** 1984-2640-21x0
- **Software Release Notes, Performance Series 1** 1984-2818-0110
- **Software Loading and Upgrade Procedure, Including Batch** 1984-2818-0210

Related Documents

You may find the following documents helpful when using this manual:

- **ABC Batch Quick Reference Guide** 1984-2818-1103
- **ABC Batch Operator’s Guide** 1984-2655-21x0
- **ABC Batch Software Manual** 1984-2654-21x0
- **Alarm Messages Manual** 1984-2657-19x1
- **Configuration Quick Reference Guide** 1984-2812-0808
- **Console Configuration Manual** 1984-2643-21x0
- **ControlBlock Configuration Manual** 1984-2646-21x0
- **Disk and Tape Functions Manual** 1984-2644-21x0
- **I/O Block Configuration Manual** 1984-2645-21x0
- **Operator’s Guide** 1984-2647-19x1
- **PeerWay Interfaces Manual** 1984-2650-21x0
- **RNI Programmer’s Reference Manual** 1984-3356-02x1
- **RNI Release Notes and Installation Guide** 1984-3357-02x1
- **Service Manual, Volume 1** 1984-2648-21x0
- **Service Manual, Volume 2** 1984-2648-31x0
- **Service Quick Reference Guide** 1984-2816-0904
- **Site Preparation and Installation Manual** 1984-2642-21x0
- **Software Discrepancies for Performance Series 1** 1984-2818-0311
- **User Manual Master Index** 1984-2641-21x0
# Contents

## Chapter 1: Rosemount Basic Language (RBL)

### Section 1: RBL Concept
- RBL Script Structures ................................................. 1-1-1
- Scripts ................................................................................. 1-1-3
- Phases ............................................................................... 1-1-4
- Labels ............................................................................... 1-1-5
- Symbols .............................................................................. 1-1-7
- Instructions ....................................................................... 1-1-8
  - Instruction Status ...................................................... 1-1-9
  - Instruction Syntax ..................................................... 1-1-9
  - Nesting Stack ............................................................. 1-1-10
- Comments ......................................................................... 1-1-11
  - Comment Examples .................................................. 1-1-11
  - Disappearing Comments ........................................... 1-1-11
- Tags and Addresses ......................................................... 1-1-12
  - ControlBlock Variable Values .................................... 1-1-12
  - ControlBlock Modes .................................................. 1-1-13
- Instructions That Retry ..................................................... 1-1-14
- Script and ControlBlock Synchronization ........................... 1-1-15
- RBL Parameter Limits ..................................................... 1-1-16
- Mathematical Operators .................................................. 1-1-18

### Section 2: Script Management
- Creating an RBL Files Folder and RBL File .............................. 1-2-2
  - Batch Script Screen ..................................................... 1-2-3
  - RBL File Contents screen ............................................. 1-2-4
- Maintaining Script Files ..................................................... 1-2-6
  - Printing a Script .......................................................... 1-2-6
  - Calling Up a Script ....................................................... 1-2-7
  - Deleting a Script .......................................................... 1-2-8
  - Copying a Script .......................................................... 1-2-9
- Writing and Editing Scripts .................................................. 1-2-10
  - Entering Text ................................................................ 1-2-11
  - Copying a Script or Script File from the Batch Script Screen 1-2-12
  - Deleting and Inserting Text ........................................... 1-2-13
  - Parsing and Writing Text to Disk .................................... 1-2-15
  - Reading and Moving Scripts .......................................... 1-2-16
  - Using Search and Replace Functions ............................... 1-2-18
Section 3: Numeric and String Symbols ........................................ 1-3-1

Shared and Private symbols .................................................... 1-3-2
Batch Recipe Use of Shared and Private Symbols ...................... 1-3-2
Batch Run Screen Use of Shared and Private Symbols .............. 1-3-4
Shared and Private Symbol Concept ....................................... 1-3-5
Defining Shared Symbols ..................................................... 1-3-6
Hex and Binary Values .......................................................... 1-3-6

Aliases .................................................................................... 1-3-7
Using Aliases .......................................................................... 1-3-9

Dot Operators ........................................................................ 1-3-9

Arrays .................................................................................... 1-3-10

aliasdim Array ...................................................................... 1-3-12

dim Array ............................................................................ 1-3-14

stringdim Array ..................................................................... 1-3-16

Virtual Arrays (Batch Only) ..................................................... 1-3-17

vdim Array ............................................................................ 1-3-18

vstringdim Array ................................................................... 1-3-19

rvdim and rvstringdim ........................................................... 1-3-20

Computing Disk Space for Virtual Arrays .............................. 1-3-22

Array Functions ..................................................................... 1-3-23

array_copy and array_init ..................................................... 1-3-23

Performance Improvements .................................................... 1-3-24

Array Copy and Initialize Concept ......................................... 1-3-25

array_copy Format ............................................................... 1-3-28

array_init Format ............................................................... 1-3-29

file_copy .............................................................................. 1-3-30

file_delete ............................................................................. 1-3-31

file_rename ........................................................................... 1-3-32

varray_file ........................................................................... 1-3-33

Constants ................................................................. 1-3-34

Global Variables ................................................................... 1-3-36

Local Variables .................................................................... 1-3-38

String Variables ................................................................... 1-3-39

String Characteristics ......................................................... 1-3-40

Hints for Using Symbols ....................................................... 1-3-42

Changing Variables ............................................................. 1-3-42

Making a Script Execute Sooner .............................................. 1-3-43

Conserving Volatile Memory ................................................ 1-3-43

Comparing Floating Point Values .......................................... 1-3-43

Section 4: RBL String Functions .............................................. 1-4-1

andbit$ and orbit$ ................................................................. 1-4-2

asc ....................................................................................... 1-4-4

byte2flag and flag2byte$ ....................................................... 1-4-5
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using byte2flag and flag2byte$</td>
<td>1-4-7</td>
</tr>
<tr>
<td>byte2flag and flag2byte$ Status Values</td>
<td>1-4-10</td>
</tr>
<tr>
<td>chr$</td>
<td>1-4-11</td>
</tr>
<tr>
<td>getbit</td>
<td>1-4-12</td>
</tr>
<tr>
<td>setbit$</td>
<td>1-4-13</td>
</tr>
<tr>
<td>str$</td>
<td>1-4-14</td>
</tr>
<tr>
<td>strstr$</td>
<td>1-4-15</td>
</tr>
<tr>
<td>strlen</td>
<td>1-4-16</td>
</tr>
<tr>
<td>swab$</td>
<td>1-4-17</td>
</tr>
<tr>
<td>val</td>
<td>1-4-18</td>
</tr>
<tr>
<td>Section 5: Branching and Conditional Instructions</td>
<td>1-5-1</td>
</tr>
<tr>
<td>chain and endchain</td>
<td>1-5-2</td>
</tr>
<tr>
<td>Using Shared Aliases and Constants in Chained Scripts</td>
<td>1-5-4</td>
</tr>
<tr>
<td>Using Private Aliases and Constants in Chained Scripts</td>
<td>1-5-6</td>
</tr>
<tr>
<td>for</td>
<td>1-5-8</td>
</tr>
<tr>
<td>gosub</td>
<td>1-5-10</td>
</tr>
<tr>
<td>goto</td>
<td>1-5-12</td>
</tr>
<tr>
<td>if, elseif, and else</td>
<td>1-5-14</td>
</tr>
<tr>
<td>if</td>
<td>1-5-14</td>
</tr>
<tr>
<td>else</td>
<td>1-5-16</td>
</tr>
<tr>
<td>elseif</td>
<td>1-5-18</td>
</tr>
<tr>
<td>return</td>
<td>1-5-20</td>
</tr>
<tr>
<td>start</td>
<td>1-5-21</td>
</tr>
<tr>
<td>until</td>
<td>1-5-22</td>
</tr>
<tr>
<td>while</td>
<td>1-5-24</td>
</tr>
<tr>
<td>Special While Cases</td>
<td>1-5-26</td>
</tr>
<tr>
<td>Section 6: Batch Management Instructions</td>
<td>1-6-1</td>
</tr>
<tr>
<td>abort</td>
<td>1-6-2</td>
</tr>
<tr>
<td>abort_task</td>
<td>1-6-3</td>
</tr>
<tr>
<td>backup</td>
<td>1-6-4</td>
</tr>
<tr>
<td>begin_task</td>
<td>1-6-5</td>
</tr>
<tr>
<td>break</td>
<td>1-6-6</td>
</tr>
<tr>
<td>bumpid</td>
<td>1-6-7</td>
</tr>
<tr>
<td>continue</td>
<td>1-6-8</td>
</tr>
<tr>
<td>cont_task</td>
<td>1-6-10</td>
</tr>
<tr>
<td>dishold and enhold</td>
<td>1-6-11</td>
</tr>
<tr>
<td>end</td>
<td>1-6-12</td>
</tr>
<tr>
<td>finish</td>
<td>1-6-13</td>
</tr>
<tr>
<td>halt</td>
<td>1-6-14</td>
</tr>
<tr>
<td>halt_task</td>
<td>1-6-15</td>
</tr>
<tr>
<td>hold</td>
<td>1-6-16</td>
</tr>
<tr>
<td>holding</td>
<td>1-6-17</td>
</tr>
<tr>
<td>kill_task</td>
<td>1-6-18</td>
</tr>
<tr>
<td>max and min</td>
<td>1-6-19</td>
</tr>
</tbody>
</table>
Section 7: Recipe Management Instructions ............................................... 1-7-1

Section 8: On Trap Instructions ............................................................. 1-8-1
Alarm Traps ................................................................. 1-8-7
Alarm Traps with Priority Designation ......................... 1-8-7
on Alarm Traps without Priority Designation ................. 1-8-8
System Traps .............................................................. 1-8-9
clear_alarm_queue ...................................................... 1-8-12

Section 9:  Alarm, Log, and Report Instructions ............... 1-9-1
alarm, almsg, and clear .............................................. 1-9-2
event ................................................................. 1-9-4
flog .............................................................. 1-9-5
open and close ..................................................... 1-9-7
  open and close Status Values .................................. 1-9-9
print and print$ ..................................................... 1-9-10
  print .......................................................... 1-9-10
  print$ .......................................................... 1-9-12
Print Options ......................................................... 1-9-13
  Format Control Options ....................................... 1-9-14
  Constant Conversion Options ................................. 1-9-15
  Decimal Conversion Options ................................. 1-9-16
  Variable Conversion Options ............................... 1-9-17
  System String Options ...................................... 1-9-18
quietlog and loudlog ................................................ 1-9-19

Section 10: Acquire/Release Instructions ....................... 1-10-1
Overview ............................................................ 1-10-2
Example of Acquiring a Task ..................................... 1-10-3
Acquire/Release Concept ......................................... 1-10-4
Acquire/Release Status Values .................................. 1-10-5
acquire and acqfirst .............................................. 1-10-7
acqwait ............................................................. 1-10-9
dequeue ........................................................... 1-10-10
masterof ............................................................ 1-10-12
on acquire_err ..................................................... 1-10-13
on lost_master ..................................................... 1-10-14
on lost_slave ....................................................... 1-10-16
  Conditions for Execution .................................... 1-10-17
on unacquire ....................................................... 1-10-19
release ............................................................. 1-10-20
relwait .............................................................. 1-10-21
unacquire .......................................................... 1-10-22
Hints for Using Acquire/Release Instructions ................ 1-10-23
  Task Synchronization ......................................... 1-10-23
  Responding to Loss of a Master Task or Slave Task .... 1-10-26
    Effects of a Lost Master Task or Lost Slave Task .... 1-10-27
    Sample Use of on lost_slave and on lost_master .... 1-10-28
  Removing Master Request From Slave Task Acquire Queue 1-10-32
  Passive and Active Masters and Slaves .................... 1-10-32
Section 11: Interactive Instructions .............................................. 1-11-1
    notify and renotify ..................................................... 1-11-2
    prompt, reprompt, and input .......................................... 1-11-4
    prompt, reprompt, and input$ ........................................ 1-11-6

Section 12: Scaling Instructions .................................................. 1-12-1
    scdp ................................................................ 1-12-3
    schi and sclo ............................................................. 1-12-4
    sclike and unscl ......................................................... 1-12-5

Section 13: Timestamp Functions .................................................. 1-13-1
    getssm .................................................................. 1-13-2
    getssss .................................................................. 1-13-3
    gettime .................................................................... 1-13-4
    gettrend .................................................................... 1-13-5
    gettrend Status Values .............................................. 1-13-7
    nil ........................................................................ 1-13-8
    puttime ..................................................................... 1-13-9
    realtime ..................................................................... 1-13-10
    %r, %rt, and %rd print Instruction Extensions ....................... 1-13-11

Section 14: Password Security ......................................................... 1-14-1
    alv2str$ .................................................................. 1-14-2
    inputp ....................................................................... 1-14-3
    whois ....................................................................... 1-14-5

Section 15: RBL Controller Instructions ........................................... 1-15-1
    bcc$ ....................................................................... 1-15-2
    chflush ..................................................................... 1-15-4
    chwait ....................................................................... 1-15-5
    commclose ................................................................... 1-15-6
    commopen ................................................................... 1-15-7
    commstat ................................................................... 1-15-10
    dectoieee ................................................................... 1-15-11
    getbyte$ ................................................................... 1-15-12
    lrc8$ ....................................................................... 1-15-13
    putbyte ..................................................................... 1-15-14
    putnbyte .................................................................... 1-15-15
    seteot and getnbyte$ ..................................................... 1-15-16

Chapter 2: RBL Controller

Section 1: Introduction ................................................................. 2-1-1
    RBL Controller Hardware .................................................. 2-1-2
    RBL Controller Software ................................................... 2-1-3
Section 2: Configuring the RBL ControlBlock .......................... 2-2-1
RBL Files ......................................................................... 2-1-5
Writing Scripts ................................................................. 2-2-4
RBL Continuous Faceplate Screen ................................... 2-2-2
RBL Monitor Screen .......................................................... 2-2-4
Using Checksums .............................................................. 2-2-6
Summation Checksum Method ........................................... 2-2-6
Longitudinal Redundancy Checksum Method ....................... 2-2-7
Scaling Conversions ........................................................... 2-2-8
Weigh Scale Sample Program ............................................ 2-2-9
RBL Loop Back Sample Program ........................................ 2-2-13

Appendixes

Appendix A: Exception Handling Example .......................... A-1
About the Example ........................................................... A-2
Restarting on Power Up .................................................... A-2
Responding to Hold, Stop, and Abort ................................. A-4
Responding to Print Failure ............................................... A-6
Exception Handling for an Operation ................................. A-7
ControlBlock Output ........................................................ A-15

Appendix B: Timing Backups to Nonvolatile Memory ............... B-1
List of Figures

Figure 1.1.1. RBL Program Hierarchical Structure .................................................. RB:1-1-2
Figure 1.1.2. Phase Format .................................................................................. RB:1-1-4
Figure 1.1.3. Label Format .................................................................................. RB:1-1-6
Figure 1.1.4. Symbol Format .............................................................................. RB:1-1-7
Figure 1.1.5. Status Format ................................................................................ RB:1-1-9
Figure 1.1.6. Sample Use of Tags and Addresses to Access ControlBlock Variables .. RB:1-1-12
Figure 1.1.7. Sample Use of Tags and Addresses to Select ControlBlock Modes .... RB:1-1-13
Figure 1.1.8. Using Rate Limits ......................................................................... RB:1-1-15
Figure 1.2.1. Batch Script Screen ..................................................................... RB:1-2-3
Figure 1.2.2. RBL File Contents Screen .............................................................. RB:1-2-4
Figure 1.2.3. Printing a Script ........................................................................... RB:1-2-6
Figure 1.2.4. Calling up a Script ........................................................................ RB:1-2-7
Figure 1.2.5. Deleting a Script ........................................................................... RB:1-2-8
Figure 1.2.6. Copying a Script .......................................................................... RB:1-2-9
Figure 1.2.7. Copying from the Batch Script Screen .......................................... RB:1-2-12
Figure 1.2.8. Search Functions ......................................................................... RB:1-2-18
Figure 1.2.9. Calling Up Scripts ........................................................................ RB:1-2-23
Figure 1.3.1. Shared and Private Symbol Concept ........................................... RB:1-3-5
Figure 1.3.2. Hex and Binary Format ................................................................. RB:1-3-6
Figure 1.3.3. Alias Format ................................................................................. RB:1-3-8
Figure 1.3.4. Dot Operator Format ................................................................... RB:1-3-9
Figure 1.3.5. Address Scheme for Array Storage Locations .............................. RB:1-3-11
Figure 1.3.6. aliasdim Instruction Format ......................................................... RB:1-3-13
Figure 1.3.7. One-Dimensional and Two-Dimensional Arrays ......................... RB:1-3-15
Figure 1.3.8. stringdim Instruction Format ....................................................... RB:1-3-16
Figure 1.3.9. One-Dimensional and Two-Dimensional vdim Arrays ................ RB:1-3-18
Figure 1.3.10. One-Dimensional and Two-Dimensional vstringdim Arrays ...... RB:1-3-19
Figure 1.3.11. rvdim Instruction Format .............................................................. RB:1-3-20
Figure 1.3.12. rvstringdim Instruction Format .................................................. RB:1-3-21
Figure 1.3.13. Disk Space Requirements for Virtual Array ............................... RB:1-3-22
Figure 1.3.14. Address scheme for array_copy instruction ............................... RB:1-3-26
Figure 1.3.14. Address scheme for array_copy instructions (continued) ............ RB:1-3-27
Figure 1.3.15. array_copy Instruction Format .................................................... RB:1-3-28
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.16</td>
<td>array_init Instruction Format</td>
<td>RB:1-3-29</td>
</tr>
<tr>
<td>1.3.17</td>
<td>file_copy Instruction Format</td>
<td>RB:1-3-30</td>
</tr>
<tr>
<td>1.3.18</td>
<td>file_delete Instruction Format</td>
<td>RB:1-3-31</td>
</tr>
<tr>
<td>1.3.19</td>
<td>file_rename Instruction Format</td>
<td>RB:1-3-32</td>
</tr>
<tr>
<td>1.3.20</td>
<td>varray_file Instruction Format</td>
<td>RB:1-3-33</td>
</tr>
<tr>
<td>1.3.21</td>
<td>Constant Format</td>
<td>RB:1-3-35</td>
</tr>
<tr>
<td>1.3.22</td>
<td>Global Variable Format</td>
<td>RB:1-3-37</td>
</tr>
<tr>
<td>1.3.23</td>
<td>Local Variable Format</td>
<td>RB:1-3-38</td>
</tr>
<tr>
<td>1.3.24</td>
<td>String Variable format</td>
<td>RB:1-3-39</td>
</tr>
<tr>
<td>1.4.1</td>
<td>andbit$ and orbit$ Function Format</td>
<td>RB:1-4-2</td>
</tr>
<tr>
<td>1.4.2</td>
<td>andbit$ and orbit$ Functions</td>
<td>RB:1-4-3</td>
</tr>
<tr>
<td>1.4.3</td>
<td>asc Function Format</td>
<td>RB:1-4-4</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Hex and Bit Representation of Two Character Bytes in User Flags</td>
<td>RB:1-4-5</td>
</tr>
<tr>
<td>1.4.5</td>
<td>flag2byte and byte2flag Format</td>
<td>RB:1-4-6</td>
</tr>
<tr>
<td>1.4.6</td>
<td>Sample Configuration Using byte2flag</td>
<td>RB:1-4-8</td>
</tr>
<tr>
<td>1.4.7</td>
<td>Sample Configuration Using flag2byte$</td>
<td>RB:1-4-9</td>
</tr>
<tr>
<td>1.4.8</td>
<td>chr$ Function Format</td>
<td>RB:1-4-11</td>
</tr>
<tr>
<td>1.4.9</td>
<td>getbit Function Format</td>
<td>RB:1-4-12</td>
</tr>
<tr>
<td>1.4.10</td>
<td>setbit$ Function Format</td>
<td>RB:1-4-13</td>
</tr>
<tr>
<td>1.4.11</td>
<td>str$ Function Format</td>
<td>RB:1-4-14</td>
</tr>
<tr>
<td>1.4.12</td>
<td>strcat$ Function Format</td>
<td>RB:1-4-15</td>
</tr>
<tr>
<td>1.4.13</td>
<td>strcmp Function Format</td>
<td>RB:1-4-16</td>
</tr>
<tr>
<td>1.4.14</td>
<td>strlen Function Format</td>
<td>RB:1-4-17</td>
</tr>
<tr>
<td>1.4.15</td>
<td>swab$ Function Format</td>
<td>RB:1-4-18</td>
</tr>
<tr>
<td>1.4.16</td>
<td>val Function Format</td>
<td>RB:1-4-20</td>
</tr>
<tr>
<td>1.5.1</td>
<td>chain and endchain Formats</td>
<td>RB:1-5-3</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Define alias Value in Child Script and Redefine Value in Parent Script</td>
<td>RB:1-5-4</td>
</tr>
<tr>
<td>1.5.3</td>
<td>Declare alias in Parent Script and Redefine alias Value in Child Script</td>
<td>RB:1-5-5</td>
</tr>
<tr>
<td>1.5.4</td>
<td>Comparison of Shared and Private Symbols in Chained Scripts</td>
<td>RB:1-5-7</td>
</tr>
<tr>
<td>1.5.5</td>
<td>for Instruction Format</td>
<td>RB:1-5-9</td>
</tr>
<tr>
<td>1.5.6</td>
<td>gosub Instruction Format (continued)</td>
<td>RB:1-5-11</td>
</tr>
<tr>
<td>1.5.7</td>
<td>goto Instruction Format</td>
<td>RB:1-5-13</td>
</tr>
<tr>
<td>1.5.8</td>
<td>if Instruction Format</td>
<td>RB:1-5-15</td>
</tr>
<tr>
<td>1.5.9</td>
<td>else Instruction Format</td>
<td>RB:1-5-17</td>
</tr>
<tr>
<td>1.5.10</td>
<td>elseif Instruction Format</td>
<td>RB:1-5-19</td>
</tr>
<tr>
<td>1.5.11</td>
<td>return Instruction Format</td>
<td>RB:1-5-20</td>
</tr>
<tr>
<td>1.5.12</td>
<td>start Instruction Format</td>
<td>RB:1-5-21</td>
</tr>
</tbody>
</table>
Figure 1.5.13. until Instruction Format ....................................................... RB:1-5-23
Figure 1.5.14. while Instruction Format ................................................... RB:1-5-25
Figure 1.6.1. abort Instruction Format ..................................................... RB:1-6-2
Figure 1.6.2. abort_task Instruction Format .............................................. RB:1-6-3
Figure 1.6.3. backup Instruction Format .................................................... RB:1-6-4
Figure 1.6.4. begin_task Instruction Format .............................................. RB:1-6-5
Figure 1.6.5. break Instruction Format ...................................................... RB:1-6-6
Figure 1.6.6. bumpid Instruction Format .................................................... RB:1-6-7
Figure 1.6.7. continue Instruction Format .................................................. RB:1-6-9
Figure 1.6.8. cont_task Instruction Format ............................................... RB:1-6-10
Figure 1.6.9. dishold and enhold Instruction Format ................................... RB:1-6-11
Figure 1.6.10. end Instruction Format ....................................................... RB:1-6-12
Figure 1.6.11. finish Instruction Format .................................................... RB:1-6-13
Figure 1.6.12. halt Instruction Format ........................................................ RB:1-6-14
Figure 1.6.13. halt_task Instruction Format ................................................ RB:1-6-15
Figure 1.6.14. hold Instruction Format ...................................................... RB:1-6-16
Figure 1.6.15. holding Instruction Format .................................................. RB:1-6-17
Figure 1.6.16. kill_task Instruction Format ............................................... RB:1-6-18
Figure 1.6.17. max and min Instruction Format ......................................... RB:1-6-19
Figure 1.6.18. next Instruction Format ...................................................... RB:1-6-20
Figure 1.6.19. nvfree and vfree Instruction Format ..................................... RB:1-6-21
Figure 1.6.20. opremain and phremain Instruction Format ......................... RB:1-6-22
Figure 1.6.21. priority Instruction Format .................................................. RB:1-6-23
Figure 1.6.22. recabort, rechold, and recresume Instruction Format ........... RB:1-6-26
Figure 1.6.23. resume and resume_task Instruction Formats ...................... RB:1-6-28
Figure 1.6.24. retry Instruction Format ..................................................... RB:1-6-29
Figure 1.6.25. sleep Instruction Format .................................................... RB:1-6-30
Figure 1.6.26. ssm and sss Instruction Format .......................................... RB:1-6-32
Figure 1.6.27. started_by Instruction Format ............................................. RB:1-6-33
Figure 1.6.28. stop Instruction Format ..................................................... RB:1-6-36
Figure 1.6.29. stop_task Instruction Format ............................................. RB:1-6-37
Figure 1.6.30. stop_task Instruction Format ............................................. RB:1-6-38
Figure 1.6.31. time Instruction Format ..................................................... RB:1-6-39
Figure 1.7.1. align_on and align_wait Instruction Format ........................ RB:1-7-3
Figure 1.7.2. badunit Instruction Format ................................................... RB:1-7-4
Figure 1.7.3. begin_recipe Instruction Format .......................................... RB:1-7-5
Figure 1.7.4. begin_recipe Instruction Demonstration of Function ............. RB:1-7-6
Figure 1.10.8. on lost_master Instruction Format .................................................. RB:1-10-15
Figure 1.10.9. on lost_slave Instruction Format ...................................................... RB:1-10-18
Figure 1.10.10. on unacquire Instruction Format .................................................... RB:1-10-19
Figure 1.10.11. release Instruction Format ............................................................. RB:1-10-20
Figure 1.10.12. relwait Instruction Format ............................................................. RB:1-10-21
Figure 1.10.13. unacquire Instruction Format .......................................................... RB:1-10-22
Figure 1.10.14. Timing Problems Associated with Terminating a Slave Request ........ RB:1-10-24
Figure 1.10.15. Verify Synchronization and Continue Processing ......................... RB:1-10-24
Figure 1.10.16. Verify Synchronization and Disconnect Master and Slave Tasks ........ RB:1-10-25
Figure 1.10.17. Master Task and Acquire Task Flow Chart .................................. RB:1-10-29
Figure 1.11.1. notify and renotify Instruction Format .......................................... RB:1-11-3
Figure 1.11.2. prompt, reprompt, and input Formats ......................................... RB:1-11-5
Figure 1.11.3. prompt, reprompt, and input$ Formats ......................................... RB:1-11-7
Figure 1.12.1. scdp Instruction Format ................................................................. RB:1-12-3
Figure 1.12.2. schi and sclo Instruction Formats ............................................... RB:1-12-4
Figure 1.12.3. sclike and unsc Instruction Formats ........................................... RB:1-12-6
Figure 1.13.1. getssm Function Format ................................................................. RB:1-13-2
Figure 1.13.2. getsss Function Format ................................................................. RB:1-13-3
Figure 1.13.3. gettime Function Format .............................................................. RB:1-13-4
Figure 1.13.4. gettrend Function Format ............................................................ RB:1-13-6
Figure 1.13.5. Configuring a Tracking Array ....................................................... RB:1-13-7
Figure 1.13.6. nil Function Format ..................................................................... RB:1-13-8
Figure 1.13.7. puttime Function Format ............................................................. RB:1-13-9
Figure 1.13.8. realtime Function Format ............................................................. RB:1-13-10
Figure 1.13.9. %r, %rt, and %rd print Instruction Extensions ................................... RB:1-13-11
Figure 1.14.1. alv2str$ Instruction Format ........................................................ RB:1-14-2
Figure 1.14.2. Inputp Instruction Format ............................................................ RB:1-14-3
Figure 1.14.3. whois Instruction Format ............................................................ RB:1-14-6
Figure 1.14.4. whois Demonstration of Function ............................................... RB:1-14-7
**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1.1</td>
<td>Instruction Examples</td>
<td>RB:1-1-8</td>
</tr>
<tr>
<td>Table 1.1.2</td>
<td>ControlBlock Modes Status</td>
<td>RB:1-1-13</td>
</tr>
<tr>
<td>Table 1.1.3</td>
<td>RBL Parameter Limits</td>
<td>RB:1-1-16</td>
</tr>
<tr>
<td>Table 1.1.4</td>
<td>Mathematical Function Options</td>
<td>RB:1-1-18</td>
</tr>
<tr>
<td>Table 1.2.1</td>
<td>RBL File Contents Screen Fields</td>
<td>RB:1-2-5</td>
</tr>
<tr>
<td>Table 1.2.2</td>
<td>Default Script Editing Colors</td>
<td>RB:1-2-21</td>
</tr>
<tr>
<td>Table 1.4.1</td>
<td>Status Values for flag2byte$ and byte2flag Functions</td>
<td>RB:1-4-10</td>
</tr>
<tr>
<td>Table 1.5.1</td>
<td>while Case Formats</td>
<td>RB:1-5-26</td>
</tr>
<tr>
<td>Table 1.6.1</td>
<td>status_task status values</td>
<td>RB:1-6-34</td>
</tr>
<tr>
<td>Table 1.6.2</td>
<td>time values</td>
<td>RB:1-6-40</td>
</tr>
<tr>
<td>Table 1.7.1</td>
<td>begin_recipe status values</td>
<td>RB:1-7-7</td>
</tr>
<tr>
<td>Table 1.7.2</td>
<td>fetch status values</td>
<td>RB:1-7-14</td>
</tr>
<tr>
<td>Table 1.7.3</td>
<td>Status Values for run_recipe Instruction</td>
<td>RB:1-7-30</td>
</tr>
<tr>
<td>Table 1.8.1</td>
<td>Alarm on Trap Format (Batch Only)</td>
<td>RB:1-8-7</td>
</tr>
<tr>
<td>Table 1.8.2</td>
<td>on Instruction Format for Batch and RBLC System Traps</td>
<td>RB:1-8-9</td>
</tr>
<tr>
<td>Table 1.8.3</td>
<td>on Instruction Format for Batch Only System Traps</td>
<td>RB:1-8-10</td>
</tr>
<tr>
<td>Table 1.9.1</td>
<td>Alarm Traps</td>
<td>RB:1-9-3</td>
</tr>
<tr>
<td>Table 1.9.2</td>
<td>Status Values for open and close Instructions</td>
<td>RB:1-9-9</td>
</tr>
<tr>
<td>Table 1.9.3</td>
<td>Format Control Options</td>
<td>RB:1-9-14</td>
</tr>
<tr>
<td>Table 1.9.4</td>
<td>String Formats (Fixed)</td>
<td>RB:1-9-15</td>
</tr>
<tr>
<td>Table 1.9.5</td>
<td>String Format Conversion Specifications</td>
<td>RB:1-9-16</td>
</tr>
<tr>
<td>Table 1.9.6</td>
<td>Format Instructions</td>
<td>RB:1-9-17</td>
</tr>
<tr>
<td>Table 1.9.7</td>
<td>String Formats (Fixed)</td>
<td>RB:1-9-18</td>
</tr>
<tr>
<td>Table 1.10.1</td>
<td>Status Values for acquire/release Instructions</td>
<td>RB:1-10-5</td>
</tr>
<tr>
<td>Table 1.12.1</td>
<td>Scaling for Field Devices</td>
<td>RB:1-12-2</td>
</tr>
<tr>
<td>Table 1.13.1</td>
<td>Status Values for gettrend function</td>
<td>RB:1-13-7</td>
</tr>
<tr>
<td>Table 1.15.1</td>
<td>commopen Instruction Parameter Options</td>
<td>RB:1-15-8</td>
</tr>
<tr>
<td>Table 1.15.2</td>
<td>Status Variables</td>
<td>RB:1-15-9</td>
</tr>
<tr>
<td>Table 2.2.1</td>
<td>RBL Continuous Faceplate Screen Fields</td>
<td>RB:2-2-3</td>
</tr>
</tbody>
</table>
Chapter 1: Rosemount Basic Language (RBL)

Section 1: RBL Concept ......................................................... 1-1-1

- RBL Script Structures .................................................... 1-1-1
- Scripts ............................................................................. 1-1-3
- Phases ............................................................................. 1-1-4
- Labels .............................................................................. 1-1-5
- Symbols ............................................................................ 1-1-7
- Instructions ...................................................................... 1-1-8
  - Instruction Status ......................................................... 1-1-9
  - Instruction Syntax ........................................................ 1-1-9
  - Nesting Stack .............................................................. 1-1-10
- Comments ........................................................................ 1-1-11
  - Comment Examples ...................................................... 1-1-11
- Disappearing Comments ................................................. 1-1-11
- Tags and Addresses ......................................................... 1-1-12
  - ControlBlock Variable Values ..................................... 1-1-12
  - ControlBlock Modes ................................................... 1-1-13
- Instructions That Retry .................................................. 1-1-14
- Script and ControlBlock Synchronization ......................... 1-1-15
- RBL Parameter Limits ..................................................... 1-1-16
- Mathematical Operators ................................................ 1-1-18

Section 2: Script Management .............................................. 1-2-1

- Creating an RBL Files Folder and RBL File ...................... 1-2-2
  - Batch Script Screen ..................................................... 1-2-3
  - RBL File Contents screen .......................................... 1-2-4
- Maintaining Script Files .................................................. 1-2-6
  - Printing a Script .......................................................... 1-2-6
  - Calling Up a Script ...................................................... 1-2-7
  - Deleting a Script ......................................................... 1-2-8
  - Copying a Script ........................................................ 1-2-9
- Writing and Editing Scripts .............................................. 1-2-10
  - Entering Text ............................................................... 1-2-11
COPYING A SCRIPT OR SCRIPT FILE FROM THE BATCH SCRIPT SCREEN
DELETING AND INSERTING TEXT
PARSING AND WRITING TEXT TO DISK
READING AND MOVING SCRIPTS
USING SEARCH AND REPLACE FUNCTIONS
SCRIPT EDITING COLORS
CALLING UP A SCRIPT

SECTION 3: NUMERIC AND STRING SYMBOLS

Shared and Private symbols
Batch Recipe Use of Shared and Private Symbols
Batch Run Screen Use of Shared and Private Symbols
Shared and Private Symbol Concept
Defining Shared Symbols
Hex and Binary Values
Aliases
Using Aliases
Dot Operators
Arrays
aliasdim Array
dim Array
stringdim Array
Virtual Arrays (Batch Only)
vdim Array
vstringdim Array
rvdim and rvstringdim
Computing Disk Space for Virtual Arrays
Array Functions
array_copy and array_init
Performance Improvements
Array Copy and Initialize Concept
array_copy Format
array_init Format
file_copy
file_delete
file_rename
varray_file
Constants
Global Variables
Local Variables
String Variables
String Characteristics
Hints for Using Symbols
Changing Variables
Making a Script Execute Sooner
Conserving Volatile Memory
Comparing Floating Point Values
### Section 4: RBL String Functions

- `andbit$` and `orbit$` ................................................................. 1-4-2
- `asc` .................................................................................. 1-4-4
- `byte2flag` and `flag2byte$` .................................................. 1-4-5
  - Using `byte2flag` and `flag2byte$` ........................................ 1-4-7
  - `byte2flag` and `flag2byte$` Status Values ...................... 1-4-10
- `chr$` ................................................................................. 1-4-11
- `getbit` .............................................................................. 1-4-12
- `setbit$` ............................................................................ 1-4-13
- `str$` ................................................................................. 1-4-14
- `strcat$` ........................................................................... 1-4-15
- `strcmp` ............................................................................. 1-4-16
- `strlen` .............................................................................. 1-4-17
- `swab$` ............................................................................. 1-4-18
- `val` ................................................................................... 1-4-19

### Section 5: Branching and Conditional Instructions

- `chain` and `endchain` ............................................................ 1-5-2
  - Using Shared Aliases and Constants in Chained Scripts .......... 1-5-4
  - Using Private Aliases and Constants in Chained Scripts .......... 1-5-6
- `for` ...................................................................................... 1-5-8
- `gosub` ............................................................................... 1-5-10
- `goto` .................................................................................. 1-5-12
- `if`, `elseif`, and `else` ......................................................... 1-5-14
  - `if` .................................................................................. 1-5-14
  - `else` ............................................................................... 1-5-16
  - `elseif` ........................................................................... 1-5-18
- `return` ............................................................................... 1-5-20
- `start` .................................................................................. 1-5-21
- `until` ................................................................................... 1-5-22
- `while` .................................................................................. 1-5-24
  - Special While Cases .......................................................... 1-5-26

### Section 6: Batch Management Instructions

- `abort` ................................................................................... 1-6-2
- `abort_task` ........................................................................ 1-6-3
- `backup` ............................................................................. 1-6-4
- `begin_task` ........................................................................ 1-6-5
- `break` .................................................................................. 1-6-6
- `bumpid` ............................................................................. 1-6-7
- `continue` ............................................................................ 1-6-8
- `cont_task` .......................................................................... 1-6-10
- `dishold` and `enhold` .......................................................... 1-6-11
- `end` .................................................................................... 1-6-12
- `finish` ................................................................................. 1-6-13
- `halt` ................................................................................... 1-6-14
- `halt_task` .......................................................................... 1-6-15
hold ........................................................................... 1-6-16
holding ...................................................................... 1-6-17
kill_task ..................................................................... 1-6-18
max and min ................................................................. 1-6-19
next ........................................................................... 1-6-20
nvfree and vfree .......................................................... 1-6-21
opremain and phremain ............................................... 1-6-22
priority ...................................................................... 1-6-23
Priority in Relation to Other System Activities .............. 1-6-24
Setting the Batch Task Priority ..................................... 1-6-25
recabort, rechold, and recresume ................................. 1-6-26
resume and resume_task ............................................. 1-6-27
retry ........................................................................... 1-6-29
sleep .......................................................................... 1-6-30
sleep Instruction Format .............................................. 1-6-30
Hints for Using a sleep Instruction ................................ 1-6-31
ssm and sss .................................................................. 1-6-32
status_task .................................................................. 1-6-33
status_task format ....................................................... 1-6-33
status_task Status Values ............................................ 1-6-34
stop ........................................................................... 1-6-36
stop_task .................................................................... 1-6-37
sync .......................................................................... 1-6-38
time ........................................................................... 1-6-39

Section 7: Recipe Management Instructions ....................... 1-7-1
align_on and align_wait .............................................. 1-7-2
badunit ....................................................................... 1-7-4
begin_recipe .................................................................. 1-7-5
display and undisplay ................................................. 1-7-8
exit ............................................................................ 1-7-10
fetch and fetch$ .......................................................... 1-7-12
filerr ......................................................................... 1-7-15
getmaterial and putmaterial ......................................... 1-7-17
getmatlim .................................................................... 1-7-18
mapunit ....................................................................... 1-7-19
materialdesc ............................................................... 1-7-21
materialunits .............................................................. 1-7-22
on unowned_unit ......................................................... 1-7-24
ownunit and disownunit ................................................. 1-7-25
recipe_local .................................................................. 1-7-27
run_recipe .................................................................... 1-7-28
started_by .................................................................... 1-7-31
usegraphic .................................................................... 1-7-33
wait_bid and send_bid ................................................... 1-7-35

Section 8: On Trap Instructions ........................................ 1-8-1
On Trap Overview ......................................................... 1-8-2
On Trap Execution Priority .............................................. 1-8-3
  More than One On Trap per ControlBlock Address .................. 1-8-3
  Execution Order in Response to Multiple Alarms or System Conditions ..... 1-8-5
Alarm Traps ....................................................................... 1-8-7
  Alarm Traps with Priority Designation .................................. 1-8-7
  on Alarm Traps without Priority Designation .......................... 1-8-8
System Traps ..................................................................... 1-8-9
clear_alarm_queue ............................................................... 1-8-12

Section 9: Alarm, Log, and Report Instructions ......................... 1-9-1
alarm, almsg, and clear ......................................................... 1-9-2
event ........................................................................ 1-9-4
flog ............................................................................ 1-9-5
open and close ................................................................. 1-9-7
  open and close Status Values ......................................... 1-9-9
print and print$ .............................................................. 1-9-10
  print ........................................................................ 1-9-10
  print$ .................................................................... 1-9-12
Print Options .................................................................... 1-9-13
  Format Control Options .............................................. 1-9-14
  Constant Conversion Options ....................................... 1-9-15
  Decimal Conversion Options ........................................ 1-9-16
  Variable Conversion Options ....................................... 1-9-17
  System String Options ................................................. 1-9-18
quietlog and loudlog .......................................................... 1-9-19

Section 10: Acquire/Release Instructions ................................. 1-10-1
Overview ..................................................................... 1-10-2
Example of Acquiring a Task ............................................. 1-10-3
Acquire/Release Concept .................................................. 1-10-4
Acquire/Release Status Values .......................................... 1-10-5
acquire and acqfirst ......................................................... 1-10-7
acqwait ...................................................................... 1-10-9
dequeue ..................................................................... 1-10-10
masterof ...................................................................... 1-10-12
on acquire_err .............................................................. 1-10-13
on lost_master ................................................................ 1-10-14
on lost_slave .................................................................. 1-10-16
  Conditions for Execution ............................................ 1-10-17
on unacquire .................................................................. 1-10-19
release ........................................................................ 1-10-20
relwait ......................................................................... 1-10-21
unacquire ...................................................................... 1-10-22
Hints for Using Acquire/Release Instructions .......................... 1-10-23
  Task Synchronization .................................................. 1-10-23
  Responding to Loss of a Master Task or Slave Task ............... 1-10-26
  Effects of a Lost Master Task or Lost Slave Task ................. 1-10-27
Sample Use of on lost_slave and on lost_master .......................... 1-10-28
Removing Master Request From Slave Task Acquire Queue .......... 1-10-32
Passive and Active Masters and Slaves ..................................... 1-10-32

Section 11: Interactive Instructions .......................................... 1-11-1
notify and renotify ............................................................. 1-11-2
prompt, reprompt, and input ................................................. 1-11-4
prompt, reprompt, and input$ .............................................. 1-11-6

Section 12: Scaling Instructions .............................................. 1-12-1
scdp .............................................................................. 1-12-3
schl and sclo ................................................................. 1-12-4
sclike and unsc .............................................................. 1-12-5

Section 13: Timestamp Functions ............................................. 1-13-1
getssm ........................................................................... 1-13-2
getssss ............................................................................. 1-13-3
gettime ............................................................................ 1-13-4
gettrend ........................................................................ 1-13-5
   gettrend Status Values .................................................. 1-13-6
nil ................................................................................. 1-13-8
puttime ............................................................................ 1-13-9
realtime ........................................................................ 1-13-10
%r, %rt, and %rd print Instruction Extensions ............................ 1-13-11

Section 14: Password Security ................................................. 1-14-1
alv2str$ ....................................................................... 1-14-2
inputp ............................................................................. 1-14-3
whois ............................................................................. 1-14-5

Section 15: RBL Controller Instructions .................................... 1-15-1
bcc$ .............................................................................. 1-15-2
chiflush ......................................................................... 1-15-4
chwait ............................................................................... 1-15-5
commclose ...................................................................... 1-15-6
commopen ...................................................................... 1-15-7
commstat ......................................................................... 1-15-10
dectoieee ....................................................................... 1-15-11
getbyte$ ............................................................................. 1-15-12
lrc8$ ............................................................................. 1-15-13
putbyte ............................................................................. 1-15-14
putnbyte ........................................................................... 1-15-15
seteot and getnbyte$ ......................................................... 1-15-16
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1</td>
<td>RBL Program Hierarchical Structure</td>
<td>1-1-2</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Phase Format</td>
<td>1-1-4</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Label Format</td>
<td>1-1-6</td>
</tr>
<tr>
<td>1.1.4</td>
<td>Symbol Format</td>
<td>1-1-7</td>
</tr>
<tr>
<td>1.1.5</td>
<td>Status Format</td>
<td>1-1-9</td>
</tr>
<tr>
<td>1.1.6</td>
<td>Sample Use of Tags and Addresses to Access ControlBlock Variables</td>
<td>1-1-12</td>
</tr>
<tr>
<td>1.1.7</td>
<td>Sample Use of Tags and Addresses to Select ControlBlock Modes</td>
<td>1-1-13</td>
</tr>
<tr>
<td>1.1.8</td>
<td>Using Rate Limits</td>
<td>1-1-15</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Batch Script Screen</td>
<td>1-2-3</td>
</tr>
<tr>
<td>1.2.2</td>
<td>RBL File Contents Screen</td>
<td>1-2-4</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Printing a Script</td>
<td>1-2-6</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Calling up a Script</td>
<td>1-2-7</td>
</tr>
<tr>
<td>1.2.5</td>
<td>Deleting a Script</td>
<td>1-2-8</td>
</tr>
<tr>
<td>1.2.6</td>
<td>Copying a Script</td>
<td>1-2-9</td>
</tr>
<tr>
<td>1.2.7</td>
<td>Copying from the Batch Script Screen</td>
<td>1-2-12</td>
</tr>
<tr>
<td>1.2.8</td>
<td>Search Functions</td>
<td>1-2-18</td>
</tr>
<tr>
<td>1.2.9</td>
<td>Calling Up Scripts</td>
<td>1-2-23</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Shared and Private Symbol Concept</td>
<td>1-3-5</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Hex and Binary Format</td>
<td>1-3-6</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Alias Format</td>
<td>1-3-8</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Dot Operator Format</td>
<td>1-3-9</td>
</tr>
<tr>
<td>1.3.5</td>
<td>Address Scheme for Array Storage Locations</td>
<td>1-3-11</td>
</tr>
<tr>
<td>1.3.6</td>
<td>aliasdim Instruction Format</td>
<td>1-3-13</td>
</tr>
<tr>
<td>1.3.7</td>
<td>One-Dimensional and Two-Dimensional Arrays</td>
<td>1-3-15</td>
</tr>
<tr>
<td>1.3.8</td>
<td>stringdim Instruction Format</td>
<td>1-3-16</td>
</tr>
<tr>
<td>1.3.9</td>
<td>One-Dimensional and Two-Dimensional vdim Arrays</td>
<td>1-3-18</td>
</tr>
<tr>
<td>1.3.10</td>
<td>One-Dimensional and Two-Dimensional vstringdim Arrays</td>
<td>1-3-19</td>
</tr>
<tr>
<td>1.3.11</td>
<td>rvdim Instruction Format</td>
<td>1-3-20</td>
</tr>
<tr>
<td>1.3.12</td>
<td>rstringdim Instruction Format</td>
<td>1-3-21</td>
</tr>
<tr>
<td>1.3.13</td>
<td>Disk Space Requirements for Virtual Array</td>
<td>1-3-22</td>
</tr>
<tr>
<td>1.3.14</td>
<td>Address scheme for array_copy instruction</td>
<td>1-3-26</td>
</tr>
<tr>
<td>1.3.14</td>
<td>Address scheme for array_copy instructions (continued)</td>
<td>1-3-27</td>
</tr>
</tbody>
</table>
1.3.15 array_copy Instruction Format ........................................... 1-3-28
1.3.16 array_init Instruction Format ........................................... 1-3-29
1.3.17 file_copy Instruction Format ........................................... 1-3-30
1.3.18 file_delete Instruction Format ........................................... 1-3-31
1.3.19 file_rename Instruction Format ........................................... 1-3-32
1.3.20 varray_file Instruction Format ........................................... 1-3-33
1.3.21 Constant Format ........................................................... 1-3-35
1.3.22 Global Variable Format .................................................... 1-3-37
1.3.23 Local Variable Format ....................................................... 1-3-38
1.3.24 String Variable format ....................................................... 1-3-39
1.4.1 andbit$ and orbit$ Function Format .................................... 1-4-2
1.4.2 andbit$ and orbit$ Functions ............................................. 1-4-3
1.4.3 asc Function Format ......................................................... 1-4-4
1.4.4 Hex and Bit Representation of Two Character Bytes in User Flags . 1-4-5
1.4.5 flag2byte and byte2flag Format ......................................... 1-4-6
1.4.6 Sample Configuration Using byte2flag ................................ 1-4-8
1.4.7 Sample Configuration Using flag2byte$ ................................ 1-4-9
1.4.8 chr$ Function Format ......................................................... 1-4-11
1.4.9 getbit Function Format ....................................................... 1-4-12
1.4.10 setbit$ Function Format ..................................................... 1-4-13
1.4.11 str$ Function Format ......................................................... 1-4-14
1.4.12 strcat$ Function Format ..................................................... 1-4-15
1.4.13 strcmp Function Format ..................................................... 1-4-16
1.4.14 strlen Function Format ....................................................... 1-4-17
1.4.15 swab$ Function Format ....................................................... 1-4-18
1.4.16 val Function Format ......................................................... 1-4-20
1.5.1 chain and endchain Formats ................................................ 1-5-3
1.5.2 Define alias Value in Child Script and Redefine Value in Parent Script 1-5-4
1.5.3 Declare alias in Parent Script and Redefine alias Value in Child Script 1-5-5
1.5.4 Comparison of Shared and Private Symbols in Chained Scripts ...... 1-5-7
1.5.5 for Instruction Format ......................................................... 1-5-9
1.5.6 gosub Instruction Format (continued) .................................... 1-5-11
1.5.7 goto Instruction Format ....................................................... 1-5-13
1.5.8 if Instruction Format ......................................................... 1-5-15
1.5.9 else Instruction Format ....................................................... 1-5-17
1.5.10 elseif Instruction Format ................................................... 1-5-19
1.5.11 return Instruction Format ................................................... 1-5-20
1.5.12 start Instruction Format ........................................... 1-5-21
1.5.13 until Instruction Format .......................................... 1-5-23
1.5.14 while Instruction Format ......................................... 1-5-25
1.6.1 abort Instruction Format .......................................... 1-6-2
1.6.2 abort_task Instruction Format ................................. 1-6-3
1.6.3 backup Instruction Format ........................................ 1-6-4
1.6.4 begin_task Instruction Format ................................. 1-6-5
1.6.5 break Instruction Format ......................................... 1-6-6
1.6.6 bumpid Instruction Format ....................................... 1-6-7
1.6.7 continue Instruction Format ...................................... 1-6-9
1.6.8 cont_task Instruction Format ................................... 1-6-10
1.6.9 dishold and enhold Instruction Format ....................... 1-6-11
1.6.10 end Instruction Format .......................................... 1-6-12
1.6.11 finish Instruction Format ....................................... 1-6-13
1.6.12 halt Instruction Format ......................................... 1-6-14
1.6.13 halt_task Instruction Format ................................ 1-6-15
1.6.14 hold Instruction Format ......................................... 1-6-16
1.6.15 holding Instruction Format .................................... 1-6-17
1.6.16 kill_task Instruction Format .................................. 1-6-18
1.6.17 max and min Instruction Format .............................. 1-6-19
1.6.18 next Instruction Format ........................................... 1-6-20
1.6.19 nvfree and vfree Instruction Format ......................... 1-6-21
1.6.20 opremain and phremain Instruction Format ................. 1-6-22
1.6.21 priority Instruction Format .................................... 1-6-23
1.6.22 recabort, rechold, and recresume Instruction Format .... 1-6-26
1.6.23 resume and resume_task Instruction Formats ............... 1-6-28
1.6.24 retry Instruction Format ........................................ 1-6-29
1.6.25 sleep Instruction Format ........................................ 1-6-30
1.6.26 ssm and sss Instruction Format .............................. 1-6-32
1.6.27 started_by Instruction Format ................................ 1-6-33
1.6.28 stop Instruction Format ......................................... 1-6-36
1.6.29 stop_task Instruction Format ................................ 1-6-37
1.6.30 stop_task Instruction Format ................................ 1-6-38
1.6.31 time Instruction Format ......................................... 1-6-39
1.7.1 align_on and align_wait Instruction Format ............... 1-7-3
1.7.2 badunit Instruction Format .................................... 1-7-4
1.7.3 begin_recipe Instruction Format ............................. 1-7-5
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7.4</td>
<td>begin_recipe Instruction Demonstration of Function</td>
<td>1-7-6</td>
</tr>
<tr>
<td>1.7.5</td>
<td>display and undisplay Instruction Format</td>
<td>1-7-9</td>
</tr>
<tr>
<td>1.7.6</td>
<td>exit Instruction Format</td>
<td>1-7-10</td>
</tr>
<tr>
<td>1.7.7</td>
<td>exit Instruction Format</td>
<td>1-7-11</td>
</tr>
<tr>
<td>1.7.8</td>
<td>fetch Instruction Format</td>
<td>1-7-13</td>
</tr>
<tr>
<td>1.7.9</td>
<td>filerr Instruction Format</td>
<td>1-7-16</td>
</tr>
<tr>
<td>1.7.10</td>
<td>getmaterial and putmaterial Instruction Format</td>
<td>1-7-17</td>
</tr>
<tr>
<td>1.7.11</td>
<td>getmatlim Instruction Format</td>
<td>1-7-18</td>
</tr>
<tr>
<td>1.7.12</td>
<td>mapunit Instruction Format</td>
<td>1-7-20</td>
</tr>
<tr>
<td>1.7.13</td>
<td>materialdesc Instruction Format</td>
<td>1-7-21</td>
</tr>
<tr>
<td>1.7.14</td>
<td>materialunits Instruction Format</td>
<td>1-7-23</td>
</tr>
<tr>
<td>1.7.15</td>
<td>on unowned_unit Instruction Format</td>
<td>1-7-24</td>
</tr>
<tr>
<td>1.7.16</td>
<td>ownunit and disownunit Instruction Format</td>
<td>1-7-26</td>
</tr>
<tr>
<td>1.7.17</td>
<td>recipe_local Instruction Format</td>
<td>1-7-27</td>
</tr>
<tr>
<td>1.7.18</td>
<td>run_recipe Instruction Format</td>
<td>1-7-28</td>
</tr>
<tr>
<td>1.7.19</td>
<td>begin_recipe Instruction Demonstration of Function</td>
<td>1-7-29</td>
</tr>
<tr>
<td>1.7.20</td>
<td>started_by Instruction Format</td>
<td>1-7-32</td>
</tr>
<tr>
<td>1.7.21</td>
<td>usegraphic Instruction Format</td>
<td>1-7-34</td>
</tr>
<tr>
<td>1.7.22</td>
<td>wait_bid and send_bid Instruction Format</td>
<td>1-7-36</td>
</tr>
<tr>
<td>1.8.1</td>
<td>More than One on Trap per Control Block Address</td>
<td>1-8-4</td>
</tr>
<tr>
<td>1.8.2</td>
<td>Execution Order for on Traps in Response to Multiple Alarms</td>
<td>1-8-5</td>
</tr>
<tr>
<td>1.8.3</td>
<td>clear_alarm_queue Instruction Format</td>
<td>1-8-12</td>
</tr>
<tr>
<td>1.9.1</td>
<td>alarm Instruction Format</td>
<td>1-9-2</td>
</tr>
<tr>
<td>1.9.2</td>
<td>event Function Format</td>
<td>1-9-4</td>
</tr>
<tr>
<td>1.9.3</td>
<td>flog Instruction Format</td>
<td>1-9-6</td>
</tr>
<tr>
<td>1.9.4</td>
<td>open and close Instruction Format (continued)</td>
<td>1-9-9</td>
</tr>
<tr>
<td>1.9.5</td>
<td>print Instruction Format</td>
<td>1-9-11</td>
</tr>
<tr>
<td>1.9.6</td>
<td>print$ Instruction Format</td>
<td>1-9-12</td>
</tr>
<tr>
<td>1.9.7</td>
<td>Characteristics of Print Options</td>
<td>1-9-13</td>
</tr>
<tr>
<td>1.9.8</td>
<td>quietlog and loudlog Instruction Format</td>
<td>1-9-20</td>
</tr>
<tr>
<td>1.10.1</td>
<td>Performing Multiple Batch Tasks</td>
<td>1-10-3</td>
</tr>
<tr>
<td>1.10.2</td>
<td>Batch acquire/release Instruction Concept</td>
<td>1-10-4</td>
</tr>
<tr>
<td>1.10.3</td>
<td>acquire and acqfirst Instruction Format (continued)</td>
<td>1-10-8</td>
</tr>
<tr>
<td>1.10.4</td>
<td>acqwait Instruction Format</td>
<td>1-10-9</td>
</tr>
<tr>
<td>1.10.5</td>
<td>dequeue Instruction Format</td>
<td>1-10-11</td>
</tr>
<tr>
<td>1.10.6</td>
<td>masterof Instruction Format</td>
<td>1-10-12</td>
</tr>
<tr>
<td>1.10.7</td>
<td>on acquire_err Instruction Format</td>
<td>1-10-13</td>
</tr>
</tbody>
</table>
1.10.8 on lost_master Instruction Format ........................................ 1-10-15
1.10.9 on lost_slave Instruction Format ........................................... 1-10-18
1.10.10 on unacquire Instruction Format .......................................... 1-10-19
1.10.11 release Instruction Format .................................................. 1-10-20
1.10.12 relwait Instruction Format .................................................. 1-10-21
1.10.13 unacquire Instruction Format ............................................. 1-10-22
1.10.14 Timing Problems Associated with Terminating a Slave Request .... 1-10-24
1.10.15 Verify Synchronization and Continue Processing ....................... 1-10-24
1.10.16 Verify Synchronization and Disconnect Master and Slave Tasks ...... 1-10-25
1.10.17 Master Task and Acquire Task Flow Chart ................................ 1-10-29
1.11.1 notify and renotify Instruction Format .................................... 1-11-3
1.11.2 prompt, reprompt, and input Formats ..................................... 1-11-5
1.11.3 prompt, reprompt, and input$ Formats ................................... 1-11-7
1.12.1 scdp Instruction Format ...................................................... 1-12-3
1.12.2 schi and sclo Instruction Formats ........................................ 1-12-4
1.12.3 sclike and unsc Instruction Formats ....................................... 1-12-6
1.13.1 getssm Function Format ....................................................... 1-13-2
1.13.2 getss$ Function Format ...................................................... 1-13-3
1.13.3 gettime Function Format .................................................... 1-13-4
1.13.4 gettrend Function Format .................................................. 1-13-6
1.13.5 Configuring a Tracking Array .............................................. 1-13-7
1.13.6 nil Function Format ........................................................... 1-13-8
1.13.7 puttime Function Format .................................................... 1-13-9
1.13.8 realtime Function Format ................................................... 1-13-10
1.13.9 \%r, \%rt, and \%rd print Instruction Extensions ............................. 1-13-11
1.14.1 alv2str$ Instruction Format ................................................. 1-14-2
1.14.2 Inputp Instruction Format .................................................... 1-14-3
1.14.2 inputp Instruction Format .................................................... 1-14-4
1.14.3 whois Instruction Format .................................................... 1-14-6
1.14.4 whois Demonstration of Function .......................................... 1-14-7
1.15.1 bcc$ Instruction Format ...................................................... 1-15-3
1.15.2 chflush Instruction Format .................................................. 1-15-4
1.15.3 chwait Instruction Format ................................................... 1-15-5
1.15.4 commclose Instruction Format ............................................. 1-15-6
1.15.5 commopen Instruction Format .............................................. 1-15-7
1.15.6 commstat Instruction Format .............................................. 1-15-10
1.15.7 dectoieee Instruction Format ............................................... 1-15-11
1.15.8  getbyte$ Instruction Format ........................................ 1-15-12
1.15.9  lrc$ Instruction Format ............................................. 1-15-13
1.15.10 putbyte Instruction Format ........................................... 1-15-14
1.15.11 putnbyte Instruction Format ......................................... 1-15-15
1.15.12 seteot and getnbyte Instruction Formats ........................... 1-15-17
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1</td>
<td>Instruction Examples</td>
<td>1-1-8</td>
</tr>
<tr>
<td>1.1.2</td>
<td>ControlBlock Modes Status</td>
<td>1-1-13</td>
</tr>
<tr>
<td>1.1.3</td>
<td>RBL Parameter Limits</td>
<td>1-1-16</td>
</tr>
<tr>
<td>1.1.4</td>
<td>Mathematical Function Options</td>
<td>1-1-18</td>
</tr>
<tr>
<td>1.2.1</td>
<td>RBL File Contents Screen Fields</td>
<td>1-2-5</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Default Script Editing Colors</td>
<td>1-2-21</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Status Values for flag2byte$ and byte2flag Functions</td>
<td>1-4-10</td>
</tr>
<tr>
<td>1.5.1</td>
<td>while Case Formats</td>
<td>1-5-26</td>
</tr>
<tr>
<td>1.6.1</td>
<td>status_task status values</td>
<td>1-6-34</td>
</tr>
<tr>
<td>1.6.2</td>
<td>time values</td>
<td>1-6-40</td>
</tr>
<tr>
<td>1.7.1</td>
<td>begin_recipe status values</td>
<td>1-7-7</td>
</tr>
<tr>
<td>1.7.2</td>
<td>fetch status values</td>
<td>1-7-14</td>
</tr>
<tr>
<td>1.7.3</td>
<td>Status Values for run_recipe Instruction</td>
<td>1-7-30</td>
</tr>
<tr>
<td>1.8.1</td>
<td>Alarm on Trap Format (Batch Only)</td>
<td>1-8-7</td>
</tr>
<tr>
<td>1.8.2</td>
<td>on Instruction Format for Batch and RBLC System Traps</td>
<td>1-8-9</td>
</tr>
<tr>
<td>1.8.3</td>
<td>on Instruction Format for Batch Only System Traps</td>
<td>1-8-10</td>
</tr>
<tr>
<td>1.9.1</td>
<td>Alarm Traps</td>
<td>1-9-3</td>
</tr>
<tr>
<td>1.9.2</td>
<td>Status Values for open and close Instructions</td>
<td>1-9-9</td>
</tr>
<tr>
<td>1.9.3</td>
<td>Format Control Options</td>
<td>1-9-14</td>
</tr>
<tr>
<td>1.9.4</td>
<td>String Formats (Fixed)</td>
<td>1-9-15</td>
</tr>
<tr>
<td>1.9.5</td>
<td>String Format Conversion Specifications</td>
<td>1-9-16</td>
</tr>
<tr>
<td>1.9.6</td>
<td>Format Instructions</td>
<td>1-9-17</td>
</tr>
<tr>
<td>1.9.7</td>
<td>String Formats (Fixed)</td>
<td>1-9-18</td>
</tr>
<tr>
<td>1.10.1</td>
<td>Status Values for acquire/release Instructions</td>
<td>1-10-5</td>
</tr>
<tr>
<td>1.12.1</td>
<td>Scaling for Field Devices</td>
<td>1-12-2</td>
</tr>
<tr>
<td>1.13.1</td>
<td>Status Values for gettrend function</td>
<td>1-13-7</td>
</tr>
<tr>
<td>1.15.1</td>
<td>commopen Instruction Parameter Options</td>
<td>1-15-8</td>
</tr>
<tr>
<td>1.15.2</td>
<td>Status Variables</td>
<td>1-15-9</td>
</tr>
</tbody>
</table>
Section 1: RBL Concept

This section describes the Rosemount Basic Language (RBL) and programming techniques used to plan and write RBL programs called scripts. Scripts consist of instructions that tell the system how to perform a program.

RBL Script Structures

The RBL programming features enable you to structure scripts so that they are more comprehensible and easier to use. Structured programming offers the following advantages:

- It organizes RBL program code into functional units that are easier to understand and identify. Each unit of code can then be considered in the context of the activity it performs.
- It simplifies the creation of new scripts. New scripts can be created by combining or modifying sections of code.
- It enables the program to define functional environments for RBL instructions according to the program sections or tasks in which the instructions are used.
- It simplifies program maintenance. Problems can be traced to specific sections of code that are causing a problem.
Figure 1.1.1 shows how the major RBL language structures define RBL program hierarchy. Indented structures are subordinate to the structures above them.

**Task or Recipe (Batch only)** is the batch application to be performed.

**Scripts** are sets of RBL code for program hardware devices, formulas, and procedures.

**Phases** are user-defined names that divide script activities into logical units of code.

**Labels** are user-defined names for subordinate groupings of code within phases.

**Instructions** are code words that name program operations.

**Figure 1.1.1. RBL Program Hierarchical Structure**
RBL programs can use seven different types of scripts.

ABC Batch recipes use scripts 1 and 2.

1. **Library script**
   Defines either batch units or operations. If associated with a batch unit, the library script defines aliases for equipment tags and addresses. If associated with an operation, the library script defines the specific activity performed by the operation.

2. **Start script**
   Defines global procedures for all scripts used by the recipe. The start script executes in parallel with other recipe scripts. Values defined in the start script remain in memory the entire time the recipe is running.

Batch tasks configured with the Batch Run screen use scripts 3 through 5.

3. **Unit script**
   Describes the field hardware devices used to make a batch of product. These devices may include transmitters, valves, vessels, and other equipment associated with one process unit.

4. **Formula script**
   Describes process endpoints, setpoints, and targets for making a batch of product. Formula data may include flows, temperatures, rates, and volumes that define the product grade.

5. **Procedure script**
   Defines the steps associated with making a batch product. The effect of steps in a procedure may be to close valves, add products, start motors, and so on.

RBLC applications use only script 6.

6. **RBLC script**
   Defines the steps used to operate an RBL Controller (RBLC). The RBL Controller provides a serial interface to asynchronous ASCII devices, such as weigh scales and tank gauging systems.

   All RBL programs can use script 7.

7. **edit script**
   The edit script is useful as an intermediate step for converting one script type into another script type. You cannot run an EDIT script.
Phases

A phase is a user-defined name of a program line that is used to divide the script into sections. Each phase represents a separate script environment. The environment is the portion of the script that is active in memory. Phases are useful for easily identifying portions of a script. The start and gosub instructions direct the execution flow of the script to specified phases. Figure 1.1.2 shows the format for a phase.

NOTE: Phases are subject to the following rules:

- Phase names cannot be indented.
- An on instruction under a phase name is active only within that phase. An on instruction before all phase names in a script is active throughout the entire script.
- A phase name can be displayed on the BFACE graphic object, but must not exceed 10 characters.

SYNTAX
The general format is:

```
x:
```

where x is the user-defined phase name. All phase names must end in a colon (for example, LOAD:).

SAMPLE INSTRUCTION

In the following example, the gosub instruction directs execution to the "FILL:" phase. The script opens an intake valve represented by variable in and closes the valve when the tank level is less than a target value specified by the totalizer target variable tottrg. The script calls a label _START to start this procedure. The return instruction returns execution to the program line after the gosub instruction.

```
gosub FILL; “Add product”
.
.
FILL: “Fill Tank activity”
shared local choose; “Fill tank or stop task”
gosub _START: “Manual Start activity”
while (tot1 < tottrg)
in=1; “Open Intake Valve”
in=0; “Close Intake Valve”
return
```

Figure 1.1.2. Phase Format
Labels

A label is a user-defined name of a program line. A label begins with the underscore (_) character and is used to divide a phase into sections or direct script execution. Labels are also useful for easily identifying sections of a script. The `goto` and `gosub` instructions direct execution of the script to the label.

**NOTE:** Labels are global or local according to where they are declared within the script:

- A local label is declared within a phase. A local label is only meaningful within the phase in which it is contained. A `goto` or `gosub` instruction may not direct execution to a local label in another phase.

- A global label is declared before the first phase of the script. A global label is accessible from the entire script.

Figure 1.1.3 shows the format for a label.
SYNTAX
The general format is:

_\text{x}_:

where _\text{x}_ is the user-defined label name. All label names must be preceded by an underscore character (for example, _MIX1_).

SAMPLE INSTRUCTION
In the following example, the gosub instruction directs execution to the _\text{START}_ label. The input instruction writes the following message to Batch Input screen:

\text{Fill tank, Y or N}\Rightarrow

If the user enters “Y” at the message prompt (\Rightarrow), the Batch Input screen prints “Start Fill Cycle” and the return instruction returns execution to the line following the gosub instruction. If the user enters “N”, the task stops executing and returns to the “Finished” state.

gosub _\text{START}; “Manual start activity”

. . .

_\text{START}; “User starts phase”

prompt

input (“Fill tank, Y or N”, \text{CHOOSE})

if (\text{CHOOSE}==\text{Y})

\text{print(*,”Start Fill Cycle”)}

\text{return; “Start Fill phase”}

else

\text{finish; “Stop Task”}

Figure 1.1.3. Label Format
Symbols

Symbols are RBL data structures that represent values in scripts. Symbols can be local variables, global variables, aliases, constants, and arrays. The symbol type and name must be declared before it can be assigned a value. The symbol name enables the system to recognize it when it is used in a program line. For ease in use and troubleshooting, all symbols should be declared at the beginning of the script.

The value of most symbols can change during execution of the task (constants are the exception). The processor keeps track of declared symbols and their values in volatile memory. The symbol names are recorded in a symbol table. Their values are held in temporary memory and backed up to nonvolatile memory at timed intervals. The backups prevent the loss of the symbols’ current values in the event that the processor reboots.

Symbols are also designated as either shared or private. Shared symbols can pass values between scripts. Private symbols can only be used within the script in which they are defined.

The same type of symbols can be written singly or in groups. If grouped, each declaration is separated by a comma. Figure 1.1.4 shows the two ways to declare two local variables.

NOTE: For specific information on symbols, see Section 3. For information on the use of shared and private symbols in chained scripts, see Section 5.

<table>
<thead>
<tr>
<th>Declare singly on separate lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared local tottrg</td>
</tr>
<tr>
<td>private local levtrg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declare as a group on the same line</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared local tottrg, levtrg</td>
</tr>
</tbody>
</table>

Figure 1.1.4. Symbol Format
Instructions

Instructions are reserved words that name the operation to be performed and initiate an action. An instruction may also direct program execution to a particular phase or label of the script. The action taken by the instruction may be qualified by a parameter that limits the action of the instruction or specifies how the instruction is to operate. Parameters are typically enclosed in parenthesis. All instructions are reserved keywords and may not be used as variables.

Table 1.1.1 shows several examples of instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gosub DRAIN</td>
<td>Directs program execution to the DRAIN phase. (For more information about the <em>gosub</em> instruction, see Section 5.)</td>
</tr>
<tr>
<td>return</td>
<td>Returns the program to the previous line of execution. The <em>return</em> instruction is used in conjunction with a <em>gosub</em> instruction. (For more information about the <em>return</em> instruction, see Section 5.)</td>
</tr>
<tr>
<td>while tot1 &lt; tottrg</td>
<td>Executes lines indented under the <em>while</em> instruction while the totalizer value (TOT1) is less than the totalizer target (TOTTRG) of 30 gallons. (For more information about the <em>while</em> instruction, see Section 5.)</td>
</tr>
<tr>
<td>sleep(3)</td>
<td>Suspends script execution for three seconds. (For more information about the <em>while</em> instruction, see Section 5.)</td>
</tr>
</tbody>
</table>
**Instruction Status**

Some RBL instructions return a status value that can be checked in the script to see if the instruction was successful. The status value is assigned to a local variable, which can then be used as an expression in a logic instruction (if, while, until) to initiate an action if the instruction is successful or unsuccessful.

These instructions include:

- `acqfirst`
- `byte2flag`
- `flag2byte`
- `release`
- `task_status`
- `acquire`
- `close`
- `masterof`
- `relwait`
- `unacquire`
- `acqwait`
- `dequeue`
- `open`
- `strcmp`

Most of these instructions can return one of two status values.

- **Value:** status==1 Instruction was successful
- **status==0** Instruction was unsuccessful

The `acquire/release` and `commstat` instructions return other status values. (For more information about acquire/release instructions, see Section 10.)

Figure 1.1.5 shows an example of a script that checks the value of a variable called “status” for the status of an `open` instruction.

**SAMPLE INSTRUCTION**

```
shared local status; "declare variable for status"
.
.
status= open(1,"REPORTS","RPT1"); "Execute open instruction"
      "and return status to variable status"
if status
    goto _PRINT
```

*Figure 1.1.5. Status Format*

**Instruction Syntax**

In order for the script editor to parse instructions, you must enter them in a specified format, called syntax. Sections 3 through 15 describe the syntax for RBL instructions.
Nesting Stack

The nesting stack keeps track of outstanding nesting instructions. Nesting instructions add modular levels to scripts in the batch task in that each nesting instruction introduces a new functional level of code within another code level. The nesting stack limit is now 40 levels. The nesting count includes:

- all active indentation levels (on, while, for, if, until, and prompt instructions) in the current script and, if chaining has occurred, in the parent script and other chained scripts.
- all active gosubs in the current script and, if chaining has occurred, in the parent script and other chained scripts.
- one count for chaining. Do not count one for each chain, just one for the original chain.

The nesting stack contains dynamic areas for the global environment and the currently executing local environment of a script. These are the areas of a script that are active in volatile memory. The global environment is dynamic throughout the execution of the script. If a phase name (if it was not called by a gosub instruction or a start instruction) introduces a new local environment, it will clear the dynamic area of the nesting stack of the previous local environment.

If chaining has occurred, dynamic areas are created for the global and current local environments of each chained script. When an endchain occurs, the dynamic areas for the chained script are deleted.

For information on chain instructions, see Section 5. For information on how volatile memory is allocated to a task, see BA: 1–4.
Comments

Comments are usually explanations or documentation included in the script that explain what the script does. The Coordinator Processor (CP) ignores comments when running the scripts. Comments are useful to the configuror or anyone who needs to maintain or debug the script.

The comment line may contain any information but must begin and end with a quotation mark. If the comment appears on the same line as an instruction, a semicolon (;) must separate the comment from the instruction.

Comment Examples

The following are examples of comments.

“**DRAIN AND FILL STORAGE TANK**”

V1=OPEN; “Open the reactor intake valve”

**NOTE:** Comment lines can slow the execution speed of batch scripts, particularly when comments occupy an entire line. This can cause problems when comment lines are included in controlled instruction loops (until, while, and certain on instructions) that must operate at fast speeds. Comments included on the same line with the existing script do not affect execution speed as much as comment lines that occupy an entire line.

Disappearing Comments

Disappearing comments are used to prevent operators and other unauthorized users from reading script comments. These comments can only be viewed on the screen if a configuror key is in the console. Disappearing comments are delimited by a single quote marks (’). Standard comments are delimited by double quote marks (").
Tags and Addresses

Batch tasks can use ControlBlock variable values and ControlBlock modes to control task states and batch hardware.

ControlBlock Variable Values

ControlBlock tags and addresses can be used in script instructions to indicate block variable values. Figure 1.1.6 shows the tag format and an example.

NOTE: Tags and addresses are subject to the following rules:

- All continuous ControlBlock variables use values in the range of 0 to 1. (For information about scaling ControlBlock values, see the sclike and unsc instructions).
- The tag can write to and read values from ControlBlock input registers but can only read ControlBlock output values.

**Exception:** If the script is used by an RBL Controller and the tag references an RBL Controller block, then the tag can write to the ControlBlock output register Q.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td>{x/y}</td>
</tr>
<tr>
<td>where x is the ControlBlock tag or address.</td>
</tr>
<tr>
<td>y is the ControlBlock variable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAMPLE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>if {TT-1/B}&lt;.25; “Tag for continuous value”</td>
</tr>
<tr>
<td>while {=2B-12/a}==1: “Address for discrete value”</td>
</tr>
</tbody>
</table>

Figure 1.1.6. Sample Use of Tags and Addresses to Access ControlBlock Variables
ControlBlock Modes

ControlBlock tags and addresses can be used to read the ControlBlock mode. ControlBlock modes are useful as expressions of conditional instructions (if, while, and others) for directing the execution of the script. However, the block mode cannot be changed from within a script. Figure 1.1.7 shows the format and an example of using ControlBlock modes.

**SYNTAX**

The general format is:

```
{x/MD}==y
```

where   
\( x \) is the ControlBlock tags or address.  
\( y \) is the ControlBlock mode.

**SAMPLE INSTRUCTION**

if \( \{TT-1/MD\}==0 \); “Execute if mode is Local”  
while \( \{=2B-12/MD\}==1 \); “Execute while mode is Manual”

Figure 1.1.7. Sample Use of Tags and Addresses to Select ControlBlock Modes

Table 1.1.2 shows the continuous input values that may be indicated with the ControlBlock mode (MD):

**Table 1.1.2. ControlBlock Modes Status**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD</td>
<td>Block Mode</td>
</tr>
<tr>
<td></td>
<td>0=LOCAL</td>
</tr>
<tr>
<td></td>
<td>1=MANUAL</td>
</tr>
<tr>
<td></td>
<td>2=AUTO</td>
</tr>
<tr>
<td></td>
<td>3=REMOTE</td>
</tr>
<tr>
<td></td>
<td>4=COMPUTER</td>
</tr>
</tbody>
</table>
Instructions That Retry

Some RBL instructions retry execution until the instruction is successful. If an error condition persists, these instructions will retry continuously, suspending execution at the instruction line.

These instructions include:
- aliases
- global variables
- task instructions: abort_task, begin_task, cont_task, halt_task, kill_task, resume_task, status_task, stop_task,
- retry
- acqwait
- relwait

These instructions can create problem if included with other instructions in expressions. For example, including an alias in an expression with a report open instruction could cause the report to open and close repeatedly:

shared alias BLCK:=129A-12/@a

if (open(1,"DEMO","RPRT_1")&BLCK)

When using instructions that retry, do not include them in expressions with instructions that can affect the system or batch process. To use the above expression correctly, write as follows:

shared alias BLCK:=129A-12/@a

if (open(1,"DEMO","RPRT_1")

   if (BLCK)
Script and ControlBlock Synchronization

If a script writes values to a ControlBlock discrete input register or to a continuous register with a Rate Limit, the values may not be immediately available for use by the script.

The following describes two cases:

- A script cannot read a discrete output until the ControlBlock completes one full evaluation cycle. An evaluation cycle is the time required for the ControlBlock to receive a value and make it available in discrete output. The evaluation cycle is listed as the “ScanTime” on the Control File Status screen for the specified Controller Processor card.

- A script may not be able to write a continuous value to a ControlBlock that has a Rate Limit associated with the register link and immediately attempt to read the value. This is because the block requires additional time to determine if the value falls within the rate limit scale. If the continuous register does not have a Rate Limit, then values written to it are available immediately to the script. Rate limits are used with PID setpoints, ratio, and bias inputs (see CB: 2–2).

In Figure 1.1.8, a Rate Limit assigned to a ControlBlock link affects the execution of the code.

DEMONSTRATION OF FUNCTION

The following example shows two scripts in which the hold instruction should never execute. If the ControlBlock link does not have a Rate Limit, then X will always equal Y, as in Script A. If a Rate Limit is assigned to the ControlBlock register \{=1a-1/B\}, the value of the register may not equal X when the script attempts to read the register. This is because the script and ControlBlock are executing at different speeds. Script B uses a sleep instruction to slow the execution of the script in synchronization with the ControlBlock execution cycle.

<table>
<thead>
<tr>
<th>Script A</th>
<th>Script B</th>
</tr>
</thead>
<tbody>
<tr>
<td>“No Rate Limit assigned to {=1a-1/A}”</td>
<td>“Rate Limit assigned to {=1a-1/B}”</td>
</tr>
<tr>
<td>for X=0, X&lt;1, X=X+.01</td>
<td>for X=0, X&lt;1, X=X+.01</td>
</tr>
<tr>
<td>{=1a-1/A}=X; Y={=1a-1/A}</td>
<td>{=1a-1/B}=X; Y={=1a-1/B}</td>
</tr>
<tr>
<td>if X_=Y</td>
<td>sleep(1.5); “Scan time of 1 sec”</td>
</tr>
<tr>
<td>hold</td>
<td>if X_=Y</td>
</tr>
<tr>
<td></td>
<td>hold</td>
</tr>
<tr>
<td></td>
<td>“Slowing script also applicable”</td>
</tr>
<tr>
<td></td>
<td>“with reading discrete registers”</td>
</tr>
</tbody>
</table>

Figure 1.1.8. Using Rate Limits
Table 1.1.3 lists RBL parameters and their corresponding limits. The parameters in this table are also affected by available volatile and nonvolatile memory.

### Table 1.1.3. RBL Parameter Limits

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aliases</td>
<td>Amount of symbol space</td>
</tr>
</tbody>
</table>
| Arrays (dim) | 250 arrays per task (or as limited by CP volatile memory)  
255 indices per array dimension |
| Arrays (vdim) | 50 x 50 kilobytes (or as limited by CP volatile memory) |
| Characters in names of constants, global variables, labels, local variables, phases, and strings | 18 characters |
| Characters on alarm line | 21 characters |
| Characters in a string variable | 269 characters |
| Constants | Amount of symbol space |
| Delete buffer for batch editor | 40 Kilobytes |
| Global variables | 250 variables per batch task or RBLC script (or as limited by the batch task symbol space) |
| Local variables | 250 variables per batch task or RBLC script (or as limited by the batch task symbol space) |
| Non Volatile Memory | Maximum available space, 56 Kilobytes |
| Open report files | A maximum number of report files open at any one time:  
10 report files per batch task  
20 report files per 68000 console  
50 report files per 68020 console |
| RBL files | As configured on Disk Folder Configuration screen (see SP: 4-1). Maximum of 1000. |
| RPN space | 31000 bytes for script code  
9000 for script comments |
| Volatile Memory | Maximum available space for a CPIV:  
Using a Batch Recipe, 50.3 Kilobytes  
Using a Batch Run screen, 56 Kilobytes |

(continued on next page)
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>10000 lines, 31000 bytes</td>
</tr>
<tr>
<td>Scripts in a file</td>
<td>100 scripts</td>
</tr>
<tr>
<td>String variables</td>
<td>250 strings per batch task or RBLC script (or as limited by the batch task symbol space)</td>
</tr>
</tbody>
</table>
| Symbol space               | 20000 bytes per batch task or as limited by CP volatile memory (batch only)  
|                            | The batch task symbol table includes names of local variables, global variables, string variables, arrays, aliases, and constants that are defined in the task, along with names of labels and phases in the currently executing script. The longer the name, the more space it takes up in the symbol table. |
| Tasks                      | 32 batch tasks or as limited by CP volatile memory (batch only)       |
| Volatile memory            | Maximum available space:  
|                            | For a CP2, 200 Kilobytes  
|                            | For a CP4, 786 Kilobytes    |
Mathematical Operators

Script instructions can use the same operators as ControlBlock logic steps. Table 1.1.4 lists the mathematical functions available in RBL.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Format</th>
<th>Example/Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>relational equality—Asks whether two variables or expression are equal</td>
<td>x==y</td>
<td>Does x equal y?</td>
</tr>
<tr>
<td>=</td>
<td>assignment—Sets one expression equal to the value of another expression</td>
<td>x=y</td>
<td>x equals y</td>
</tr>
<tr>
<td>( )</td>
<td>order evaluation—Operations within parentheses are performed first</td>
<td>(x+y) *z</td>
<td>(2+10)*4 is 48</td>
</tr>
<tr>
<td>x?y:z</td>
<td>if–else selection</td>
<td>v=x?y:z</td>
<td>If x is true, v equals y, if x is false, v equals z</td>
</tr>
<tr>
<td>round</td>
<td>rounding function—Variable or expression is rounded to nearest integer</td>
<td>round x</td>
<td>round 2.2 is 2, round 2.6 is 3</td>
</tr>
<tr>
<td>int</td>
<td>integer function—Truncates a number to an integer</td>
<td>int x</td>
<td>int 4.73 is 4, int -4.2 is -.8</td>
</tr>
<tr>
<td>fract</td>
<td>fraction function—Returns the fractional part of a number</td>
<td>fract x</td>
<td>fract 3.2 is .2, fract -3.4 is .6</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value function</td>
<td>abs x</td>
<td>abs 5 is 5, abs -5 is 5</td>
</tr>
<tr>
<td>sign</td>
<td>signum (sign) function Returns +1, 0, or -1</td>
<td>sign x</td>
<td>sign -4 is -1</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root function—Returns the square root of a number</td>
<td>sqrt x</td>
<td>sqrt 16 is 4, sqrt .0001 is .01</td>
</tr>
<tr>
<td>sqrl</td>
<td>square root limited gain function Returns the square root of x if x&gt;1% of scale; otherwise returns 10*x</td>
<td>sqrl x</td>
<td>sqrl 16=4, sqrl .0001=.001</td>
</tr>
<tr>
<td>sin</td>
<td>sine trig function (radians)</td>
<td>sin x</td>
<td>sin 1 is .841471</td>
</tr>
<tr>
<td>asin</td>
<td>inverse sine trig function</td>
<td>asin x</td>
<td>asin 1 is 1.57080</td>
</tr>
</tbody>
</table>

NOTE: x, y, and z may be single discrete variables (@a through @p), single continuous variables (A–O), or expressions.

(continued on next page)
### Table 1.1.4. Mathematical Function Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Format</th>
<th>Example/Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos</td>
<td>cosine trig function (radians)</td>
<td>( \cos x )</td>
<td>( \cos 0 ) is 1.000</td>
</tr>
<tr>
<td>acos</td>
<td>inverse cosine trig function</td>
<td>( \arccos x )</td>
<td>( \arccos .5 ) is 1.04720</td>
</tr>
<tr>
<td>tan</td>
<td>tangent trig function (radians)</td>
<td>( \tan x )</td>
<td>( \tan .90 ) is 1.26016</td>
</tr>
<tr>
<td>atan</td>
<td>inverse tangent trig function</td>
<td>( \arctan x )</td>
<td>( \arctan .5 ) is .463648</td>
</tr>
<tr>
<td>log2</td>
<td>base 2 log function</td>
<td>( \log_2 x )</td>
<td>( \log_2 600 ) is 9.23</td>
</tr>
<tr>
<td>exp2</td>
<td>exponent of 2 function</td>
<td>( \exp^2 x )</td>
<td>( \exp^2 6 ) is 26 is 64</td>
</tr>
<tr>
<td>log</td>
<td>base 10 log function</td>
<td>( \log x )</td>
<td>( \log 52,400 ) is 4.72</td>
</tr>
<tr>
<td>ln</td>
<td>base e log function (natural log)</td>
<td>( \ln x )</td>
<td>( \ln 15 ) is 2.71</td>
</tr>
<tr>
<td>exp</td>
<td>exponent of e function</td>
<td>( \exp x )</td>
<td>( \exp 1 ) is 2.72</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>( x^{**} y )</td>
<td>( 2^{**} 3 ) is 8</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>( x*y )</td>
<td>( 4*3 ) is 12</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>( x/y )</td>
<td>( 12/4 ) is 3</td>
</tr>
<tr>
<td>%</td>
<td>divide modulo (remainder)</td>
<td>( x%y )</td>
<td>( 5%2 ) is 1</td>
</tr>
<tr>
<td>max</td>
<td>maximum value function</td>
<td>( x \max y )</td>
<td>( 10 \max 1 ) is 10</td>
</tr>
<tr>
<td>min</td>
<td>minimum value function</td>
<td>( x \min y )</td>
<td>( 10 \min 1 ) is 1</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>( x+y )</td>
<td>( 7+2 ) is 9</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>( x-y )</td>
<td>( 9-7 ) is 2</td>
</tr>
<tr>
<td>_ =</td>
<td>relational inequality</td>
<td>( _ = x )</td>
<td>( 3_ = 6 ) is true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>relational greater than or equal to</td>
<td>( x&gt;=y )</td>
<td>( 8&gt;=6 ) is true</td>
</tr>
<tr>
<td>&gt;</td>
<td>relational greater than</td>
<td>( x&gt;y )</td>
<td>( 12&gt;4 ) is true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relational less than or equal to</td>
<td>( x&lt;=y )</td>
<td>( 2&lt;=2 ) is true</td>
</tr>
<tr>
<td>&lt;</td>
<td>relational less than</td>
<td>( x&lt;y )</td>
<td>( 5&lt;8 ) is true</td>
</tr>
<tr>
<td>_</td>
<td>unary logical negation</td>
<td>( _ x )</td>
<td>( x ) is negated</td>
</tr>
<tr>
<td>&amp;</td>
<td>conditional and—true if both expressions are true; false otherwise</td>
<td>( x&amp;y )</td>
<td>returns true (1) if ( x ) and ( y ) are &gt;0 otherwise, returns false (0)</td>
</tr>
</tbody>
</table>

**NOTE:** \( x, y, \) and \( z \) may be single discrete variables (@a through @p), single continuous variables (A–O), or expressions.

(continued on next page)
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Format</th>
<th>Example/Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>conditional inclusive or—true if either one expression or another or both is true; false otherwise</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>↑</td>
<td>conditional exclusive or—true if either one expression or another is true; false otherwise</td>
<td>x↑y</td>
<td>returns true (1) if x&gt;0 or y&gt;0, but not both otherwise, returns false (0)</td>
</tr>
</tbody>
</table>

**NOTE:** x, y, and z may be single discrete variables (@a through @p), single continuous variables (A–O), or expressions.
Section 2: Script Management

This section includes procedures that you use to maintain and edit scripts used in batch and RBLC applications.

Maintenance procedures enable you to create and organize files of batch scripts. A maximum of 100 scripts can be stored in an RBL File Contents directory. Editing functions enable you to enter characters in a script, parse program lines, and write scripts to disk.
Creating an RBL Files Folder and RBL File

This procedure creates the following batch storage media:

- The RBL Files Folder, if one does not already exist.
- A new RBL File.
- The first batch or RBLC script in the new RBL File.

None of these media exist until you press [CTRL] [W] to write the first script in the new file to disk.

Figure 1.2.1 shows how to create and copy an RBL File and an initial script.

-To create an RBL File and first script:

1. Specify the first script and file.
   - For a batch script and script file, type:
     \[BAS: (volume, filename) [ENTER]\]
   - For a RBLC script and script file, type:
     \[RBLS: (volume, filename) [ENTER]\]

   The volume specifies the console disk; the filename specifies the script file. If there is no file by this name in the volume, it is created when you press [ENTER]. It contains a default blank script called “script”.

   A blank Batch Script screen or RBLC Script screen appears for the script called “script”. You can use this script to write, edit, and save new scripts to the script file.

2. Cursor to the “Script” field and type a new script name. Press [ENTER] to replace default name “script” with this name.

3. Cursor below line #1 to the upper left corner of the blue END-OF-SCRIPT designation. Enter at least two lines of script code to put character space into the file and give it size. Pressing [ENTER] twice to create two blank lines will also give the script space.


   A new RBL file is created containing two scripts: “script” and your new script.
Figure 1.2.1 shows the Batch Script screen as it appears when you create a new RBL File.

![Batch Script Screen Diagram]

- **To select a script type:**
  - Enter one of the following script types: LIBR, START, UNIT, FORM, PROC, RBLC, EDIT.

- **To change editing mode:**
  - Cursor to field and press [ENTER], or [CTRL] [T] to change.
  - “Insert” indicates that entered characters are inserted in the text.
  - “Replace” indicates that entered characters replace existing characters.

Volume and filename of the RBL file that contains this script
Name of this script

Program lines (empty lines in this example)
End of script (default color is blue)
Size of the script, in bytes (max. size is 31000)

**Figure 1.2.1. Batch Script Screen**
RBL File Contents screen

The RBL File Contents screen displays a list of scripts within an RBL file. This screen is useful for selecting scripts within a file to print or to edit. Figure 1.2.2 shows a RBL File Contents screen and Table 1.2.1 lists and describes the fields on this screen.

To call up the RBL File Contents screen, type:

RBLF (node, filename) [ENTER]
### Table 1.2.1. RBL File Contents Screen Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field 1</td>
<td>The following fields show disk storage parameters for a script:</td>
</tr>
<tr>
<td>Disk name</td>
<td>The disk volume in which the RBL file resides.</td>
</tr>
<tr>
<td>On drive</td>
<td>The disk drive in which the RBL file resides.</td>
</tr>
<tr>
<td>RBL file</td>
<td>The RBL file in which the batch script displayed on the RBL File Contents screen resides.</td>
</tr>
<tr>
<td>Created on</td>
<td>The date on which the RBL file was created.</td>
</tr>
<tr>
<td>Last Modified</td>
<td>The year, month, day and time when a script in the RBL file was last created or modified.</td>
</tr>
<tr>
<td>Last Backup Time</td>
<td>The time when the RBL file was last backed up to disk.</td>
</tr>
<tr>
<td>Field 2</td>
<td>The following fields identify software parameters for a script:</td>
</tr>
<tr>
<td>Script name</td>
<td>The names of the batch scripts that reside within the RBL file.</td>
</tr>
<tr>
<td>Type</td>
<td>The type of the batch script: start, library, unit, formula, procedure, edit, or RBLC script.</td>
</tr>
<tr>
<td>Rpn size</td>
<td>The number of bytes used by the script. A batch script can use a maximum of 31000 bytes (or as limited by CP volatile memory).</td>
</tr>
<tr>
<td>Sym size</td>
<td>The symbol space in bytes used by the script. A batch script can use a maximum of 16000 bytes (or as limited by CP volatile memory).</td>
</tr>
<tr>
<td>Lines</td>
<td>The number of lines in the batch script.</td>
</tr>
<tr>
<td>Modification time</td>
<td>The month, day and time when the script was created or last modified.</td>
</tr>
<tr>
<td>Level</td>
<td>The software level of the script. The level enables customers to keep track of script levels to determine if the script should be updated to a new software level.</td>
</tr>
<tr>
<td>Checksum</td>
<td>An internal identification code that is used to monitor changes to the script. If the script is changed, the checksum changes value.</td>
</tr>
<tr>
<td>Mod</td>
<td>The number of modifications to the script used in ABC Batch. When the Control Recipe is validated, the current modification level is compared to a past modification level at the time the Control Recipe was created. If the levels differ, the Control Recipe performs the validation action specified in the “Validate” field on the Batch Operations Table: None, Halt, or Warn.</td>
</tr>
</tbody>
</table>
Maintaining Script Files

- On the RBL File Contents screen, the following fields are used to perform script operations:
  - [ENTER] to print
  - [SELECT] to read
  - [CTRL] [d] to Delete
  - [CTRL] [q] to copy

Printing a Script

- To print a script:
  - Cursor to the script and press [ENTER]. The script file is sent to the printer, as shown in Figure 1.2.3.

<table>
<thead>
<tr>
<th>Disk name</th>
<th>RBL FILE CONTENTS</th>
<th>21-Mar-93</th>
<th>23:28:38</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBL FILE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Created on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Backup Time:</td>
<td>Destination Script &gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Descriptor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---ENTER-to-print----SELECT-to-read----CTRL-d to Delete----CTRL-q to copy---

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Type</th>
<th>Size</th>
<th>Sym size</th>
<th>Lines</th>
<th>Modification time</th>
<th>level</th>
<th>Checksum Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>LIBR</td>
<td>1045</td>
<td>255</td>
<td>55</td>
<td>11-Nov-91 15:07:50</td>
<td>11.6</td>
<td>KPM72A</td>
</tr>
<tr>
<td>Add_Raw_1</td>
<td>LIBR</td>
<td>1045</td>
<td>316</td>
<td>70</td>
<td>11-Nov-91 13:17:40</td>
<td>11.6</td>
<td>DKULN8</td>
</tr>
<tr>
<td>Add_Raw_2</td>
<td>LIBR</td>
<td>1195</td>
<td>267</td>
<td>80</td>
<td>11-Nov-91 13:45:12</td>
<td>11.6</td>
<td>LJ7N40</td>
</tr>
<tr>
<td>AUTO_ID</td>
<td>PROC</td>
<td>280</td>
<td>104</td>
<td>29</td>
<td>10-Nov-91 10:15:17</td>
<td>11.6</td>
<td>SSU74A</td>
</tr>
</tbody>
</table>

Press [ENTER]

Figure 1.2.3. Printing a Script
## Calling Up a Script

**To call up a script:**
- Cursor to the script and press [SELECT], as shown in Figure 1.2.4.

Batch scripts are displayed on the Batch Script screen. RBL scripts are displayed on the RBLC Script screen.

### Press [SELECT]

<table>
<thead>
<tr>
<th>Disk name</th>
<th>RBL FILE CONTENTS</th>
<th>21-Mar-93</th>
<th>23:28:38</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>At node On drive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBL FILE</td>
<td>SIZE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Created on</td>
<td>Destination Script &gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Backup Time:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Descriptor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

------ENTER-to-print------SELECT-to-read------CTRL-d to Delete------CTRL-q to copy------

<table>
<thead>
<tr>
<th>Script name</th>
<th>Type</th>
<th>Rpn size</th>
<th>Sym size</th>
<th>Lines</th>
<th>Modification time</th>
<th>level</th>
<th>Checksum</th>
<th>Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>LIBR</td>
<td>1045</td>
<td>255</td>
<td>55</td>
<td>11-Nov-91 15:07:50</td>
<td>11.6</td>
<td>KPM72A</td>
<td>7</td>
</tr>
<tr>
<td>Add_Raw_1</td>
<td>LIBR</td>
<td>1045</td>
<td>316</td>
<td>70</td>
<td>11-Nov-91 13:17:40</td>
<td>11.6</td>
<td>DKULN8</td>
<td>8</td>
</tr>
<tr>
<td>Add_Raw_2</td>
<td>LIBR</td>
<td>1195</td>
<td>267</td>
<td>80</td>
<td>11-Nov-91 13:45:12</td>
<td>11.6</td>
<td>LJ7N40</td>
<td>5</td>
</tr>
<tr>
<td>AUTO_ID</td>
<td>PROC</td>
<td>280</td>
<td>104</td>
<td>29</td>
<td>10-Nov-91 10:15:17</td>
<td>11.6</td>
<td>SSU74A</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 1.2.4. Calling up a Script
Deleting a Script

- To delete a script:
  - Cursor to the script and press [CTRL] and [d] at the same time, as shown in Figure 1.2.5.

![Figure 1.2.5. Deleting a Script]
Copying a Script

To copy a script:

1. Cursor to the script and press [CTRL] and [q] at the same time. The following field appears in the top right corner of the screen:

   Destination Script.

2. Enter the name of the script to which you want to transfer the copy in the "Destination Script" field, as shown in Figure 1.2.6. A copy or a script with this name appears in the script list.

![Figure 1.2.6. Copying a Script](image)
Writing and Editing Scripts

Use the Batch Script screen (shown in Figure 1.2.7) to edit batch scripts. (Use the RBLC script screen to edit RBLC scripts.) Both screens have similar functions. This section describes script editing techniques and functions.

You can use the editing functions to enter characters in a script, parse program lines, and write scripts to disk. To perform editing functions, hold down the [CTRL] key and press another key at the same time. In this chapter, these actions are described simply as [CTRL] [key], (for example, [CTRL] [W]).

NOTE: Do not edit scripts while they are running. If the CP reboots, it will detect a different checksum for the script and will not run it. The system generates the message Fatal RPN.
Entering Text

To write a script, you enter characters on the RBLC Script and Batch Script displays to form program lines. When you first enter a line of characters at the console, the line only exists in the console screen memory and it appears yellow on the screen. The yellow indicates that the line has not been checked for errors and has not been entered into the script.

- **To toggle between text Insert mode and Replace mode:**
  - Cursor to the "insert/replacement" field and press [ENTER].

  or

  - Press [CTRL] [T] from anywhere on the screen.

- **To enter text characters on a program line:**
  - Move the cursor to the desired location and type the text characters. The program lines containing the characters turn yellow.
    - If the script editor is in Insert mode, new characters are inserted to the left of the cursor.
    - If the script editor is in Replace mode, new characters replace the characters beneath the cursor.

- **To create an empty line above the existing line:**
  - Move the cursor to any character in the existing line.
    Press [CTRL] [I].
Copying a Script or Script File from the Batch Script Screen

The following procedure describes how to copy a Script File, a script, or both without leaving the Batch Script screen. Figure 1.2.7 shows the Batch Script screen.

To copy a script and script file:

1. Cursor to the “File” and “Script” fields and make any desired changes. Press [ENTER] at each field to enter the changes.

   If the specified script does not exist, the message appears on the screen:

   Script does not exist on disk yet. (CTRL ‘w’)

   The screen has been called up, but not yet created.

2. Press [CTRL] and [W] at the same time to create the script and write it to disk.

Figure 1.2.7. Copying from the Batch Script Screen
Deleting and Inserting Text

You can delete text to remove errors or revise scripts. Lines that are deleted from a script are placed into a delete buffer. You can retrieve the last text deleted from the delete buffer. When you retrieve text, it is inserted onto a program line at the cursor. The delete and insert functions enable you to move text to other lines in the script, an activity sometimes referred to as “cut and paste”.

- **To delete a character:**
  - Move the cursor to the desired character. Press [CTRL] [X].

- **To delete a line:**
  - Move the cursor to any character in the existing line. Press [CTRL] [D]. The deleted line is placed in a delete buffer.

**NOTE:** You can delete a sequences of lines by pressing [CTRL] [D] repeatedly. However, do not move the track ball until you have completed deleting lines in the sequence. Moving the trackball signals the end of a delete sequence. The delete buffer will replace any lines you delete before moving the track ball with lines you delete after moving the track ball.

- **To insert the delete buffer into the text:**
  1. Move the cursor to the desired location.
  2. Press [CTRL] [Y]. The text in the delete buffer is entered onto the line at the cursor.
NOTE:

- The delete buffer is cleared when you leave the Batch Script screen or delete subsequent text after moving the track ball. Once the buffer is cleared, the contents cannot be retrieved.

- The delete buffer will not allow you to delete more lines than the delete buffer can hold (maximum of 240 lines). The following message is displayed:

  Delete buffer full, cannot delete any more lines

  If you see this message, press [CTRL] [Y] to insert the current contents of the delete buffer in the script, or move the track ball to a new line to delete additional text.

- If the delete buffer contains more lines than can be inserted in the script, the contents of the delete buffer cannot be copied. This feature prevents you from filling up the script by repeatedly pressing [CTRL] [Y].

  If you cannot insert the contents of the delete buffer, you must first save the script before continuing.
Parsing and Writing Text to Disk

Program lines must be parsed to become part of the script. Parsing checks the syntax of the program lines. Parsing does not store the program lines onto the disk.

The following colors indicate the parse status of the program line:

- **Green**: Program lines that have been successfully parsed appear green on the screen.
- **Red**: Program lines that have been unsuccessfully parsed and that have generated syntax errors appear red on the screen. The first character in the probable error blinks.
- **Yellow**: Program lines that have been edited but not parsed appear yellow on the screen.

You must write a script to disk to save it and use it again.

- **To parse a line and start a new line under it:**
  1. Make sure the script is in the insert mode.
  2. Move the cursor to the end of the desired line and press [ENTER].

- **To parse all unparsed lines in the script:**
  - Press [CTRL] [P] from anywhere on the screen.

- **To parse all unparsed lines and write the entire script to disk, if no syntax errors exist:**
  - Press [CTRL] [W] from anywhere on the screen. It is recommended that you use this command frequently when writing scripts.

- **To undo changes made to the line last edited before it is parsed:**
  - Press [CTRL] [U] from anywhere on the screen.

- **To write a script to disk without parsing for all errors (some errors such as goto labels are parsed):**
  - Press [CTRL] [E] from anywhere on the screen. The script is saved as an EDIT only type script and cannot be loaded to the Coordinator Processor. A script that is in the edit mode can be moved between the Batch Script screen and the RBLC Script screen.
Reading and Moving Scripts

You can read a script from the disk to the console, move program lines between scripts, or move a script between a Batch Script file and an RBLC Script file.

- When you read a script from disk, the script overwrites any script currently in the console memory. You can use the read function to delete a script by overwriting it (refer to “Deleting Scripts”) or you can make duplicates of existing scripts.

- When you change an RBLC script to a batch script or vice versa, the script is moved between a Batch Script file and an RBLC Script file. You can use this feature to change the status of a script.

To read a script from disk:

1. Move the cursor to the “File” and “Script” fields and enter the desired volume, filename, and script name. The following message appears:

   Script exists on disk. Hasn’t been read. (CTRL ‘r’)

2. Press [CTRL] [R] from anywhere on the screen to read the script to the screen.
To copy lines between scripts:

1. Press [CTRL] [D] once on each line that you want to copy. The lines are deleted from the screen and stored in a delete buffer.

   **CAUTION**

   If you want to copy more than one line, do not move the cursor between line deletes or all but the last line you delete will be cleared from the delete buffer. Because you cannot move the cursor, you can only delete multiple lines if they follow in sequence.

2. Without moving the cursor, press [CTRL] [Y] to restore all deleted lines.

3. Press [CTRL] [W] to save the deleted lines in the original script.

4. Move the cursor to the "Script" field. Enter the name of the script to which you want to copy the lines. If the script exists, the following message appears:

   Script exists on disk. Hasn’t been read. (CTRL ‘r’)

5. Press [CTRL] [R] to read the script to the screen.

6. Move the cursor to the line below where you want the copied lines to appear and press [CTRL] [Y] to copy the lines.

To move a script between a Batch script file and an RBLC script file:

1. Press [CTRL] [E]. The script is saved as an EDIT only type script.

2. Move the cursor to the command line and call up a script file.
   - For the batch script file, type:
     ```
     BAS [ENTER]
     ```
   - For the RBLC script and script file, type:
     ```
     RBLS [ENTER]
     ```

3. Delete any instructions that cannot be used in the new script type.

4. Press [CTRL] [W]. The script is saved either as a batch script or an RBLC script.
Using Search and Replace Functions

You can use search functions to search forward and backward in the script for a string of text. An optional replace function lets you replace the string of text with another string of text. The search function searches for a text string that you enter in the upper-left corner of the display line, as shown in Figure 1.2.8.

![Batch Script](image)

**Figure 1.2.8. Search Functions**

**NOTE:** If the search function cannot find the character string, it displays the following message:

String not found

- To search forward in the script to find a character string:
  1. Move the cursor to the display line.
  2. Type a slash mark (/) followed by the character string as follows:
     `/search_string [ENTER]`
     
     A slash mark (/) preceding the character string instructs the search function to search forward in the script. The screen scrolls to the script line that contains the character string.

- To search backwards in the script to find a character string:
  1. Move the cursor to the display line.
  2. Type a question mark (?) followed by the character string as follows:
     `?search_string [ENTER]`
     
     A question mark (?) preceding the character string instructs the search function to search backward in the script. The screen scrolls to the script line that contains the character string.
To search forward in a script to find a label or phase name:
1. Move the cursor to the display line.
2. Type the label, phase name, or line number as follows:
   phase: [ENTER]
   or
   _label [ENTER]
   The screen scrolls to the script line that contains the phase or label name.

To search forward in a script to find a line number:
1. Move the cursor to the display line.
2. Type the line number and press [ENTER]. The screen scrolls to the script line identified by the line number.

To search for and replace a character string:
1. Move the cursor to the display line.
2. Type two character strings separated by slash marks (/) as follows:
   /search_string/replace_string/ [ENTER]
   The editor scrolls the screen to the first script line that contains the character string you want to find and replaces it. Each time you press [ENTER], the editor will search and replace the next instance of the character string in the script.

To confirm a search and replace of a character string:
1. Move the cursor to the display line.
2. Type two character strings separated by slash marks (/) and the confirm parameter “c” as follows:
   /search_string/replace_string/c [ENTER]
   The screen scrolls to the first script line that contains the character string you want to find and pauses, so you can choose whether or not to replace it. The following prompt appears:
   replace string(y/n)
3. Type either “y” to replace the character string, or “n” not to replace the string, and press [ENTER]. Each time you press [ENTER], the screen scrolls to the next instance of the character string. The confirmation parameter “y” or “n” remains in effect until you change it.
To search and replace a character string globally in the script:

1. Move the cursor to the display line.
2. Type two character strings separated by slash marks (/), and the global parameter “g” as follows:

   /search_string/replace_string/g  [ENTER]

   The editor will automatically find each instance of the character string and replace it throughout the entire script.
Script Editing Colors

The colors described in this chapter for editing functions are the default editing colors. You use the Configuration Color Usage screen (mnemonic **CCU**) to configure screen colors. Table 1.2.2 lists the default script editing colors.

<table>
<thead>
<tr>
<th>Color Configuration Screen Field</th>
<th>Editing Function</th>
<th>Default color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Program</td>
<td>Parsed program lines with no syntax errors.</td>
<td>2r green</td>
</tr>
<tr>
<td>Batch Error</td>
<td>Parsed program lines with parsing errors.</td>
<td>15r red</td>
</tr>
<tr>
<td>Batch Modified</td>
<td>Program lines that have been edited but not parsed.</td>
<td>11r yellow</td>
</tr>
<tr>
<td>Batch End</td>
<td>The end of the script.</td>
<td>3r blue</td>
</tr>
<tr>
<td>Batch Current Line</td>
<td>The currently executing line. Monitoring only.</td>
<td>4r cyan</td>
</tr>
<tr>
<td>Batch Comments</td>
<td>All comments within a script.</td>
<td>6n orange</td>
</tr>
</tbody>
</table>
Calling Up a Script

You can call up batch and RBLC scripts in several different ways, as shown in Figure 1.2.9. From the command line, you can type the mnemonic for a script or display screen associated with the script and press [ENTER]. From a screen display, you cursor to a field associated with the script or to another display screen and press [SELECT]. You may have to move through several levels of display screens to reach the script.
Figure 1.2.9. Calling Up Scripts
Section 3:  
Numeric and String Symbols

RBL symbols are data structures that represent numeric and character string values. Symbols include variables, arrays, constants, and aliases. Symbols are typically used in arithmetic computations or to pass data values between scripts, in the script and a ControlBlock (as in the case of an alias), or in different tasks running on the same batch node.

Symbols are stored in a symbol table in volatile memory while the batch task is executing. Symbol values are backed up to nonvolatile memory at timed intervals. In the event of a reboot, the CP will reload the most recently saved symbol values from nonvolatile memory. For more information about symbols, see BA: 1-4.

NOTE:

- The following terms are used in association with aliases and constants.
  
  definition: Defining an alias for a block address or a constant for a value. The alias or constant must be defined before it can be declared for use in a script.

    alias myblock:=13A-1
    constant myvalue:15

In ABC Batch, you should define all symbols in the library script for the batch unit (specified in the Batch Unit Table).

declaration: Declaring an alias or constant for use in an RBL script. An alias or constant declaration consists of the keyword alias or constant and the symbol name. Symbols need to be declared before they can be used.

    alias myblock
    constant myvalue

- If you want to back up symbol values to nonvolatile memory at preselected points in the script, use the backup instruction (see Section 6).

- Alias values are not backed up to nonvolatile memory. The alias address or tag for the ControlBlock is read from the script and the alias value is always current in the ControlBlock that is indicated by the address.
Shared and Private symbols

Symbols are designated as either shared or private in RBL scripts. Shared and private symbols are used differently depending on the type of Batch configuration option you choose:

- Batch Recipe
- Batch Run Screen

Batch Recipe Use of Shared and Private Symbols

The Batch Working Recipe can coordinate the execution of several scripts in several separate tasks when running a batch configuration. In recipe applications, shared and private symbols function as follows:

Shared symbols:

- Are global symbols that can be used by all scripts associated with the recipe.
- Are stored in the recipe symbol table for the entire duration of recipe execution. The batch unit that is assigned to the Working Recipe specifies the task address of the shared symbol table.
- Can be used to pass values between scripts used in parallel operations or consecutive operations.
- In start scripts, “shared” is the default symbol value assigned when the script is created.
- Shared global variables can be defined in task 1 of the batch node for use by other tasks running in the same batch node. (For more information on global variables, see the discussion of global variables in this section.)
Private symbols:

- Are local symbols that can be used only within the script in which they are defined.
- Are stored in the script symbol table for the operation task (operations use task 33 to 64) only while the library script is executing. When the script completes execution, the private symbols are cleared from the symbol table, freeing up that symbol space for use by other operations.
- In library scripts, “private” is the default symbol value assigned when the script is created.
- You cannot define a private alias in a script for a batch unit and declare the alias in a script for an operation.
- All parameters must be private local variables.
- Are shared by library scripts for a material and operation linked by a recipe Material icon. (This is the only case in which private symbols function like shared symbols.)
Batch Run Screen Use of Shared and Private Symbols

The Batch Run screen can specify up to three scripts for use with a task. The *chain* and *endchain* instructions allow the use of additional scripts as parent and child scripts. The task can only run one script at a time. Shared and private symbols function as follows:

**Shared symbols:**
- Can pass values in chained scripts between the parent script and child script.
- Can pass values between unit, formula, and procedure scripts.
- In procedure scripts, shared is the default symbol value. When updated to Version 16, variables become shared local variables.
- Cannot be declared for more than one symbol with the same name in different scripts.

**Private symbols:**
- Cannot pass values between scripts.
- Can be used to represent different values for symbols with the same name and type in different scripts. For example, if two scripts are joined by a *chain* instruction, the system regards private local variables named *x* in the two scripts as different variables.
- Can be used to release memory assigned to private variables for use by other symbols declared in subsequent scripts.
Shared and Private Symbol Concept

Figure 1.3.1 shows the differences in how shared and private symbols are manipulated by the Working Recipe. Shared symbols are available to all scripts and are stored in the symbol table specified by the batch unit. Private symbols can only be used by the script in which they are declared. The private symbol table is associated with the task in which the operation is run.

Notice that all scripts can read and write to shared symbols; only the script running in the task can read and write to private symbols.

Figure 1.3.1. Shared and Private Symbol Concept
Defining Shared Symbols

We recommend that you define all shared symbols in one script. When a batch process is run from a Working Recipe, define the shared symbols in the library script associated with the recipe batch unit. When a batch process is run from a Batch Run screen, define the shared symbols in a unit script. If the CP should reboot, the symbol table will restore the symbols significantly faster if the symbols have been defined in only one script.

In ABC Batch, the restore time using one script is faster because the shared symbols are always stored in the Working Recipe symbol table, not in the symbol tables for the tasks in which the scripts are actually run. If shared symbols are defined in more than one script, the Working Recipe symbol table will have to restore the symbols for each operation from disk, one script at a time.

Hex and Binary Values

Constants and local variables can be used to represent values in hex and binary notation. The hex and binary values can simplify binary operations in which hex and binary values are transferred between the ControlBlock and batch script.

Hex values are preceded by the dollar sign prefix ($). Binary values are preceded by the pound sign prefix (#).

Figure 1.3.2 shows the use of hex and binary values.

DEMOWSTRATION OF FUNCTION

shared constant mask3:$0FA9; “Define hex value for constant”
shared local mask5
mask5=#0101_1010_1111_0000; “Assign binary value to variable”

Figure 1.3.2. Hex and Binary Format
Aliases

An alias is a user-defined name for a ControlBlock address or tag. A private alias can be used in different program lines throughout one script. A shared alias can be used by different scripts. Aliases are usually defined as shared aliases in a single script for use by all scripts used in the batch process.

In ABC Batch, private aliases are intended for use in library scripts associated with materials (defined on the Batch Materials Table).

Figure 1.3.3 shows the alias format.
SYNTAX
The general format is:

```
alias x:y
```

where  
x is the name of the alias. An alias name can contain up to 18 alphanumeric characters, the first character of which must be alphabetic. The names of instructions are reserved words and cannot be used as alias names.


y is the tag name or address of the associated ControlBlock variable. If x has been defined in a previous batch script, then y does not need to be specified.

SCRIPT TYPES
Aliases can be used in the following script types:

```
PROC LIBR FORM
RBLC START UNIT
```

SAMPLE INSTRUCTION
```
shared alias fillit:Loader/@a; “Block tag Loader, input register @a”
```

DEMONSTRATION OF FUNCTION
Aliases must be defined in one script before they are declared for use in another script. More than one alias can be defined on one line by separating them with commas.

```
“define aliases”
shared alias setpoint: tic101/B,temp:tic101/A
shared alias blockvlv: v1051/@i
“declare aliases”
shared alias setpoint, temp, blockvlv
“assign values to aliases”
blockvlv=1
setpoint =350/500
until temp>setpoint
```

NOTE: The Batch Monitor screen shows the current address of the alias where the alias is defined in the script. For example, if this unit script runs before the procedure script, the program line is displayed on the Batch Monitor screen as:
```
shared alias setpoint:tic101/B,temp:tic101/A,blockvlv:v1051/@i
```
Therefore, allow room for the extra characters in the line.
Using Aliases

NOTE: Aliases are subject to the following rules:

- All ControlBlock variables use analog values in the range of 0 to 1. For information about scaling ControlBlock values, see the `sclike` and `unsc` instructions.
- Alias instructions can write to and from ControlBlock input registers but can only read ControlBlock output values.
- You only need to specify the ControlBlock address when you first define the alias. Subsequent declarations of the alias in the script do not need to include the ControlBlock address.

Dot Operators

Dot operators allow one alias to access all registers or output for a single ControlBlock. The dot operator specifies which registers or output blocks the alias can access.

Dot operators can be used to manipulate bit vectors. Hex and binary values can be transferred between the script and ControlBlock user flags.

Figure 1.3.4 shows the dot operator format.

SYNTAX
The general format is:

```
x.y
```

where

- \( x \) is the alias name.
- \( y \) is the ControlBlock register.

DEMONSTRATION OF FUNCTION

```
shared alias procval=x=1A-8
procval.B=350/500; “Write value to ControlBlock =1A-8, register B”
procval.@a=0; “Write value to ControlBlock =1A-8, register @a”
procval.@b=1; “Write value to CB =1A-8, register @b”
read=procval.a; “Read ControlBlock =1A-8 output a”
```

Figure 1.3.4. Dot Operator Format
Arrays

An array is a user-defined name for a set of storage locations. RBL includes four different types of arrays. Each type of array is used to store a different type of data.

- **dim array** Stores numeric variables.
- **aliasdim array** Stores ControlBlock addresses or tags.
- **stringdim array** Stores string variables.
- **vdim (virtual) array** Stores numeric variables in an SRU file.

All arrays have the following characteristics:

- An array is configured through an array declaration. The declaration specifies the following array parameters:
  - the type of array
  - the number of storage locations used by the array.
  - the dimensions of the array.

- Arrays are stored in two different ways:
  - The `dim`, `aliasdim`, and `stringdim` arrays are stored in the Coordinator Processor (CP) memory during task execution. When the task completes execution, the array values are cleared from memory.
  - Arrays are limited to 250 arrays per script or by the nonvolatile space available to the Coordinator Processor.
  - A virtual or `vdim` array is stored on a console hard disk in the SRU Data Folder. A virtual array is useful if you require a permanent record of the array.

- An array storage location is a variable that represents a process value.

- A storage location is identified by a unique address called an index. The first storage location always begins at the index 0, which is why the highest index is always one less than the number of storage locations specified in the array declaration. For example, the array declaration `dim FACTORS(9)` creates an array with 9 storage locations indexed from FACTOR(0) through FACTOR(8).

- A one-dimensional array uses one index to identify the address of a storage location. Storage locations are indexed in sequential order.
A two-dimensional array uses two indexes to identify the address of a storage location. The first index gives the array row; the second index gives the array column. The maximum number of rows is called the array height; the maximum number of columns is called the array width. Multiplying the array height \((x)\) times the array width \((y)\) gives the total number of storage locations assigned to the array.

Arrays can be passed between scripts used in parallel operations or in consecutive operations. Procedure scripts that share arrays must be linked by chain instructions.

Figure 1.3.5 compares the address scheme of one-dimensional and two-dimensional arrays.
aliasdim Array

The *aliasdim* instruction creates an array of aliases. The alias array is a user-defined name for a set of ControlBlock addresses or tags. Each address or tag in the array is identified by a unique index number. The number is used during read and write operations to identify the storage locations of alias values in the array.

**NOTE:** If you intend to use dot operators with alias arrays, we recommend that you do not include registers for block tags and addresses assigned to alias arrays. Block addresses and tags without registers offer greater flexibility: For example:

- If an alias address `block(10)` is assigned a block address without a register as follows:
  ```
  block(10):=13A-1
  ```
  Then the following dot operator is allowed:
  ```
  block(10).A.a
  ```

- However, if `block(10)` is assigned a block address with register A as follows:
  ```
  block(10):=13A-1/A
  ```
  Then the following dot operator is not allowed:
  ```
  block(10).a
  ```
even though the assignment in this case attempts to duplicate the assignment:
  ```
  block(10).A.a.
  ```

Figure 1.3.6 shows the format for aliasdim arrays.
The general format of a one-dimensional array:

```
aliasdim w(x)
```

where

- `w` is the name of the alias array.
- `x` is the number of storage locations in the array (maximum of 255). For example, `x=5` creates an array of 5 storage locations indexed 0 to 4.

The general format of a two-dimensional array:

```
aliasdim w(x,y)
```

where

- `w` is the name of the alias array.
- `x` is the number of storage locations in the array height (maximum of 255). For example, `x=5` creates an array of 5 rows indexed 0 to 4.
- `y` is the number of storage locations in the array width (maximum of 255). For example, `x=5` creates an array of 5 columns indexed 0 to 4.

**SCRIPT TYPES**

`aliasdim` can be used in the following script types:

```
PROC LIBR FORM
START UNIT
```

**SAMPLE INSTRUCTION**

- “One-dimensional array”
  - shared aliasdim addr(20)
- “Two-dimensional array”
  - shared aliasdim addr(20,20)

**DEMONSTRATION OF FUNCTION**

- “Define alias”
  - shared aliasdim addr(20)
- “Assign a tag”
  - shared alias addr(1):=1A−3
- “Write values to the alias”
  - addr(1)=1

**Figure 1.3.6. aliasdim Instruction Format**
dim Array

The *dim* instruction creates a variable array. Each variable in the array is identified by a unique index number. The number is used during read and write operations to identify the storage locations of alias values in the array.

Figure 1.3.7 shows the format for the array.
SYNTAX
The general format of a one-dimensional array is:
\[
dim w(x)
\]
where \( w \) is the name of the array.
\( x \) is the number of storage locations in the array width (255 maximum). For example, \( y = 5 \) creates array of 5 storage locations indexed 0 to 4. \( x \) has a default value of 1.

The general format for a two-dimensional array is:
\[
dim w(x,y)
\]
where \( w \) is the array name
\( x \) is the number of rows in the array height (255 maximum). For example, \( x = 5 \) creates an array of 5 rows indexed 0 to 4. \( x \) has a default value of 1.
\( y \) is the number of columns in the array width (255 maximum). For example, \( y = 5 \) creates array of 5 columns indexed 0 to 4. \( y \) has a default value of 1.

SCRIPT TYPES
\textit{dim} can be used in the following script types:
\begin{itemize}
  \item PROC
  \item LIBR
  \item FORM
  \item START
  \item UNIT
\end{itemize}

SAMPLE INSTRUCTION
“one-dimensional array”
shared dim b(10)
“two-dimensional array”
shared dim g(10,10)

DEMONSTRATION OF FUNCTION
local i,j
shared dim g(10,10); “declare two-dimensional array”
\[
\]
\[
\]
\[
\]
\[
\]
for i=0,i<10,i=i+1; “increment x index”
for j=0,j<10,j=j+1; “increment y index”
g(i,j)=time(1)
while g(i,j)==time(1); “loop every 1 second”

Figure 1.3.7. One-Dimensional and Two-Dimensional Arrays
stringdim Array

The stringdim instruction creates a string array. The string array is a user-defined name for a set of character string values. Each character string value in the array is identified by a unique index number. The number is used during read and write operations to identify the storage locations of character string values in the array.

Figure 1.3.8 shows the format for the stringdim array.

### SYNTAX

The general format of a one-dimensional array:

```
stringdim a$(x)
```

where `a$` is the name of the array.

- `x` is the number of storage locations in the array (maximum of 255). For example, `x=5` creates an array of 5 storage locations indexed 0 to 4.

The general format of a two-dimensional array:

```
stringdim a$(x,y)
```

where `a$` is the name of the array.

- `x` is the number of storage locations in the array height (maximum of 255). For example, `x=5` creates an array of 5 rows indexed 0 to 4.
- `y` is the number of storage locations in the array width (maximum of 255). For example, `x=5` creates an array of 5 columns indexed 0 to 4.

### SCRIPT TYPES

stringdim can be used in the following script types:

- `PROC LIBR FORM`
- `START UNIT`

### SAMPLE INSTRUCTION

- “One-dimensional array”
  
  `shared stringdim text$(20)`

- “Two-dimensional array”
  
  `shared stringdim text$(20,20)`

### DEMONSTRATION OF FUNCTION

- “Define constants”
  
  `shared stringdim text$(20)`

- “Write values to constants”
  
  `text$(0)="Operation-A"; text$(1)="Operation-B"`

---

**Figure 1.3.8. stringdim Instruction Format**
Virtual Arrays (Batch Only)

Virtual arrays ($vdim$, $vstringdim$, $rvdim$, and $rvstringdim$) store values in a disk file in the SRU Data Folder. The data file for the virtual array is located on the same disk volume as the script in which the array is declared. The SRU Data Folder is located on a console hard disk. The $vdim$ and $vstringdim$ instructions for the virtual array specify the name of the data file used by the array and the array dimensions. The data file is created when batch writes or reads an array value.

**NOTE:** The virtual array is available only for batch tasks. RBL Controller applications cannot use virtual arrays.

SRU Data Folders have the following characteristics:

- There are two types of virtual arrays:
  - $vdim$: The $vdim$ array is used for floating point values.
  - $vstringdim$: The $vstringdim$ array is used for character string values.

- The SRU Data Folder may be configured as large as necessary to accommodate virtual arrays. The folder size is limited only by the available space on the disk and the 60,000 element array limit. (For information about configuring folders, see DT: 2-1.)

- More than one task may reference the same data file, provided that the array declaration is the same for each task.

- If a filename is not specified in the $vdim$ or $vstringdim$ instruction, the system automatically assigns the name $\$$DEFAULT as data filename for the virtual array.

- An individual string array can have a maximum of 60,000 array elements.

- A $vstringdim$ array element cannot have more than 269 characters. Each array variable requires 30 bytes of volatile memory during use. Since actual data values are stored on disk, the virtual string array requires only 42 bytes of nonvolatile memory.

- A virtual array file assigns only the array disk space for those variables that have values. For example, a $vstringdim$ array $X$ having an index of $X(55,1000,69)$ will not be assigned significant disk space until the array variables are actually used.

- The $varay\_file$ instruction can change files used by virtual arrays. The $on\ varray\_err$ trap can assume control of a script in response to an error in execution of the array.
**vdim Array**

Figure 1.3.9 shows the format for the vdim instruction.

### SYNTAX

The general format of a one-dimensional array is:

```plaintext
vdim w(x):file
```

where

- **w** is the array name.
- **x** is the number of storage locations in the array. For example, `x = 5` creates storage locations indexed 0 to 4.
- **file** is the name of a file in the SRU Data Folder.

The general format for a two-dimensional array is:

```plaintext
vdim w(x,y):file
```

where

- **w** is the array name.
- **x** is the number of rows in the array height. For example, `x = 5` creates 5 array rows indexed 0 to 4.
- **y** is the number of columns in the array height. For example, `y = 5` creates 5 array columns indexed 0 to 4.
- **file** is the name of a file in the SRU Data Folder.

### SCRIPT TYPES

`vdim` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

### SAMPLE INSTRUCTION

- “one-dimensional array”
  shared vdim B(1000):CONFIG
- “two-dimensional array”
  shared vdim G(50,1000):CONFIG
- “Writing a value to a two-dimensional vdim element”
  `G(40,500)=100.1`

---

**Figure 1.3.9. One-Dimensional and Two-Dimensional vdim Arrays**
vstringdim Array

Figure 1.3.10 shows the format for the \textit{vstringdim} instruction.

\begin{tabular}{|l|}
\hline
\textbf{SYNTAX} \\
The general format of a one-dimensional array is:  \\
vstringdim w$(x,z)$:file  \\
where \hspace{0.5cm} w$\$ is the array name (maximum of 18 characters).  \\
x \hspace{0.5cm} is the number of storage locations in the array. For example, \(x = 100\) creates storage locations indexed 0 to 99.  \\
z \hspace{0.5cm} is the maximum number of characters in each character string. Character strings can have up to 269 characters.  \\
file \hspace{0.5cm} is the name of a data file in the SRU Data Folder used by the array (maximum of nine characters).  \\
\hline
The general format for a two-dimensional array is:  \\
vstringdim w$(x,y,z)$:file  \\
where \hspace{0.5cm} w$\$ is the array name (maximum of 18 characters).  \\
x \hspace{0.5cm} is the number of rows in a two-dimensional array. For example, \(x = 5\) creates 5 array rows indexed 0 to 4.  \\
y \hspace{0.5cm} is the number of columns in a two-dimensional array. For example, \(y = 100\) creates 100 array columns indexed 0 to 99.  \\
z \hspace{0.5cm} is the maximum number of characters in each character string. Character strings can have up to 269 characters.  \\
file \hspace{0.5cm} is the name of a data file in the SRU Data Folder (maximum of nine characters).  \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{SCRIPT TYPES} \\
vstringdim can be used in the following script types:  \\
\hspace{0.5cm} PROC \hspace{0.5cm} START \hspace{0.5cm} UNIT  \\
\hspace{0.5cm} LIBR \hspace{0.5cm} FORM  \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{SAMPLE INSTRUCTION} \\
“one-dimensional array with a maximum of 269 characters per array variable”  \\
shared vstringdim B$(1000,269)$:CONFIG  \\
“two-dimensional array with a maximum of 269 characters per array variable”  \\
shared vstringdim G$(55,1000,269)$:CONFIG  \\
“Writing a value to a two-dimensional vstringdim element”  \\
G$(40,500)$=print$($“The level is %d FEET”,LEVEL$)$  \\
\hline
\end{tabular}

\textbf{Figure 1.3.10. One-Dimensional and Two-Dimensional vstringdim Arrays}
rvdim and rvstringdim

Redundant virtual arrays (rvdim and rvstringdim) create redundant virtual array files on the primary and backup volumes. The primary and backup volumes are specified on the Batch Configuration screen. Updates of redundant virtual array files are transparent to the user. For more information on redundant databases, see BA: 2-1.

Figure 1.3.11 shows the format for the rvdim instruction.

**SYNTAX**

The general format for the rvdim instructions is:

```
rvdim w(x,y):file
```

where
- `w` is the array name.
- `x` is the number of rows in the array height. For example, `x = 5` creates 5 array rows indexed 0 to 4.
- `y` is the number of columns in the array width. For example, `y = 5` creates 5 array columns indexed 0 to 4.
- `file` is the name of the redundant and backup files in the SRU Data Folders.

**SCRIPT TYPES**

The rvdim instruction can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**SAMPLE INSTRUCTION**

- “one-dimensional array”
  - shared rvdim B(1000):CONFIG
- “two-dimensional array”
  - shared rvdim G(50,1000):CONFIG
- “Writing a value to a two-dimensional rvdim element”
  - `G(40,500)=100.1`
Figure 1.3.12 shows the format for the *rvstringdim* instruction.

### SYNTAX

The general format for the *rvstringdim* instructions is:

```plaintext
rvstringdim w$(x,y,z):file
```

where
- `w$` is the array name (maximum of 18 characters).
- `x` is the number of rows in the array height. For example, `x = 5` creates 5 array rows indexed 0 to 4.
- `y` is the number of columns in the array width. For example, `y = 5` creates 5 array columns indexed 0 to 4.
- `z` is the maximum number of characters in each character string. Character strings can have up to 269 characters.
- `file` is the name of the redundant and backup files in the SRU Data Folder.

### SCRIPT TYPES

The *rvdim* instruction can be used in the following script types:

- PROC  
- START  
- UNIT  
- LIBR  
- FORM

### SAMPLE INSTRUCTION

- “one-dimensional array”
  - shared rvstringdim B$(1000,269):CONFIG
- “two-dimensional array”
  - shared rvstringdim G$(50,1000,269):CONFIG
- “Writing a value to a two-dimensional rvstringdim element”
  - `G$(40,500)=print$("The level is %d FEET",LEVEL)`
Computing Disk Space for Virtual Arrays

The SRU Data Folder must be configured large enough to accommodate the disk storage requirements of virtual arrays. To determine the total space required for the SRU Data Folder:

- Multiply the array index by 4 bytes (the disk storage required for a floating point value) and add 1% for overhead. This is the minimum space required for the SRU Data Folder. Figure 1.3.13 shows a formula to determine the disk storage in bytes for an array.

- If more than one array share the same file, use the index of the largest array to determine disk space.

- If more than one array file share the same SRU Data Folder, add the space required for each array file to obtain the minimum space required for the folder.

(For additional information about configuring folders, see DT.)

EQUATION FOR CONFIGURING SRU DATA FOLDER

The disk storage requirements for a virtual array is determined by the following equation:

\[ S = I_1 \times I_2 \times 4(\text{bytes}) \times 1.01 \]

where

- \( S \) = Disk and RBL folder space (in bytes) required for the array.
- \( I_1 \) = Number of storage locations in the array width.
- \( I_2 \) = Number of storage locations in the array height. \( I_2 \) is not required for a one-dimensional array.

NOTE: A floating point value requires 4 bytes of disk space plus 1% overhead.

DEMONSTRATION OF EQUATION

shared vdim g(100,100):config

100 x 100 x 4(bytes) x 1.01 = 40,400 bytes of disk space required

Figure 1.3.13. Disk Space Requirements for Virtual Array
Array Functions

Array functions are instructions used to manipulate arrays.

**NOTE:** Array functions generate alarm messages that add to PeerWay message traffic. To prevent overwhelming the PeerWay with messages, allow at least three seconds between array functions. You can use a `sleep(3)` instruction.

array_copy and array_init

The `array_copy` instruction is used to copy array elements into another array. The `array_init` instruction is used to initialize array elements to a specified value.

**NOTE:** The `array_copy` instruction does not copy data for string arrays (`stringdim`), redundant arrays (`rdim` and `rstringdim`), and virtual string arrays (`vstringdim`).
**Performance Improvements**

The `array_copy` and `array_init` instructions can copy or initialize as many as 256 array elements at a time. By contrast, the following `for` loop can only initialize or copy one element at a time:

```plaintext
for i=0,i<500,i=i+1
for j=0,j<500,j=j+1
    Array_A(i,j)=1.1
    Array_B(i,j)=Array_C(i,j)
```

By using the `array_copy` and `array_init` instructions, you can increase performance by a factor of 256 for arrays with row lengths greater or equal to 256 elements or a factor equal to the row length for arrays with row lengths less than 256 elements.
Array Copy and Initialize Concept

Figure 1.3.14 demonstrates an array copy address scheme used by the `array_copy` instruction. Array addresses specify starting and ending dimensions for data copied.

```
rs=array_copy(S1,D1,0,1,2,1,3)
```

For the source array, the first address specifies the beginning row and column of the data copied.

```
rs=array_copy(S1,D1,0,1,1,2,2,1,3)
```

The second address specifies the last row and column of the data copied.

```
rs=array_copy(S1,D1,0,1,1,2,2,1,3)
```

For the destination array, the address specifies the beginning row and column of the copied data.

```
rs=array_copy(S1,D1,0,1,1,2,2,1,3)
```

The `array_init` instruction specifies starting and ending array dimensions of elements that are initialized.

```
rs=array_init(S1,1.1,0,1,1,2,2,1,3)
```
Array copy with 0 offset and no wrap around

rs = array_copy(S1,D1,0,1,1,2,2,1,3)

<table>
<thead>
<tr>
<th>y index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x index</td>
<td>0</td>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3,0</td>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
</tr>
</tbody>
</table>

Source array S1

<table>
<thead>
<tr>
<th>y index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x index</td>
<td>0</td>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3,0</td>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
</tr>
</tbody>
</table>

Destination array D1

Copy: row/columns 1,1 to 2,2
To: row/column 1,3

Figure 1.3.14. Address scheme for array_copy instruction

(continued on next page)
Array copy with 0 offset and wrap around

\[ rs = \text{array_copy}(S1, D1, 0, 1, 1, 2, 2, 1, 4) \]

Source array S1

<table>
<thead>
<tr>
<th>y index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x index</td>
<td>0</td>
<td>0.0, 0.1, 0.2, 0.3, 0.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.0, 1.1, 1.2, 1.3, 1.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.0, 2.1, 2.2, 2.3, 2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.0, 3.1, 3.2, 3.3, 3.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copy: row/columns 1,1 to 2,2
To: row/column 1,4

Destination array D1

<table>
<thead>
<tr>
<th>y index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x index</td>
<td>0</td>
<td>0.0, 0.1, 0.2, 0.3, 0.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.0, 1.1, 1.2, 1.3, 1.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.0, 2.1, 2.2, 2.3, 2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.0, 3.1, 3.2, 3.3, 3.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Array copy with 1 offset and wrap around

\[ rs = \text{array_copy}(S1, D1, 1, 1, 1, 2, 2, 1, 4) \]

Source array S1

<table>
<thead>
<tr>
<th>y index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x index</td>
<td>0</td>
<td>0.0, 0.1, 0.2, 0.3, 0.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.0, 1.1, 1.2, 1.3, 1.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.0, 2.1, 2.2, 2.3, 2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.0, 3.1, 3.2, 3.3, 3.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copy: row/columns 1,1 to 2,2
To: row/column 1,4

Destination array D1

<table>
<thead>
<tr>
<th>y index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x index</td>
<td>0</td>
<td>0.0, 0.1, 0.2, 0.3, 0.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.0, 1.1, 1.2, 1.3, 1.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.0, 2.1, 2.2, 2.3, 2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.0, 3.1, 3.2, 3.3, 3.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.3.14. Address scheme for \textit{array_copy} instructions (continued)
array_copy Format

Figure 1.3.15 shows the format for the *array_copy* instruction.

**SYNTAX**

The general format for the *array_copy* instructions is:

\[ q = \text{array_copy}(r, s, t, u, v, w, x, y, z) \]

where

- \( q \) is a variable used to check the instruction status. A 1 means the instruction was successful; a 0 means it was unsuccessful.
- \( r \) is the source array from which data is copied.
- \( s \) is the destination array to which data is copied.
- \( t \) is the array offset in columns in the destination array. For example, if the column of the destination array is 5 and the offset is 1, copied data begins at column 6.
- \( u \) is the beginning row in the source array from which data is copied.
- \( v \) is the beginning column in the source array from which data is copied.
- \( w \) is the ending row in the source array from which data is copied.
- \( x \) is the ending column in the source array from which data is copied.
- \( y \) is the beginning row in the destination array to which data is copied.
- \( z \) is the beginning column in the destination array to which data is copied.

**SCRIPT TYPES**

*array_copy* instruction can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**DEMONSTRATION OF FUNCTION**

“Copy data for dimensions 1,1 to 5,5 from Source_A to 2,2 in Dest_A”

\[ rs = \text{array_copy}(\text{Source}_A, \text{Dest}_A, 0, 1, 1, 5, 5, 2, 2) \]

Figure 1.3.15. *array_copy* Instruction Format
array_init Format

Figure 1.3.16 shows the format for the array_init instruction.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format for the array_init instructions is:</td>
</tr>
<tr>
<td>q=array_init(r,s,t,u,v,w,x,y,z)</td>
</tr>
<tr>
<td>where q is a variable used to check the instruction status.</td>
</tr>
<tr>
<td>r is array which you want to initialize.</td>
</tr>
<tr>
<td>s is the initialization value.</td>
</tr>
<tr>
<td>t is the array offset in columns. For example, if the start column of the array is 5 and the offset is 1, initialization begins at column 6.</td>
</tr>
<tr>
<td>u is the row at which you want to begin initialization.</td>
</tr>
<tr>
<td>v is the column at which you want to begin initialization.</td>
</tr>
<tr>
<td>w is the row at which you want to end initialization.</td>
</tr>
<tr>
<td>x is the column at which you want to end initialization.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_init instruction can be used in the following script types:</td>
</tr>
<tr>
<td>PROC START UNIT</td>
</tr>
<tr>
<td>LIBR FORM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Initialize array Source_A for dimensions 1,1 to 5,5”</td>
</tr>
<tr>
<td>rs=array_init(Source_A,1.1,0,1,1,5,5)</td>
</tr>
</tbody>
</table>

Figure 1.3.16. array_init Instruction Format
**file_copy**

The `file_copy` instruction is used to copy a virtual array file, virtual array of strings file, or a report file into another file of the same file type. You can copy a file in one disk volume into a file in another disk volume.

**NOTE:**
- The `file_copy` instruction does not copy files for redundant virtual arrays (`rvdim`) or redundant virtual string arrays (`rvstringdim`).
- When copying reports, close the report before copying it.

Figure 1.3.17 shows the format for the `file_copy` instruction.

### SYNTAX

The general format for the `file_copy` instruction is:

```
    u = file_copy(v,"w","x","y","z")
```

where
- `u` is a variable that is used to check instruction status.
- `v` is the folder in which the file resides. Enter one of the following key numbers for the folder type:
  - **Key Number** | **Folder Type**
    - 1 | SRU DATA
    - 2 | REPORTS

- `w` is the source volume in which the folder and original file reside. The volume name can be either a character string or a string variable. Enclose character strings in quotes.

- `x` is the name of the file from which you want to copy. The file name can be either a character string or a string variable. Enclose character strings in quotes.

- `y` is the name of the destination volume in which the copy file resides. The volume name can be either a character string or a string variable. Enclose character strings in quotes.

- `z` is the name of the file to which you want to copy. The file name can be either a character string or a string variable. Enclose character strings in quotes.

### SCRIPT TYPES

`file-copy` can be used in the following script types:

- **PROC**
  - START
  - UNIT

- **LIBR**
  - FORM
file_delete

The `file_delete` instruction deletes a virtual array file, virtual array of strings file, or a report file.

**NOTE:** The `file_delete` instruction does not delete files for redundant virtual arrays (`rvdim`) or redundant virtual string arrays (`rvstringdim`).

Figure 1.3.18 shows the format for the `file_delete` instruction.

### SYNTAX

The general format for the `x` instructions is:

```plaintext
u=file_delete(v,"w","x")
```

where

- `u` is a variable that is used to check instruction status.
- `v` is the folder in which the file resides. Enter one of the following key numbers for the folder type:
  - Key Number | Folder Type
    - 1 | SRU DATA
    - 2 | REPORTS
- `w` is the volume in which the folder and file reside. The volume name can be either a character string or a string variable. Enclose character strings in quotes.
- `x` is the original name of the file that you want to delete. The file name can be either a character string or a string variable. Enclose character strings in quotes.

### SCRIPT TYPES

`file_rename` can be used in the following script types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC</td>
<td>START</td>
</tr>
<tr>
<td>LIBR</td>
<td>FORM</td>
</tr>
</tbody>
</table>

Figure 1.3.18. file_delete Instruction Format
The `file_rename` instruction is used to rename a virtual array file, virtual array of strings file, or a report file.

**NOTE:** The `file_rename` instruction does not delete files for redundant virtual arrays (rvdim) or redundant virtual string arrays (rvstringdim).

Figure 1.3.19 shows the format for the `file_rename` instruction.

### SYNTAX
The general format for the `file_rename` instructions is:

```
u=file_rename(v,"w","x","y")
```

- **u** is a variable that is used to check instruction status.
- **v** is the folder in which the file resides. Enter one of the following key numbers for the folder type:

<table>
<thead>
<tr>
<th>Key Number</th>
<th>Folder Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SRU DATA</td>
</tr>
<tr>
<td>2</td>
<td>REPORTS</td>
</tr>
</tbody>
</table>

- **w** is the volume in which the folder and file reside. The volume name can be either a character string or a string variable. Enclose character strings in quotes.
- **x** is the original name of the file that you want to rename. The file name can be either a character string or a string variable. Enclose character strings in quotes.
- **y** is the new name of the file that you want to rename. The file name can be either a character string or a string variable. Enclose character strings in quotes.

### SCRIPT TYPES
`file_rename` can be used in the following script types:

- **PROC** START UNIT
- **LIBR** FORM

---

**Figure 1.3.19. file_rename Instruction Format**
varray_file

The varray_file instruction changes files used by a virtual array (vdim or vstringdim). After the script executes the varray_file instruction, the virtual array writes to or reads from data in the new file. The array cannot access data in the original file, unless the script executes the varray_file instruction again to change back to the original file.

Figure 1.3.20 shows the format of the varray_file instruction.

**SYNTAX**
The general format for the x instructions is:

\[ x=\text{varray}\_\text{file}("y",z) \]

where \( x \) is a variable used to check the instruction status. Values include:

- 0 The instruction successfully changed files assigned to the virtual array.
- 1 The new file assigned to the virtual array is undefined or has 0 dimensions.
- 2 The new file assigned to the virtual array is the wrong file type.
- 3 The new file assigned to the virtual array (\( \text{vstringdim} \) only) is the wrong file size.
- 4 The new file assigned to the virtual array (\( \text{vstringdim} \) only) is the wrong version.

\( y \) is the name of the new virtual array file.

\( z \) is the name of the virtual array.

**NOTE:** File names can be a character string enclosed in quotes or be represented by a string variable.

**SCRIPT TYPES**

varray_file can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**DEMONSTRATION OF FUNCTION**

shared vdim B(1000):PROC_A
B(10)=5.5
varray_file("PROC_B",B)
“vdim B is now assigned to an SRU file named PROC_B”

Figure 1.3.20. varray_file Instruction Format
A constant is a user-defined name for a floating point number. Constants must be declared before they can be used in a script.

The following two constants are defined by the system. They do not need to be declared:

- PI represents the constant 3.141593.
- EXP represents the natural logarithm constant 2.718282.

**NOTE:** A constant value may be changed by a declaration of the constant in a chained script. If the batch node should reboot, however, the current value may not be recovered.

Figure 1.3.21 shows the format of a constant declaration.
SYNTAX
The general format is:

```
constant a:x
```

where `a` is the name of the constant. A constant name can contain up to 18 alphanumeric characters, the first character of which must be alphabetic. The names of instructions are reserved words and cannot be used as constant names.

`x` is a floating point number. If `x` has been declared in a previous batch script in this task, then `x` does not need to be specified.

SCRIPT TYPES

`constant` can be used in the following script types:

```
PROC LIBR FORM
START UNIT
```

DEMONSTRATION OF FUNCTION

```
shared constant factor:23.47; "Define constant"
shared local total
.
.
if (total*factor)>3000; "use constant in computation"
   gosub PHASE2
.
.
PHASE2:
   "execute phase activities"
return
```

Figure 1.3.21. Constant Format
Global Variables

A global variable is a user-defined name for specific data or a specific value (floating point number). Global variables are used to pass data between tasks. Use varies for each of the following:

**Use in Batch:** A global variable can be shared by any of the 64 batch tasks in a batch node. A global variable must be defined as a shared local variable in the first batch task in the batch node and then declared as a shared global variable in other batch tasks that wish to use that variable. Any task that can access a global variable can read from and write to the global variable. Task 1 must be running before the other tasks can access the global variable.

**Use in RBLC Scripts:** A global variable can be shared by any of the six blocks in an RBL Controller Processor. A global variable must be declared as a local variable in the first RBL block and then as a global variable in other RBL blocks that wish to use that variable. Any task that can access a global variable can read from and write to the variable. Block 1 must be running for other tasks to access the global variable.

Figure 1.3.22 shows the format of a global variable declaration.
The general format is:

\[
\text{global } x
\]

where \( x \) is the name of the local variable declared in task 1. A local variable can contain up to 18 alphanumeric characters, the first character of which must be alphabetic. The names of instructions are reserved words and cannot be used as names for a global variable.

The following script types can be used:

- \text{PROC }
- \text{LIBR }
- \text{FORM }
- \text{RBLC }
- \text{START }
- \text{UNIT }

In the following demonstration, \( \text{sum} \) is first declared as a local variable in \text{TASK1}. Then \text{TASK2} declares \( \text{sum} \) as a global variable and reads from and writes to \( \text{sum} \).

\begin{figure}
\begin{center}
\begin{tikzpicture}
\node at (0,0) {\textbf{TASK 1}};
\node at (1.5,0) {\textbf{TASK 2}};
\node at (0.75,-0.5) {\textbf{TASK 1}};
\node at (2.25,-0.5) {\textbf{TASK 2}};
\draw[->] (0.75,0) -- (2.25,0);
\node at (0,0) {1 \text{ shared local sum}};
\node at (0,1) {2 \text{ sum = 10}};
\node at (0,2) {3 \text{ while sum < 15}};
\node at (0,3) {4 \text{ instruction aaa}};
\node at (0,4) {5 \text{ instruction bbb}};
\node at (2,0) {1 \text{ shared global sum}};
\node at (2,1) {2 \text{ x=sum; “x now equals 10”}};
\node at (2,2) {3 \text{ sum=20; “sum in task 1 now equals 20”}};
\node at (2,3) {4 \text{ “task 1 will now execute instruction bbb”}};
\end{tikzpicture}
\end{center}
\caption{Global Variable Format}
\end{figure}
Local Variables

A local variable is a user-defined name for specific data or a specific value (floating point number). When a local variable is declared, the initial value is 0. There can be up to 250 local variables in one batch task or RBLC script.

NOTE: For specific rules on using shared and private local variables in batch, see the discussion at the beginning of this section.

Figure 1.3.23 shows the format of a local variable declaration.

**SYNTAX**
The general format is:

```
local x
```

where `x` is the name of the variable. A local variable name can contain up to 18 alphanumeric characters, the first character of which must be alphabetic. The names of instructions are reserved words and cannot be used as names for local variables.

**SCRIPT TYPES**
`local` can be used in the following script types:

<table>
<thead>
<tr>
<th>Script Type</th>
<th>PROC</th>
<th>LIBR</th>
<th>FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RBLC</td>
<td>START</td>
<td>UNIT</td>
</tr>
</tbody>
</table>

**SAMPLE INSTRUCTION**

```
shared local TIMER1
```

**DEMONSTRATION OF FUNCTION**

In the following example, a local variable is used to pass data between two scripts.

In a Unit script:

```
shared local temp; “Declare local variable”
temp=550; “Define local variable”
```

In a Procedure script:

```
shared local temp; “Declare local variable again”
prompt
print(*,”The value of temp is %d”,temp); “Display temp value on Batch Input screen”
```

Message that appears on the Batch Input screen:

```
value of temp is 550
```

---

Figure 1.3.23. Local Variable Format
A string variable is a user-defined name for a group of characters followed by a $ character: for example a$, entry$, and input$. Strings are used in different instructions to accomplish various functions. For example, the prompt instruction uses strings to display messages for an operator on a Batch Input screen. The alarm instruction uses strings for generating and clearing alarms. The event instruction uses strings to generate entries for the event lists. The print and flog instructions use strings to copy messages to a file (batch only applications).

Figure 1.3.24 shows the format for string variables.

**SYNTAX**
The general format is:

```
string x$
```

where x is the name of the string variable. You declare the string variable on one line and define its value on a subsequent line.

**SCRIPT TYPES**
`string` can be used in the following script types:

```
PROC   LIBR   FORM
RBLC   START   UNIT
```

**SAMPLE INSTRUCTIONS**
shared string a$

**DEMONSTRATION OF FUNCTION**
shared string a$, b$

a$ = “this string holds the following characters"
b$ (0,39) = “this string can hold exactly 40 characters”
String Characteristics

String variables have the following characteristics:

- String variables are declared with the following format:
  ```
  string a$
  ```

- String variables are assigned values by enclosing the desired string text within quotation marks as shown below:
  ```
  a$="This is a test string"
  ```

- A string variable dynamically changes size to accommodate the assigned characters.

- String values can be defined as a specific size with the function `a$(m,n)`, where `m` is the zero-based index of the first character of the string segment and `n` is the index of the last character. Spaces may be used to fill out a string to a desired number of characters. The following assignment assigns space in the string variable `a$` for a maximum of 50 string characters, though only 46 characters are actually assigned to `a$`. The last four spaces, indexes `a$(46,49)`, are empty.
  ```
  a$(0,49)="This is a test string containing 50 characters."
  ```

- Parts of the string value can be accessed by specifying the character index with the function `a$(m,n)`. For example, using the string variable `a$` defined above:
  - The command `a$(0,0)` accesses the first character of a string variable, which is the character “T”.
  - The command `a$(0,13)` accesses the first 14 characters of the string, which is string segment “This is a test”.
  - The command `a$(37,49)` accesses the characters 37 through 49, which is the segment “50 characters”.

- String values can be assigned to specified segments of the string by specifying the indexes for the string segments as follows:
  ```
  a$(10,10)=x; "Place x in the 10 segment of a$"
  ```

- String values may be copied from one variable to another using an expression such as `a$=b$`.

- String variables may be passed between variables in separate scripts that have been linked through a `chain` instruction. For more information on `chain` and `endchain` instructions, see Section 5 in this chapter.
String values can be cleared from the nonvolatile memory by overwriting string variables with null string values (for example, a$=“ ”). If a task uses a lot of string variables, the nonvolatile memory may become inundated with string values. The accumulation of string values in the memory can interfere with normal batch processing activities. The following example uses a null string to clear a string value from the nonvolatile memory.

    a$ = “ ”, “null string ‘ ’ overwrites a$”

String variables can be displayed on a BFACE graphic object, but must not exceed 10 characters.

**NOTE:** String variables should not exceed 269 characters. Excessively large variables can cause the script to execute slower or exceed available nonvolatile memory.
Hints for Using Symbols

The following are tips that are helpful for using symbols in scripts.

Changing Variables

The following procedure changes a symbol type (such as local to alias or alias to global) without changing the symbol name.

To change the symbol type:

**NOTE:** Do not press `[CTRL-P] or [CTRL-W] until the very end of this procedure.

1. Change the symbol to the desired symbol type. Then move the cursor to the end of that line and press [ENTER] to parse that line and create the new symbol type.

2. Now do a search for all occurrences of the old symbol.
   a. Move the cursor to the search field in the upper left corner of the Batch or RBLC Script screen.
   b. In that field type a slash character (/), followed by the old symbol name, and press [ENTER]. The script will scroll to the next occurrence of the symbol within the file.
      (To search upwards, use a question mark (?) instead of a slash mark.
   c. Move the cursor to the end of the line that contains the old symbol name and press [ENTER]. This will reparse this line, changing the previous reference of the old symbol to a reference of the new symbol type.
   d. Search for the next occurrence of the old symbol name. Move the cursor to the search field, enter a slash (/), and press [ENTER]. The display will remember what pattern it had searched for previously and will search for the next occurrence of the same pattern.

3. Repeat step 2 until each line that referenced the old symbol has been reparsed.

4. Search for the name “Undef” as described in step 2. This will find any undefined phase names or variables, aliases, constants, or array names within the script. Such undefined references will cause a fatal runtime error during script execution. Resolve these undefined references before executing the script.

5. When finished, press [CTRL-W] to write the script to disk.
Making a Script Execute Sooner

Do not execute the local variable, global variable, string, array, alias, or constant definition and declaration statements in a script. The information provided by these statements is included in the symbol table and executing these lines is a waste of time.

- **To skip these lines:**
  1. Put all of these lines at the end of the script after the `finish` instruction.
  
  or

  2. Start the script with a `goto` instruction that bypasses these lines.

Conserving Volatile Memory

Variable names declared in the script reduce the amount of volatile memory available to the task. When declared, the variable names are stored in a symbol table in volatile memory for the duration of task execution. You can conserve volatile memory by using these variable names over again for variable values wherever possible, instead of declaring new variable names for new values.

Comparing Floating Point Values

You should use caution when comparing numbers with floating point values. Floating point numbers can have up to nine decimal places. A Batch Monitor screen can only display up to six decimal places.

It is possible for the Batch Monitor screen to display a floating point value as 5.5 when it is actually 5.5000001. Consequently, the following comparison will not work:

```
alias REGISTER
if REGISTER==5.5
```

However, the following comparison will work:

```
alias REGISTER
if (abs(REGISTER-5.5)<.001)
```
Section 4: RBL String Functions

RBL string functions are instructions that are used to manipulate the values of character string variables in different ways. These functions convert string values to other types of values (numerical, ASCII, or binary) and vice versa. They are also used to perform character counts, string concatenations, or string comparisons.
andbit$ and orbit$

The *andbit* function performs “and” operations on the binary bits of two character bytes. The *orbit* function performs “or” operations on the binary bits of two character bytes. Each instruction returns a single eight bit character value. Figure 1.4.1 shows the andbit and orbit formats. Figure 1.4.2 shows the *andbit* and *orbit* functions.

**NOTE:** If there is more than one character in the string expression, specify the index of the character byte on which to perform the operation (for example, (b$(2,2)). Otherwise, the function will perform operations on all the character bytes in one string (usually the smallest string) and ignore the character bytes in the other string.

### SYNTAX
The general format is:

```
a$=andbit$(b$,c$)
a$=orbit$(b$,c$)
```

where:
- **a$** is a string variable that receives the character byte returned by *andbit* or *orbit*.
- **b$** is the first string expression. If there is more than one character in the string expression, specify the character byte on which to perform the operation (for example, (b$(2,2)).
- **c$** is the first character byte in the string expression. If there is more than one character in the string expression, specify the character byte on which to perform the operation (for example, (c$(128)).

### SCRIPT TYPES
*andbit* and *orbit* can be used in the following script types:

```
PROC      LIBR      FORM
RBLC      START      UNIT
```

*Figure 1.4.1. andbit$ and orbit$ Function Format*
The ASCII bit representation of the first character (3) of one string

0011 0011
0111 0100
0011 0000

The ASCII bit representation of the first character (1) of another string

0011 0011
0111 0100
0111 0111

The result of the andbit function on the first character. The result is the bit representation of the ASCII character 0.

The result of the orbit function on the first character. The result is the bit representation of the ASCII character w.

Figure 1.4.2. andbit$ and orbit$ Functions
The `asc` function converts the first character of a string to the decimal ASCII representation of the character (0-255). You can use either a character string delimited by quotes or a character string variable as the string expression. If you use a character string variable, you can convert characters other than the first character in the string by specifying the character position; for example: `a=asc(b$(2,2))`. Figure 1.4.3 shows the format for the `asc` function.

**SYNTAX**
The general format is:

```
a=asc(b$)
```

where `a` is a variable that receives the ASCII value returned by `asc`.  

`b$` is a string expression.

**SCRIPT TYPES**
`asc` can be used in the following script types:

```
PROC    LIBR   FORM  
RBLC    START   UNIT
```

**SAMPLE FUNCTION**
```
a=asc("b")
a=asc(b$)
```

**DEMONSTRATION OF FUNCTION**
shared local data1
```
data1=asc("a"); "assign to data1 the ascii decimal value of character a, which is 97."
```
byte2flag and flag2byte$

The *byte2flag* function writes two characters of a string variable to the 16 user flags (discrete outputs a to p) of a ControlBlock. The *flag2byte* function reads the 16 user flags (discrete outputs a to p) from the ControlBlock and writes them into two characters of a string variable. You can use ControlBlock logic to test or set flag values.

The user flags a through p represent the bit configuration of the two character bytes in binary notation. Flags a through h indicate the bit configuration of the second character in the character string; flags i through p indicate the bit configuration of the first character in the character string.

Figure 1.4.4 shows the hex and bit representation of two character bytes in user flags. Figure 1.4.5 shows the format for *byte2flag* and *flag2byte* functions. For more information about user flags, see CB: 7-3.

---

![Diagram showing the hex and bit representation of two character bytes in user flags](image)

**Figure 1.4.4. Hex and Bit Representation of Two Character Bytes in User Flags**
SYNTAX
The general format is:
\[ a$ (u,v) = \text{flag2byte$(w,x,y,z)} \]
\[ z = \text{byte2flag(a$(u,v),w,x,y)} \]
where
- \( a$ \) is the character string expression.
- \( u \) is the index of the first character in the string expression. Flags i through p represent the bit configuration of the first character.
- \( v \) is the index of the second character in the string expression. Flags a through h represent the bit configuration of the second character.
- \( w \) is the alias or address for the ControlBlock from which the user flag values are read or written to by the function.
- \( x \) is an integer or variable that offsets the value of the ControlBlock address (parameter \( w \)). The offset address can be less than or greater than the current block address. For example, a value of -2 will change alias: =1a-10 to alias:=1a-8. Zero (0) indicates that the value of the ControlBlock address should not be changed.
- \( y \) is an integer or variable that corresponds to the ControlBlock analog register being read from or written to. The integers 0 and 1 through 15 correspond to registers Q and A through O.
- \( z \) is a local variable that can be checked in the script to see if the function was successful.

DEMONSTRATION OF FUNCTION
The example below shows the use of the \text{byte2flag} and \text{flag2byte} function.
shared string a$, b$
shared alias block:=1a-1/a;
shared local rsb,rsf
.a
.a
.a
.a
a$=\text{"u2"}; \text{“u is hex 55, 2 is hex 32”}
rsb=\text{byte2flag(a$(0,1),block,0,0)}
b$(0,1)=\text{flag2byte$(block,0,0,rsf)}$
alarm(3,1,b$): \text{“alarm prints ‘u2’ ”}
hold

Figure 1.4.5. flag2byte and byte2flag Format
Using byte2flag and flag2byte$

Figure 1.4.6 and Figure 1.4.7 show sample configurations in which byte2flag and flag2byte$ calculate binary equivalents of decimal numbers and vice versa. A simplified ControlBlock configuration shows the use of ControlBlock logic to test or set flag values. User flags in ControlBlock logic display input/output data values in binary notation and transfer the values between the block and the script. (For more information on user flags, see CB: 7.)

- In Figure 1.4.6, the byte2flag instruction writes ASCII values for decimals 18 and 74 as user flags to block =1A-01, register C. C.u causes the block to display the ASCII values in binary notation on Discrete Faceplate screen. Faceplate positions 1 to 8 represent the binary value of 74; positions 9 to 16 represent the binary value of 18.

- In Figure 1.4.7, output step a assigns I/O field input for register D to user flags C.u and Q.u so that D.u input from a remote source can be displayed on the Discrete Faceplate screen using C.u and can be written to the batch script using Q.u. The flag2byte$ instruction reads the 16 user flags for Q.u and writes them as ASCII values to the string variable bits$. Faceplate positions 1 to 8 represent the binary value of hex $BF and decimal 190; positions 9 to 16 represent the binary value of hex value $00 and decimal value 18.
Batch Script AA
shared alias inpt:=1A-01/C
shared alias outpt=1A-01
shared local rs,dec1,dec2
shared string bits$
$
"Convert decimal to ASCII values"
bits$(0,0)=chr$(18)
bits$(1,1)=chr$(74)

"write ASCII values to block flags"
\texttt{byte2flag(bits$(0,1),inpt,0,0)}

"read block flags to ASCII values"
bits$(0,1)=flag2byte$(outpt,0,0,rs)

"Convert ASCII values to decimals"
dec1=asc(bits$(0,0))
dec2=asc(bits$(1,1))
prompt; "Display decimal values"
print(*,"^n Hex $00 =\%d",dec1)
print(*,"^n Hex $BF =\%d",dec2)

RS3: Rosemount Basic Language (RBL)  RBL String Functions
CB CONTINUOUS LINKS

INPUT SOURCE
A ⇒
B ⇒
C ⇒ ENTRY
D ⇒ 2E-12
E ⇒
F ⇒
G ⇒

CB DISCRETE FACEPLATE

Tag Descriptor
Addr=1A-0 1 Function
POS 1 2 3 4 5 6
DISPLAY ⇒ C.u ⇒ ⇒ ⇒ ⇒
Tag
POS 7 8 9 10 11 12
DISPLAY ⇒ ⇒ ⇒ ⇒ ⇒ ⇒
Tag
POS 13 14 15 16 17
DISPLAY ⇒ ⇒ ⇒ ⇒ ⇒
Tag

CB DISCRETE DIAGRAM

Step ⇒ a

ON⇒ Q.u= C.u=D.u
OFF⇒

Batch Script AA
shared alias inpt=1A-01/C
shared alias outpt=1A-01
shared local rs,dec1,dec2
shared string bits$
.
.
.’

‘Convert decimal to ASCII values’
bits$(0,0)=chr$(18)
bits$(1,1)=chr$(74)
.
.

‘write ASCII values to block flags’
byte2flag(bit$(0,1),inpt,0,0)

‘read block flags to ASCII values’
bits$(0,1)=flag2byte$(outpt,0,0,rs)

‘Convert ASCII values to decimals’
dec1=asc(bits$(0,0))
dec2=asc(bits$(1,1))
prompt; ‘Display decimal values’
print(*,”\n Hex $00 =%d”,dec1)
print(*,”\n Hex $BF =%d”,dec2)

Figure 1.4.7. Sample Configuration Using flag2byte$
byte2flag and flag2byte$ Status Values

The `byte2flag` and `flag2byte$` functions return numerical status values that can be checked in the script to see if the function was successful. Table 1.4.1 describes the status values for `byte2flag` and `flag2byte$`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte2flag</td>
<td>1</td>
<td>Character string is empty of values.</td>
</tr>
<tr>
<td>flag2byte$</td>
<td>2</td>
<td>Illegal block number used.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Function successfully executed.</td>
</tr>
</tbody>
</table>
The *chr$* function converts a decimal number (0–255) to its equivalent ASCII character. This function is the inverse of the *asc* function. Figure 1.4.8 shows the format for the *chr$* function.

### SYNTAX
The general format is:

```
a$=chr$(b)
```

where:
- `a$` is a string variable that receives the character value returned by *chr$*.
- `b` is an ASCII numeric value (0–255)

### SCRIPT TYPES
*chr$* can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

### DEMONSTRATION OF FUNCTION
- shared string `a$`
- 
- 
- `a$=chr$(32); “this instruction assigns to a$ the character associated with ASCII value 32.”`

**Figure 1.4.8. chr$ Function Format**
The `getbit` function returns the state (0 or 1) of a specified bit in a character byte. Each character is made up of eight bits. The `getbit` function returns the state of one of the bits of a character. Figure 1.4.9 shows the format for the `getbit` function.

**SYNTAX**
The general format is:

```
a=getbit(b$,c)
```

where

- `a` is variable that receives the state (0 or 1) value returned by `getbit`.
- `b$` is the string expression containing the byte. If a string is specified, the first byte of the string is used.
- `c` is the numeric bit position (0-7).

**DEMONSTRATION OF FUNCTION**

```
shared local holder
shared string char$

char$="abcd"
holder=getbit(char$,4);  "This instruction sets the variable holder equal to"
                      "the state of the 4th bit of the first byte of string char$"
```

**SCRIPT TYPES**

`getbit` can be used in the following script types:

```
PROC LIBR FORM
RBLC START UNIT
```

**Figure 1.4.9. getbit Function Format**
The `setbit$` function sets each of up to eight specified bits in a string byte to a value of 1. If an entire string expression is specified, the byte of the first character is used. Figure 1.4.10 shows the format for the `setbit$` function.

**SYNTAX**
The general format is:

```
a$ = setbit$(b$, 0, 1, 2, 3, 4, 5, 6, 7)
```

where
- `a$` is a string variable that receives the altered character value returned by `setbit$`.
- `b$` is the string expression. The setbit function affects the first character in `b$`.
- `0-7` are the bits that are set in the specified byte. One or more bits may be specified.

**SCRIPT TYPES**
`setbit` can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

**DEMONSTRATION OF FUNCTION**

```
shared local a$
shared string exp$
.
.
.
exp$ = "AB"
display("exp$ = ", exp$); "Displays A -- the value is 41 Hex, 0100 0001 binary.”
a$ = setbit$(exp$, 2)
display("a$ = " , a$); "Displays a -- the value is 61 Hex, 0110 0001 binary.”
```

Figure 1.4.10. `setbit$` Function Format
The `str$` function converts a floating point number to an ASCII numeric string. Figure 1.4.11 shows the format for the `str$` function.

**SYNTAX**
The general format is:
```plaintext
a$=str$(b)
```
where `a$` is a string variable the receives the ASCII numeric string returned by `str$`. `b` is the floating point value to be converted to a string.

**SCRIPT TYPES**
`str$` can be used in the following script types:
- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

**DEMONSTRATION OF FUNCTION**
shared local i
shared string a$

i=1.34
a$=str$(i); “This instruction places the characters 1.34 into the string a$.”

Figure 1.4.11. `str$` Function Format
**strcat$**

The `strcat$` function returns a string that is the result of concatenating a second string to the end of a first string. Figure 1.4.12 shows the format for the `strcat$` function.

**NOTE:** If you attempt to print a concatenated string that contains the ASCII null character "chr(0)", the printer will stop printing text at the ASCII null character.

### SYNTAX

The general format is:

```
a$=strcat$(b$,c$)
```

where
- `a$` is the string variable that receives the concatenated expression returned by `strcat$`.
- `b$` is the first string expression.
- `c$` is the second string expression.

### SCRIPT TYPES

`strcat$` can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

### DEMONSTRATION OF FUNCTION

shared string str1$, str2$, rtrnstr$

```
str1$ ="This string"
str2$="is a test."
rtrnstr$=strcat$(str1$,str2$); "rtrnstr$ now contains the message 'This string is a test'.'
```

**Figure 1.4.12. strcat$ Function Format**
The `strcmp` function compares string expressions. If the strings are equivalent, a 0 is returned. If the strings do not match, a 1 is returned. Figure 1.4.13 shows the format for the `strcmp` function.

**SYNTAX**
The general format is:

```plaintext
a=strcmp(b$,c$)
```

where:
- `a` is a variable that receives the status value returned by `strcat$`. This variable is optional.
- `b$` is the first string expression.
- `c$` is the second string expression.

**SCRIPT TYPES**
`strcmp` can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

**DEMONSTRATION OF FUNCTION**
The example below uses the `if` instruction to compare the string variable `DAY$` with the string “Tuesday”. If the strings match, a 0 is returned; the negation symbol (\~) then assigns a value of 1 to the expression, the `if` instruction is satisfied, and the `gosub` instruction is executed.

```plaintext
shared string day$.

day$="tuesday"
if ~strcmp(day$,"tuesday"); "If strings are not equivalent"
    gosub WASH
    "continue program"

WASH; "phase"
    "perform wash activities"
return
```

Figure 1.4.13. `strcmp` Function Format
The `strlen` function returns the character count, in bytes, of a string expression. Each character in a string expression is represented by one byte.

Figure 1.4.14 shows the format for the `strlen` function.

### SYNTAX

The general format is:

```
a=strlen(b$)
```

where

- `a` is a variable that receives the character count in bytes returned by `strlen`.
- `b$` is the string expression.

### SCRIPT TYPES

`strlen` can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

### DEMONSTRATION OF FUNCTION

```plaintext
shared local cnt
shared string exp$

exp$="This string is 33 characters long"
cnt=strlen(exp$); "This instruction sets the variable cnt to a value of 33."
```

Figure 1.4.14. `strlen` Function Format
The `swab$` function swaps two ASCII bytes when passing the byte values from one string variable to another. The order of the two bytes is transposed in the string variable that receives the bytes.

Figure 1.4.15 shows the format for the `swab$` function.

**SYNTAX**
The general format is:

```
x$ = swab$(y$)
```

where  
- `x$` is the string variable that receives the swapped bytes returned by `swab$`.  
- `y$` is the string variable that passes the two bytes.

**SCRIPT TYPES**
`swab$` can be used in the following script types:

- PROC  
- LIBR  
- FORM  
- RBLC  
- START  
- UNIT

**SAMPLE INSTRUCTION**

```
fourth$ = swab$(third$)
```

**DEMONSTRATION OF FUNCTION**
The example below shows the use of the `swab` function. Two numerical values “59” and “100” are converted to ASCII byte values “;” and “d” and are concatenated together in the string variable `third$`. The values in `third$` are swapped with the values in the character string variable `fourth$`.

- shared string `first$`, `second$`, `third$`, `fourth$`
-  
- “swab function, ‘swap third$ and fourth$’."
- `first$ = chr$(59);`  “Convert ‘59’ to ascii character;”  
- `second$ = chr$(100);`  “Convert ‘100’ to ascii character d”  
- `third$ = strcat$(first$, second$);`  “Combine ; and d in a string”  
- `fourth$ = swab$(third$);`  “Swap the byte order of ; and d”

Figure 1.4.15. `swab$` Function Format
The `val` function converts numeric characters in a string to a floating point number. The input string is scanned until the first digit is found. Once the number is found, each character is read and converted to numeric format until a non-digit character is found.

If the conversion is successful, the returned status is the index of the non-digit character that terminated the conversion. If the conversion is unsuccessful, the returned value is 0. This function is the inverse of the `str` $function. Figure 1.4.16 shows the format for the `val` function.
SYNTAX
The general format is:
\[ a = \text{val}(b\$, c) \]
where 
\( a \) is a variable that receives the index number returned by \text{val}. A zero indicates that the conversion was unsuccessful.
\( b\$ \) is the string variable to be converted to a floating point number.
\( c \) is the floating point variable that receives the floating point number returned by \text{val}.

\textbf{NOTE:} Do not use aliases or instructions that retry as arguments for \( c \). For information on instructions that retry, see Section 1 in this chapter.

SCRIPT TYPES
\textit{val} can be used in the following script types:
- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

SAMPLE INSTRUCTION
\textit{val}(a\$, \text{NUMBER})

DEMONSTRATION OF FUNCTION
The example below converts the string expression \( a\$ \) to a floating point number. The floating point number 4.52 is returned to the variable \text{rtrn1}. The index of the character that terminated the conversion (16) is returned to the variable \text{stat}.

In lines 3 and 4, the string is scanned again for a numeric value, beginning with the termination character (16th character).

```plaintext
shared string a$
shared local stat,rtrn1,rtrn2,nextloc
a$="pressure is 4.52 and 3.33."
stat=val(a$,rtrn1)
nextloc=stat
stat=val(a$(nextloc,nextloc),rtrn2)
```

Figure 1.4.16. \textit{val} Function Format
Section 5: Branching and Conditional Instructions

This section describes branching and conditional instructions. Branching and conditional instructions direct the execution of the batch or RBLC script.

- Branching instructions (\texttt{goto, gosub, start, return, chain}, and \texttt{endchain}) direct execution of the script to a specified phase, label, or script.
- Conditional instructions (\texttt{until, while, if, else, elseif}, and \texttt{for}) direct execution of the script if a specified logic condition is true or false.
chain and endchain

The *chain* instruction suspends the execution of a procedure script, called the parent script, and starts executing another procedure script named by the *chain* instruction, the child script. The child script can use a *chain* instruction to chain to a third script, and so on. Chained scripts must be on the same disk volume as the parent script.

The *endchain* instruction in a child script is optional and always brings program execution of the original parent script to the line after the *chain* instruction. The *endchain* instruction maintains indentation and on traps in the original parent script regardless of how many child scripts have been chained.

**In Batch Scripts:** In batch scripts, chaining passes local and global variables, aliases, and constants from one script to another. If the variables are declared in both scripts, any changes to the variables in a child script are effective in any subsequent child and parent scripts.

**In RBL Scripts:** In RBL scripts, chaining does not pass local variables and aliases from one script to another. Values in the parent script are initialized to zero when chaining to a child script and so the child script begins program execution with variable values initialized to zero. When program execution returns to the parent script, local variables and aliases are restored to the parent script as they were at the time of the first *chain* instruction.

**NOTE:** *chain* and *endchain* instructions are subject to the following rules:

- The *chain* and *endchain* instructions can only be used in batch procedure scripts. They will not parse in ABC Batch library or start scripts.

- Open print files and global variables are not altered by chaining. If the parent script has an open print file, the child script has access to the file without having to reopen it. Figure 1.5.1 shows the *chain* and *endchain* instruction formats.

- If the script being chained does not exist on the “Primary” or “Backup” volume, the *chain* instruction generates a soft runtime error and puts the task into a hold state. To resume processing, the parent script should contain an *on chain_err* instruction (see the *on* instruction section). If the requested script is not found, the task branches to the *on chain_err* and resumes normal processing.

- The *chain* instruction refers to the primary and backup volumes on the Batch Configuration screen to find a file.
If the Coordinator Processor reboots, the task will recover the most recently defined value of the constant or alias.

**SYNTAX**

The general format for the chain instruction is:

```
chain("x","y")
```

where

- `x` is the name of the RBL file.
- `y` is the name of the child script. The file and script come from the disk specified on the Batch Run screen as the “Primary” or “Backup” for whatever unit, formula, or procedure script is chaining.

**NOTE:** You can use either text strings or string variables to represent file and script names. Text strings must be enclosed in quotes.

The general format for the `endchain` instruction in a child script is:

```
endchain
```

**SCRIPT TYPES**

`chain` and `endchain` can be used in the following script types:

- PROC
- UNIT
- FORM

**SAMPLE INSTRUCTION**

Parent script or child script:

```
chain ("LIBRARY","PURGE"); "Direct execution to child script"
```

Child script:

```
endchain; "Return execution to parent script"
```

**DEMONSTRATION OF FUNCTION**

Parent script: FILL

```
1  FILL:
2  type=4
3  chain ("LIBRARY","PURGE")
4  if error==0
5  start fill
```

Child script: PURGE

```
1  if type
2  error=0
3  "return to parent script"
4  endchain
```

---

**Figure 1.5.1. chain and endchain Formats**
Using Shared Aliases and Constants in Chained Scripts

Shared constants and aliases function as follows in chained scripts. (You cannot use `chain` and `endchain` instructions in ABC Batch scripts.)

**WARNING**

Read this information thoroughly before using private aliases in chained scripts.

- A shared constant or alias can be defined in either the parent script or the child script for use in both scripts.
- The definition of a shared constant or alias can be changed in the child script. When the parent script resumes execution, the constant or alias is restored to its original definition before chaining, as shown in Figure 1.5.2

```
PARENT_1  Proc:
  “Define valve for VC2”
  shared alias valve:VC2/@i:
  valve = 1; “open valve VC2”
  sleep(5)
  chain(“LIBRARY”,“CHILD_1”) valve=1; “open valve VC2”

  CHILD_1  Proc:
  “redefine valve for VC4”
  shared alias valve:VC4/@i;
  valve=1; “open valve VC4”
  sleep(5)
  endchain
```

Figure 1.5.2. Define alias Value in Child Script and Redefine Value in Parent Script

- If an alias or constant is defined in the child script but not in the parent script, the parent script uses the definition from the child script after chaining is completed, as shown in Figure 1.5.3
Figure 1.5.3. Declare alias in Parent Script and Redefine alias Value in Child Script
Using Private Aliases and Constants in Chained Scripts

- Private constants or aliases cannot be used to pass values between the parent script and child script.
- You can only define private constants or aliases in the script in which it is used.
- The child script cannot change the definition of a private constant or alias defined in the parent script.
- The symbol table does not retain definitions and values for private constant or alias in a child script after execution returns to the parent script.
- When a parent script chains to a child script, all private symbol values for the parent script are cleared from the symbol table, freeing up memory for use by the child script. When execution returns to the parent script, previously cleared values for private constants and private aliases in the parent script are restored to the symbol table.

**NOTE:** Private local variables in the parent script do not restore their original values after chaining. These values are lost, as shown in Figure 1.5.4.
PARENT SCRIPT

“Parent script chains to child script”
private local counter
private alias block1
private alias block2: =13A-2
shared alias block3
shared alias block4: =13A-4

counter=100
chain(“file1”,“child”)

“Chaining is complete at this point.”
“aliases now have the following values”
“block1 is not defined”
“block2 is defined as =13A-2”
“block3 is defined as =13A-53”
“block4 is defined as =13A-4”
“counter value is now 0”

CHILD SCRIPT

“Child script chained to the parent script”
private alias block1: =13A-51
private alias block2: =13A-52
shared alias block3:=13A-53
shared alias block4: =13A-54
endchain

Figure 1.5.4. Comparison of Shared and Private Symbols in Chained Scripts
The **for** instruction is used to execute a series of program lines a specified number of times. The **for** instruction begins by executing an initial condition expression. The initial condition expression is executed only once. Then the **for** instruction execution cycle begins by evaluating a test expression. If the expression is true, all lines indented under the **for** instruction are executed, followed by the control expression. Then the cycle begins again by evaluating the test expression. The cycle continues until the test expression becomes false. At that point, execution of the script continues at the first line that is not indented under the **for** instruction.

An expression is false if the expression<=0.
An expression is true if the expression>0.

**NOTE:**

- If you delete the **for** instruction when editing a script, all lines indented underneath it lose their indentation when the script is parsed. To prevent loss of indentation, make changes to the instruction text without deleting the line.

- A blank line must follow the last line in a sequence of indented lines.

**CAUTION**

Unless slowed with a sleep instruction, a **for** loop can overwhelm the ControlFile and PeerWay with messages.

Figure 1.5.5 shows the format for the **for** instruction.
SYNTAX
The general format is:

```
for x,y,z
```

where

- **x** is the initial condition expression. x is executed one time before the loop is entered.
- **y** is the test expression. y is executed each time the for instruction execution cycle begins. If y is true, the indented lines under the for instruction are executed. If y is false, the next line not indented under the for instruction is executed.
- **z** is the control expression. z is executed after the indented lines of the for instruction.

SCRIPT TYPES

```
for can be used in the following script types:
  PROC    LIBR    FORM
  RBLC    START    UNIT
```

SAMPLE INSTRUCTION

```
for job=0,job<11,job=job+1
```

DEMONSTRATION OF FUNCTION

The example below shows the use of a for instruction. The first time that the script executes the for instruction, the variable **job** is set to zero. If the test expression **job<11** is true, then the indented instruction **aaa** and **bbb** are executed. The control expression **job=job+1** is then executed. The for instruction execution cycle then begins again with the evaluation of the test expression **job<11**.

If the test expression **job<11** is false, then the non-indented script line **ccc** is executed.

```
  shared local job
  
  
  
  for job=0,job<11,job=job+1
  "execute for loop 10 times"
  instruction aaa
  instruction bbb
  instruction ccc
```

Figure 1.5.5. for Instruction Format
The `gosub` instruction directs execution of the script to one of three types of locations in the script:

- A phase name in the script.
- A label name in the beginning of the script before the first phase has been declared.
- A label name in the same phase as the `gosub` instruction.

(For a description of phases and labels, see Section 1.)

The `gosub` instruction is typically used to direct execution of the script to a location in the script that is used more than once. Instead of repeating the program lines, they are grouped together and identified with a phase name or label name. When the group of lines is needed in the script, a `gosub` instruction is used to direct execution to the phase or label name. The lines introduced by the phase or label end with a `return` instruction so that execution returns to the line after the `gosub` instruction.

When a `gosub` directs execution to another phase in the script, the `on alarm` instructions for the branching phase and the destination phase are both in effect. The "Major" status field on the Batch Run screen shows the location of the original phase. The "Detail" status field shows the location of the new phase branched to by the `gosub` instruction.

Figure 1.5.6 shows the `gosub` instruction format.

---

**SYNTAX**

The general format is:

```
gosub x
```

where `x` is a phase or label name in the script.

**SCRIPT TYPES**

`gosub` can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>LIBR</th>
<th>FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBLC</td>
<td>START</td>
<td>UNIT</td>
</tr>
</tbody>
</table>

**SAMPLE INSTRUCTION**

- `gosub _LABEL1; "Direct script to a label"
- `gosub PHASE1: "Direct script to a phase"

Figure 1.5.6. `gosub` Instruction Format
DEMONSTRATION OF FUNCTION

The example below shows the use of the `gosub` and `return` instructions within a phase. If the expression `pres<45` is true, the `gosub` instruction directs execution of the script to the label `_label1`. The `return` instruction in `_label1` returns execution to the line after the `gosub` instruction.

```plaintext
PHASE1:
  shared local pres
  pres=40
  instruction aaa
  if pres<45
    gosub _LABEL1
  else
    instruction bbb
  instruction ccc
.
.
._LABEL1:
  "perform phase activities"
  return
```

The example below shows the use of the `gosub` instruction between two phases. If the expression `pres<45` is true, the `gosub` instruction directs execution of the script to the phase name `phase2`. The `return` instruction returns execution to the line after the `gosub` instruction in `phase1`.

```plaintext
PHASE1:
  shared local pres
  if pres<45
    gosub PHASE2
  else
    instruction aaa
  instruction bbb
.
.
PHASE2:
  "perform phase activities"
  return
```

Figure 1.5.6. `gosub` Instruction Format (continued)
The `goto` instruction directs execution of the script to one of two types of locations in the script:

- A label name in the same phase as the `goto` instruction.
- A label name in the beginning of the script before the first phase has been declared.

For a description of phases and labels, see Section 1 in this manual.

The specified label must exist before it is used by the `goto` instruction. Figure 1.5.7 shows the `goto` instruction format.
DEMONSTRATION OF FUNCTION
The example below shows the use of the `goto` instruction. If the expression `pres<45` is true, the `goto` instruction directs execution of the script to the label name `_LABEL1`.

PHASE1:
- shared local `pres`
- `pres=0`
- “perform phase activities”
- `_LABEL1; “label subroutine”`
- if `pres<45`
  - `pres=pres+5; “increment press by 5”`
  - `goto _LABEL1; “return to label and execute subroutine again”`
- else
  - instruction `aaa`
  - instruction `bbb`

Figure 1.5.7. goto Instruction Format
if, elseif, and else

The if, elseif, and else instructions are used to execute program lines in a script conditionally.

NOTE:
- If you delete an if, elseif, or else instruction when editing a script, all lines indented underneath it lose their indentation when the script is parsed. To prevent loss of indentation, make changes to the instruction text without deleting the line.
- A blank line must follow the last line in a sequence of indented lines.

if

The if instruction evaluates an expression. If the expression is true, all lines indented under the if instruction are executed. If the expression is false, execution of the script continues at the first line that is not indented under the if instruction.

An expression is false if the expression<=0.
An expression is true if the expression>0.

Figure 1.5.8 shows the if instruction format.
SYNTAX
The general format is:

\[
\text{if } x \\
\text{where } x \text{ is the test expression.}
\]

SCRIPT TYPES
if can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>LIBR</th>
<th>FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBLC</td>
<td>START</td>
<td>UNIT</td>
</tr>
</tbody>
</table>

SAMPLE INSTRUCTION
if stpt<55

DEMONSTRATION OF FUNCTION
The example below shows the use of an if instruction. If the test expression \( stpt<55 \) is true, then instructions \textit{aaa} and \textit{bbb} are executed, followed by instruction \textit{ccc}. If the test expression is false, then script execution continues at instruction \textit{ccc}.

\[
\text{if } stpt<55 \\
\text{instruction aaa} \\
\text{instruction bbb} \\
\text{instruction ccc}
\]

Figure 1.5.8. if Instruction Format
else

The *else* instruction is used with the *if* instruction to execute a group of program lines conditionally if the test expression in the *if* instruction is false.

If the expression is false, all lines indented under the *else* instruction are executed. If the expression is true, execution of the script continues at the first line that is not indented under the *else* instruction.

An expression is false if the expression<=0. An expression is true if the expression>0.

Figure 1.5.9 shows the *else* instruction format.
SYNTAX
The general format is:

```
if x
    script line
    script line
else
    script line
    script line
where x is the test expression.
```

SCRIPT TYPES
`else` can be used in the following script types:

```
PROC LIBR FORM
RBLC START UNIT
```

SAMPLE INSTRUCTION
```
if flow>120; start DRAIN_1
else start DRAIN_2
```

DEMONSTRATION OF FUNCTION
The example below shows the use of the `if` and `else` instructions. If the text expression `flow>120` is true, then instruction `aaa` is executed, followed by instruction `ccc`.

If the test expression is false, then instruction `bbb` is executed, followed by instruction `ccc`.
```
if flow>120
    instruction aaa
else
    instruction bbb
instruction ccc
```

Figure 1.5.9. else Instruction Format
The *elseif* instruction is used with the *if* instruction to provide more than one conditional test. Program lines indented under an *elseif* instruction are executed if all previous related *if* and *elseif* instructions are false, and if the test expression in the *elseif* instruction is true. If all previous related *if* and *elseif* instructions are false and this *elseif* instruction is false, execution of the script continues at the first line that is not indented under this *elseif* instruction.

An expression is false if the expression \( \leq 0 \).
An expression is true if the expression \( > 0 \).

Figure 1.5.10 shows the *elseif* instruction format.
**SYNTAX**
The general format is:

```plaintext
if x
    script line
elseif y
    script line
where x is the test expression of the if instruction.
y is the test expression of the elseif instruction.
```

**SCRIPT TYPES**

`elseif` can be used in the following script types:

- PROC LIBR FORM
- RBLC START UNIT

**SAMPLE INSTRUCTION**

```plaintext
if temp<78
    instruction aaa
elseif temp<100
    instruction bbb
else temp<144
    instruction ccc
instruction ddd
```

**DEMONSTRATION OF FUNCTION**
The example below shows the use of `if`, `elseif`, and `else` instructions. If all three of the text expression are false, then only `instruction ddd` is executed.

```plaintext
if temp<78
    instruction aaa
elseif temp<100
    instruction bbb
else temp<144
    instruction ccc
instruction ddd
```

*Figure 1.5.10. elseif Instruction Format*
The `return` instruction is used with the `gosub` instruction to return execution of the script to the program line after the `gosub` instruction. The `return` instruction is also used with the `on` instruction to return execution of the script to the line that was executing when the `on` instruction was triggered.

Refer to the `gosub` instruction and `on` instruction sections for more information. Figure 1.5.11 shows the `return` instruction format.

**SYNTAX**

The general format is:

```
return
```

**SCRIPT TYPES**

`return` can be used in the following script types:

```
PROC    LIBR    FORM
RBLC    START    UNIT
```

**DEMONSTRATION OF FUNCTION**

```
gosub PHASE1
hold
PHASE1:
"execute phase activities"
return; "return to first program line after gosub"
```

*Figure 1.5.11. return Instruction Format*
**start**

The *start* instruction directs execution of the script to a phase name in the same script. The specified phase name must exist before it is used by the *start* instruction. Figure 1.5.12 shows the *start* instruction format.

### SYNTAX

The general format is:

```
start x
```

where `x` is a phase name.

### SCRIPT TYPES

*start* can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

### SAMPLE INSTRUCTION

```
start PHASE1
```

### DEMONSTRATION OF FUNCTION

The example below shows the use of the *start* instruction. If the expression `pres<45` is true, the *start* instruction directs execution of the script to the phase name `PHASE1`.

```
PHASE1:
    "perform phase activities"
    
    shared local pres
    "perform phase activities"
    
    if pres<45
        start PHASE1; "expression is true; start phase"
    else
        sleep(5)
    finish
```

*Figure 1.5.12. start Instruction Format*
The *until* instruction evaluates an expression. If the expression is false, all lines indented under the *until* instruction are executed. Then, the *until* instruction is executed again. This cycle continues until the expression in the *until* instruction becomes true. At that point, execution of the script continues at the first line that is not indented under the *until* instruction.

An expression is false if the expression $\leq 0$.
An expression is true if the expression $> 0$.

**NOTE:**

- If you delete the *until* instruction when editing a script, all lines indented underneath it will lose their indentation when the script is parsed. To prevent loss of indentation, make changes to the instruction text without deleting the line.
- A blank line must follow the last line in a sequence of indented lines.

**CAUTION**

Unless slowed with a sleep instruction, an *until* loop can overwhelm the ControlFile and PeerWay with messages.

Figure 1.5.13 shows the format for the *until* instruction.
SYNTAX
The general format is:

```
until x

where x is the test expression.
```

SCRIPT TYPES
`until` can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

SAMPLE INSTRUCTION
```
until total>5
```

DEMONSTRATION OF FUNCTION
In the example below, if `total>5` is false, then the indented `instruction aaa` is executed. The `until` instruction and all indented program lines immediately following the `until` instruction are continually executed until the test expression is true.

If `total>5` is true, then program execution continues at the non-indented `instruction bbb`.

```
until total>5
  instruction aaa
  sleep(2)
instruction bbb
```

Figure 1.5.13. until Instruction Format
while

The *while* instruction evaluates an expression. If the expression is true, all lines indented under the *while* instruction are executed. Then, the *while* instruction is executed again. This cycle continues until the expression in the *while* instruction becomes false. At that point, execution of the script continues at the first line that is not indented under the *while* instruction.

An expression is false if the expression <= 0.
An expression is true if the expression > 0.

**NOTE:**
- If you delete the *while* instruction when editing a script, all lines indented underneath it lose their indentation when the script is parsed. To prevent loss of indentation, make changes to the instruction text without deleting the line.
- A blank line must follow the last line in a sequence of indented lines.

**CAUTION**

Unless slowed with a sleep instruction, a *while* loop can overwhelm the ControlFile and PeerWay with messages.

Figure 1.5.14 shows the format for the *while* instruction.
### Syntax
The general format is:

```plaintext
while x
    where x is the test expression.
```

### Script Types
`while` can be used in the following script types:

- PROC  LIBR  FORM
- RBLC  START  UNIT

### Sample Instruction

```plaintext
while total>5
```

### Demonstration of Function
In the example below, if `total>5` is true, then `instruction aaa` is executed. The `while` instruction and all indented program lines immediately following the `while` instruction are continually executed until the test expression is false.

If `total>5` is false, then execution continues at `instruction bbb`.

```plaintext
while total>5
    instruction aaa
    sleep(2)
    instruction bbb
```

---

**Figure 1.5.14. while Instruction Format**
**Special While Cases**

Two special cases of the `while` instruction control the execution of the script while special events occur. These special events include acquiring a slave task or backing up a task from the nonvolatile memory. If the event in the `while` expression is true, all lines indented under it are executed. The `while` loop will be executed repeatedly as long as the special event is true.

Table 1.5.1 shows the `while` instruction format.

<table>
<thead>
<tr>
<th><code>while</code> Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while acquire(x,y)</code></td>
<td>Executes a slave task statement and returns a time value for the task. If a timeout value of 0 is returned, the <code>while</code> statement discontinues the loop.</td>
</tr>
</tbody>
</table>
| `while nobackup`                   |Suspends a task until the task can be backed up to the nonvolatile memory. The system will suspend backup if AC power to the nonvolatile memory falls below 20 volts. The `while nobackup` instruction is typically used in an `on no_backup` trap to suspend the trap. The `on no_backup` trap trips if nonvolatile memory falls below 20 volts. For example:  
  
  `on no_backup`  
  `while nobackup`  
  
  **NOTE:** You can use the `backup` instruction in place of the `while nobackup` instruction to suspend the task until a backup can be performed. |
Section 6:
Batch Management Instructions

Batch management instructions control or monitor the execution of batch tasks in several ways by:

- Changing the run time states of batch tasks.
- Transferring control of script execution from one nesting level to another.
- Displaying time values associated with task execution.
- Incrementing Batch IDs.
- Backing up symbol values to nonvolatile memory.

**NOTE:** In ABC Batch, runtime instructions are considered either global to the recipe or local to the task.

- Global runtime instructions affect the runtime states of all tasks used by the recipe.
- Local runtime instructions only affect the runtime state of the task running the script in which they are included. Other tasks used by the recipe are not affected.
abort

Triggers the execution of an on abort trap. If there is no on abort trap, the abort instructions will not affect execution of the batch script. Figure 1.6.1 shows the format of the abort instruction.

<table>
<thead>
<tr>
<th>SYNTAX AND Sample Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is: abort</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort can be used in the following script types:</td>
</tr>
<tr>
<td>PROC   LIBR   FORM</td>
</tr>
<tr>
<td>RBLC   START   UNIT</td>
</tr>
</tbody>
</table>

Figure 1.6.1. abort Instruction Format
The `abort_task` instruction triggers the execution of an `on abort` trap in a script running in another task. If there is no `on abort` trap, the `abort_task` instruction will not affect execution of the batch script. Figure 1.6.2 shows the format for the `abort_task` instruction.

**SYNTAX**

The general format is:

```
abort_task("x")
```

where `x` is the Batch Run screen tag or PeerWay address of a batch task. You can use a character string or string variable for the task name (for example, a$).

The PeerWay address consists of the ControlFile address and task address:

- **ControlFile address 77**
  (If omitted, `abort_task` uses current ControlFile address.)
- **Task address 02**
  (Must have two digits or the instruction will fail.)

**SCRIPT TYPES**

`abort_task` can be used in the following script types:

- PROC  START  UNIT
- LIBR  FORM

**SAMPLE INSTRUCTION**

Using the Batch Run screen tag:

```
abort_task("PROCESS")
```

Using the PeerWay address:

```
abort_task("7702")
```

Using a string variable:

```
a$="7702" or a$="PROCESS"
abort_task(a$)
```

---

**Figure 1.6.2. abort_task Instruction Format**
The *backup* instruction causes the program execution to hold until it is backed up to the Nonvolatile Memory card. See Figure 1.6.3 for the *backup* instruction format.

For general information on the Nonvolatile Memory card, see OV: 1–2. For information on how nonvolatile memory is used in batch tasks, see BA: 1–4.

<table>
<thead>
<tr>
<th>SYNTAX AND SAMPLE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>backup</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>backup</em> can be used in the following script types:</td>
</tr>
<tr>
<td>PROC</td>
</tr>
<tr>
<td>LIBR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup;; &quot;Phase sets ingredient load parameters&quot;</td>
</tr>
<tr>
<td>shared local target</td>
</tr>
<tr>
<td>target=42</td>
</tr>
<tr>
<td>backup;; &quot;Wait for setup to be backed up to nonvolatile memory.&quot;</td>
</tr>
</tbody>
</table>

**Figure 1.6.3. backup Instruction Format**
The *begin_task* instruction starts a task that is in the IDLE or finished state. Figure 1.6.4 shows the format of the *begin_task* instruction.

**SYNTAX**

The general format is:

```
begin_task("x")
```

where `x` is the Batch Run screen tag or PeerWay address of a batch task. You can use a character string or string variable for the task name (for example, `a$`).

The PeerWay address consist of the ControlFile address and task address:

- **ControlFile address 77**
  (If omitted, *begin_task* uses current ControlFile address.)
- **Task address 02**
  (Must have two digits or the instruction will fail.)

**SCRIPT TYPES**

*begin_task* can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

**SAMPLE INSTRUCTION**

Using the Batch Run screen tag:

```
begin_task("PROCESS")
```

Using the PeerWay address:

```
begin_task("7702")
```

Using a string variable:

```
a$="7702" or a$="PROCESS"
begin_task(a$)
```

---

**Figure 1.6.4. begin_task Instruction Format**
break

The *break* instruction transfers control from an indented set of program lines in a *for*, *while*, or *until* instruction out to the next level of indentation.

Figure 1.6.5 shows the format of the *break* instruction.

<table>
<thead>
<tr>
<th>SYNTAX AND SAMPLE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td>break</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>break</em> can be used in the following script types:</td>
</tr>
<tr>
<td>PROC   START   UNIT</td>
</tr>
<tr>
<td>LIBR   FORM</td>
</tr>
</tbody>
</table>

*Figure 1.6.5. break Instruction Format*
**bumpid**

The *bumpid* instruction increments the last character of the batch ID string on the Batch Run screen. The batch ID string can be used to track batches of product, similar to a serial number. (For more information about the batch ID string on the Batch Run screen, see BA: 3-1.)

The *bumpid* instruction also resets the batch timer (specified by the “BTIME” field on the process graphics batch faceplate).

Figure 1.6.6 shows the *bumpid* instruction format.

---

**SYNTAX AND SAMPLE INSTRUCTION**

The general format is:

```
  bumpid
```

---

**SCRIPT TYPES**

*bumpid* can be used in the following script types:

- PROC
- UNIT
- FORM

---

**DEMONSTRATION OF FUNCTION**

```
phase1:: "Phase starts script instructions"
bumpid
open(1,CCBOOT,REPRT)
print(1,"Batch ID is now %ID")
```

---

Figure 1.6.6. bumpid Instruction Format
continue

The *continue* instruction transfers control from the *continue* program line to the *for* or the *while* program line. Generally, *continue* is used with a *if* instruction.

**NOTE:** If a *continue* instruction is used in a *while*, *until*, or *for* loop, the first line after the loop must be blank. If the first line after the loop contains an instruction, execution will jump from the *continue* instruction to the first line after the loop, discontinuing execution of the loop.

Figure 1.6.7 shows the *continue* instruction format.
SYNTAX AND SAMPLE INSTRUCTION
The general format is:
continue

SCRIPT TYPES
continue can be used in the following script types:
- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

DEMONSTRATION OF FUNCTION
The example below shows the use of a continue instruction. The while instruction executes script lines while total is less than 400. If the if condition counter==20 is true, the continue instruction directs execution of the script to the beginning of the while loop.

shared alias total:TOT-1Z; "define alias"
.
.
.
shared alias total; "declare alias"
shared local counter; "declare variable"
counter=5
.
.
while total<400
    instruction aaa
    counter=(total-380); "double check block value"
    if counter==20; continue; "Continue execution on the while instruction line"
    instruction bbb
    sleep(2)

instruction ccc

Figure 1.6.7. continue Instruction Format
The `cont_task` instruction continues a task that is in the Halt state. It has no effect if the task is in any other state.

Figure 1.6.8 shows the format for the `cont_task` instruction.

**SYNTAX**

The general format is:

```
cont_task("x")
```

where `x` is the Batch Run screen tag or PeerWay address of a batch task. You can use a character string or string variable for the task name (for example, `a$`)

The PeerWay address consists of the ControlFile address and task address:

7702 or 77:02

**SCRIPT TYPES**

`cont_task` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**SAMPLE INSTRUCTION**

Using the Batch Run screen tag:

```
cont_task("PROCESS")
```

Using the PeerWay address:

```
cont_task("7702")
```

Using a string variable:

```
a$="7702" or a$="PROCESS"
```

```
cont_task(a$)
```

---

**Figure 1.6.8. cont_task Instruction Format**
dishold and enhold

The *dishold* instruction disables a “hold” command in order to prevent an operator from holding the execution of the recipe. The *enhold* instruction enables the “hold” command that has been disabled. The “hold” command can be issued by the “hold” field on the Batch Monitor screen, “Hold” field on a BFACE process graphic, or RBL *hold* instruction in a script.

**NOTE:** If a “hold” command is disabled by the *dishold* instruction, the *on hold* instruction will not execute. However, if the script later executes an *enhold* instruction, the *on hold* instruction will execute at that time.

Figure 1.6.9 shows the *dishold* and *enhold* instruction format.

### SYNTAX
The general format is:

- dishold
- enhold

### SCRIPT TYPES
*dishold* and *enhold* can be used in the following script types:

- PROC START UNIT
- LIBR FORM

### DEMONSTRATION OF FUNCTION

- on hold
- while holding
- instruction aaa
- sleep(2)

**PHASE_1:**
- dishold; “hold field and on hold instruction disabled”
- “Perform phase activities”

**PHASE_2:**
- enhold; “hold field and on hold instruction enabled”
- “Perform phase activities”

---

**Figure 1.6.9. dishold and enhold Instruction Format**
end

The *end* instruction terminates execution of the RBL script. Figure 1.6.10 shows the format of the *end* instruction.

<table>
<thead>
<tr>
<th>SYNTAX AND SAMPLE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>end</em> can be used in the following script types:</td>
</tr>
<tr>
<td>PROC   START   UNIT</td>
</tr>
<tr>
<td>LIBR   FORM</td>
</tr>
</tbody>
</table>

**Figure 1.6.10. end Instruction Format**
In a procedure script, the *finish* instruction stops the execution of the task at the finish program line. In a library script, the *finish* instruction stops the execution of all tasks used by the recipe.

Figure 1.6.11 shows the *finish* instruction format.

<table>
<thead>
<tr>
<th>SYNTAX AND SAMPLE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td><em>finish</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>finish</em> can be used in the following script types:</td>
</tr>
<tr>
<td>PROC LIBR FORM</td>
</tr>
<tr>
<td>RBLC START UNIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <em>finish</em> instruction stops the program if the allotted time for the process equals zero.</td>
</tr>
<tr>
<td>shared alias total:TOT-12; “Define alias for continuous block value”</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>shared alias total; “Declare alias”</td>
</tr>
<tr>
<td>shared constant stp:250</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>If total==stp</td>
</tr>
<tr>
<td><em>finish</em></td>
</tr>
</tbody>
</table>

Figure 1.6.11. *finish* Instruction Format
The `halt` instruction suspends execution of the script. Figure 1.6.12 shows the format of the `halt` instruction.

**SYNTAX AND SAMPLE INSTRUCTION**

The general format is:

```
halt
```

**SCRIPT TYPES**

`halt` can be used in the following script types:

<table>
<thead>
<tr>
<th>Script Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC LIBR FORM</td>
<td>PROC LIBR FORM</td>
</tr>
<tr>
<td>RBLC START UNIT</td>
<td>RBLC START UNIT</td>
</tr>
</tbody>
</table>

Figure 1.6.12. `halt` Instruction Format
halt_task

The *halt_task* instruction halts the execution of the task until either a *cont_task* instruction is executed, or the task is continued from the Batch Run screen, Batch Monitor screen, or graphic BFACE object. Figure 1.6.13 shows the format for the *halt_task* instruction.

**NOTE:** If the task is in the halt state when the Coordinator Processor reboots, dies, or switches to a redundant support card, an *on power_up* trap will not execute until you continue the task. Call up the Batch Monitor screen (BAM:) and select the “cont” field.

(For more information on batch screens, see BA: 3-1.)

**SYNTAX**
The general format is:

```
halt_task("x")
```

where `x` is the Batch Run screen tag or PeerWay address of a batch task. You can use a character string or string variable for the task name (for example, `a$`).

The PeerWay address consists of the ControlFile address and task address:

```
7702 or 77:02
```

*ControlFile address 77*  
(If omitted, *halt_task* uses current ControlFile address.)

*Task address 02*  
(Must have two digits or will cause runtime error.)

**SCRIPT TYPES**

*halt_task* can be used in the following script types:

```
PROC START UNIT
LIBR FORM
```

**SAMPLE INSTRUCTION**

Using the Batch Run screen tag:

```
halt_task("PROCESS")
```

Using the PeerWay address:

```
halt_task("7702")
```

Using a string variable:

```
a$="7702" or a$="PROCESS"
```

```
halt_task(a$)
```

*Figure 1.6.13. halt_task Instruction Format*
hold

Sets the holding flag. If the *hold* instruction is enabled, the set flag causes an *on hold* trap to execute, or a holding instruction to be true. If there is no *on hold* trap, the *hold* instruction will not affect execution of the batch script.

The *dishold* and *enhold* instructions are used to disable and enable the *hold* instruction. The default state of the *hold* instruction is enabled. If the *hold* instruction is disabled with the *dishold* instruction, the hold flag will not trip the *on hold* trap until the *enhold* instruction is executed. The *dishold* and *enhold* instructions are useful to prevent a hold interruption of a routine until it is safe to do so.

**NOTE:** The “hold” field on the Batch Monitor screen performs the same function as the *hold* instruction.

Figure 1.6.14 shows the format of the *hold* instruction.

### SYNTAX AND SAMPLE INSTRUCTION
The general format is:

```
hold
```

### SCRIPT TYPES
*hold* can be used in the following script types:

```
PROC  LIBR  FORM
RBLC  START  UNIT
```

**Figure 1.6.14. *hold* Instruction Format**
**holding**

The *holding* instruction returns a “yes” or “no” value to indicate that a holding flag for the batch task has been set. The holding flag is set by a *hold* instruction. A *resume* instruction clears the holding flag.

Figure 1.6.15 shows the format of the *holding* instruction.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
</table>
| The general format is:  
| holding |

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
</table>
| *holding* can be used in the following script types:  
| PROC   | START | UNIT |
| LIBR   | FORM  |

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
</table>
| on hold  
| while holding  
| display(x)  
| sleep(10)  
| resume; “resume clears holding flag”  
| undisplay(x)  
| .  
| .  
| hold  
| instruction bbb |

*Figure 1.6.15. holding Instruction Format*
**kill_task**

The *kill_task* instruction kills the execution of the task and puts the task in the Idle state. Figure 1.6.16 shows the format for the *kill_task* instruction.

### SYNTAX

The general format is:

```
kill_task("x")
```

where `x` is the Batch Run screen tag or PeerWay address of a batch task. You can use a character string or string variable for the task name (for example, `a$`).

The PeerWay address consists of the ControlFile address and task address:

7702 or 77:02

ControlFile address 77
(Task address 02
(If omitted, *kill_task* uses current ControlFile address.)
(Must have two digits or the instruction will fail.)

### SCRIPT TYPES

The *kill_task* instruction can be used in the following script types:

- PROC START UNIT
- LIBR FORM

### SAMPLE INSTRUCTION

Using the Batch Run screen tag:

```
kill_task("PROCESS")
```

Using the PeerWay address:

```
kill_task("7702")
```

Using a string variable:

```
a$="7702"  or a$="PROCESS"
kil_task(a$)
```

---

**Figure 1.6.16. kill_task Instruction Format**
**max and min**

The *max* and *min* instructions compare two values and assign the larger or smaller of the two values to a variable.

Figure 1.6.17 shows the format of the *max* and *min* instructions.

**SYNTAX**
The general format is:

\[ x = y \ max \ z \]

\[ x = y \ min \ z \]

where \( x \) is a variable that receives the maximum or minimum value.

**SCRIPT TYPES**
*max* and *min* can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

**SAMPLE INSTRUCTION**

```
shared local x,y,val1,val2
val1=5
val2=10
x=val1 max val2
y=val1 min val2
print(1,"Maximum value is %d",x)
print(1,"Minimum value is %d",y)
```

*Figure 1.6.17. max and min Instruction Format*
The **next** instruction is used to terminate the execution of a conditional **while**, **for**, or **until** instruction. The **next** instruction is executed by selecting the BFACE object “NEXT” command on a process graphic. When **next** is executed, the recipe continues execution at the first script line after the nested script lines in the instruction loop.

The **next** instruction is useful for terminating a conditional instruction loop from a BFACE graphic object or a Batch Monitor screen, if the termination condition in the instruction expression has not been met.

The **next** instruction is similar to a Batch Monitor screen “break” command. However, the **next** instruction allows the script configuror, and not the operator, to select break points in the script for conditional loops. The operator can only select NEXT for loops that have been designated by the **next** instruction.

Figure 1.6.18 shows the format of the **next** instruction.

---

**SYNTAX**
The general format is:

```plaintext
next
```

**SCRIPT TYPES**

**next** can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>START</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBR</td>
<td>FORM</td>
<td></td>
</tr>
</tbody>
</table>

**DEMONSTRATION OF FUNCTION**

```plaintext
while {LT101}<.9
    instruction aaa
    next
    sleep(10)
    instruction bbb; “Continue execution after executing next”
```

Figure 1.6.18. **next** Instruction Format
nvfree and vfree

The *nvfree* and *vfree* instructions return values for available nonvolatile memory and volatile memory. Memory values are assigned to a variable or array.

Figure 1.6.19 shows the format of the *nvfree* and *vfree* instructions.

**SYNTAX**
The general format is:

```
x = nvfree
x = vfree
```
where `x` is a variable that receives values for nonvolatile memory or volatile memory.

**SCRIPT TYPES**

*nvfree* and *vfree* can be used in the following script types:

```
PROC     START    UNIT
LIBR     FORM
```

**SAMPLE INSTRUCTION**

```
shared local x,y
x=nvfree
y=vfree
```

Figure 1.6.19. *nvfree* and *vfree* Instruction Format
The **opremain** instruction displays the expected time remaining in the execution of an operation. The **phremain** instruction displays the expected time remaining in the execution of a phase. These instructions are assigned an initial time value and when executed begin counting backwards from the assigned value to 0. The **opremain** and **phremain** instructions can be used with the **display** instruction to display time values on the Batch Monitor screen, the Batch Recipe Viewing screen, and the Process Graphic screen.

The **opremain** and **phremain** instructions can be used with a **while** instruction to suspend execution of the script for a specified length of time. In this capacity, the **opremain** and **phremain** instructions can function like a **sleep** instruction, but have an advantage in that they can display the amount of remaining time the script will be suspended.

Figure 1.6.20 shows the format of the **opremain** and **phremain** instruction.

### SYNTAX

The general format is:

```plaintext
opremain=x
```

where \( x \) is a time value.

The general format is:

```plaintext
phremain=x
```

where \( x \) is a time value.

### SCRIPT TYPES

**opremain** and **phremain** can be used in the following script types:

- PROC START UNIT
- LIBR FORM

### DEMONSTRATION OF FUNCTION

```plaintext
opremain=200
phremain=50
display("Operation time",opremain)
display("Phase time",phremain)
while (opremain>0); "Suspend execution of script for 200 seconds"
```

Figure 1.6.20. **opremain** and **phremain** Instruction Format
**priority**

The *priority* instruction specifies a number that determines:

- How fast the task is executed. The priority number can range from 0 to 1000.
- The execution speed of a task in relation to other tasks running simultaneously in the Coordinator Processor (CP). The CP gives the execution priority to tasks with the highest priority number.

The priority number is displayed at the bottom of the Batch Monitor and RBL Monitor screens. The priority number can be changed in increments of 25 by pressing [ENTER] on the commands “slow” or “fast” on the Batch Monitor screen, or by using the *priority* instruction in a batch script. Figure 1.6.21 shows the format of the *priority* instruction.

**SYNTAX**

The general format is:

```
priority (x)
```

where `x` is a number from 1 to 1000. The higher the number, the faster the maximum possible execution speed.

**SCRIPT TYPES**

*priority* can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

**SAMPLE INSTRUCTION**

```
priority(100)
```

**DEMONSTRATION OF FUNCTION**

This example shows the use of a *priority* instruction. The priority speed is set before and after the script instructions.

```
BEGIN::"phase introduces instructions"
priority(100)
instruction aaa
instruction bbb
priority(500)
```

Figure 1.6.21. *priority* Instruction Format
Priority in Relation to Other System Activities

The Batch Coordinator Processor (CP) performs many different process functions. These processes compete for available CP resources. The CP regulates competition by assigning a priority to each function:

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Process Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>High</td>
<td>Program Download</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Controller and CP Redundancy</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NV Memory backup and retrieval</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Alarms</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Messages</td>
</tr>
<tr>
<td>3</td>
<td>Low</td>
<td>Batch Task execution</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Background Diagnostics</td>
</tr>
</tbody>
</table>

The batches processing attempts to consume all of the available CP idle time. If higher priority processes such as alarms or controller links use more of the CP, then less memory time is available for batch.

You can prevent a particular batch task from consuming all of the remaining CP in two ways:

1. Adding a `sleep` instruction to the batch script.
2. Lowering the batch task priority.
Setting the Batch Task Priority

When setting priority, you should keep in mind three points:

1. The relationship between task priority and the maximum execution rate is nonlinear, as shown below.

<table>
<thead>
<tr>
<th>Batch Task Priority</th>
<th>Maximum Execution Rate (Lines Per Second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>As many as possible</td>
</tr>
<tr>
<td>999</td>
<td>1000</td>
</tr>
<tr>
<td>975</td>
<td>40</td>
</tr>
<tr>
<td>950</td>
<td>20</td>
</tr>
<tr>
<td>900</td>
<td>10</td>
</tr>
<tr>
<td>800</td>
<td>5</td>
</tr>
<tr>
<td>700</td>
<td>3.3</td>
</tr>
<tr>
<td>600</td>
<td>2.5</td>
</tr>
<tr>
<td>500</td>
<td>2</td>
</tr>
<tr>
<td>400</td>
<td>1.7</td>
</tr>
<tr>
<td>100</td>
<td>1.1</td>
</tr>
</tbody>
</table>

This relationship is reflected by the following formula.

\[
\text{Maximum Execution Rate} = \frac{1000}{(1000 - \text{Priority})}
\]
where 1000 is the maximum lines per second that the task is capable of executing.

2. The task priority affects only the maximum execution rate. A dozen tasks running at priority 975 cannot all execute at the maximum rate, but a single task may approach the maximum rate for short periods when it is not competing with other batch tasks.

3. Batch tasks also compete with other processes such as backup or alarms for the CP resources. Competition for memory (string variables), disk access (vdims), and controllers (aliases) will slow down a task. Furthermore, the use of aliases, virtual arrays, and chaining creates PeerWay traffic. The excess PeerWay traffic imposes a higher load on the higher priority CP processes and leaves less CP available for the batch tasks to use.
recabort, rechold, and recresume

The recabort, rechold, and recresume instructions control the execution of the recipe. These instructions enable a script running in a task to control the execution of all tasks used by the recipe. Figure 1.6.22 shows the format of the recabort, rechold, recresume instructions.

**SYNTAX**

The general format is:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recabort</td>
<td>Abort all the tasks used by the recipe. Triggers the execution of an on abort trap in all batch scripts associated with the recipe. If there is no on abort trap, the recabort instruction will not affect execution of the batch script.</td>
</tr>
<tr>
<td>rechold</td>
<td>Hold all the tasks used by the recipe. Sets the value of holding flags to true. When true, the holding flag triggers the execution of on hold traps in all batch scripts associated with the recipe. If there is no on hold trap, the rechold instruction will not affect execution of the batch tasks.</td>
</tr>
<tr>
<td>recresume</td>
<td>Continue all the tasks used by the recipe. Disables hold states for all batch scripts associated with the recipe.</td>
</tr>
</tbody>
</table>

**SCRIPT TYPES**

recabort, rechold, and recresume instructions can be used in the following script types:

- START
- LIBR

Figure 1.6.22. recabort, rechold, and recresume Instruction Format
The `resume` and `resume_task` instructions disable hold states set by the `hold` instruction and continue a script suspended by an `on hold` trap. The `resume` instruction terminates holding flags within the same script; the `resume_task` instruction terminates holding flags in a script running in another task. Figure 1.6.23 shows the format of the resume and resume task instructions.
**SYNTAX**

The general format is:

```plaintext
resume
resume_task("y")
```

where `y` is the Batch Run screen tag or PeerWay address of a batch task running the script in which you want to terminate holding flags. You can use either a character string or string variable for the task name or address. Character strings must be enclosed in quotes.

The PeerWay address consists of the ControlFile and task address:

- **ControlFile address** 77
- **Task address** 02

(If omitted, `status_task` instruction will fail.)

**DEMONSTRATION OF FUNCTION**

```plaintext
on abort
  resume
  return
on hold
  while holding
    sleep(2)
  return
PHASE_1:
  "Perform phase activities"
If tag_a == 1
  hold
```

**SCRIPT TYPES**

`resume_task` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

---

*Figure 1.6.23. resume and resume_task Instruction Formats*
The `retry` instruction directs execution from within a conditional instruction loop (such as `while`, `for`, or `until`) to a label in the script. The `retry` instruction is executed by selecting the BFACE object “RETRY” on a process graphic. The instruction must be an indented line within the loop. When `retry` is executed, the recipe terminates the loop and continues execution at the specified label.

The `retry` instruction is useful for exiting a conditional loop so that you can retry the loop again.

The `retry` instruction is similar to `goto` instruction, except that it allows the operator to choose when to direct execution from the loop to the label.

Figure 1.6.24 shows the format of the `retry` instruction.

---

**SYNTAX**

The general format is:

```
retry x
```

where `x` is a label name in the script.

**SCRIPT TYPES**

`retry` can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

**DEMONSTRATION OF FUNCTION**

```
_LABEL1:
  while {LT101}<.9
  instruction aaa
  retry _LABEL1
  sleep(10)
  instruction bbb
```

Figure 1.6.24. retry Instruction Format
The *sleep* instruction suspends script execution at the program line for a specified time interval. The time interval is the minimum sleep time; the maximum sleep time could be a few seconds longer, depending upon the response time of the Coordinator Processor.

### sleep Instruction Format

Figure 1.6.25 shows the *sleep* instruction format.

**SYNTAX**
The general format is:

```
sleep (x)
```

where x is a user-defined time interval, in seconds.

**SCRIPT TYPES**
*sleep* can be used in the following script types:

- PROC LIBR FORM
- RBLC START UNIT

**SAMPLE INSTRUCTION**

- sleep (3)
- sleep (5)
Hints for Using a sleep Instruction

- Do not rely on the sleep instruction to synchronize two tasks.
- A batch node crash will interrupt the sleep instruction. The sleep instruction time is not saved in nonvolatile memory.
- Unless slowed with a sleep instruction, while, for, and until loops can overwhelm the ControlFile and PeerWay with messages.

Changing a loop instruction:

```plaintext
while(alias)

to:
while(alias)
sleep(1)
```

Slows down batch message requests, reducing demand on the CP and controller processor.

If your batch process requires a faster reaction time, the sleep time should be no faster than 1/2 the scan time of the controller processor to prevent a batch task from accessing the controller process faster than blocks can update values.

- If a long sleep time is necessary, you should not include the entire sleep time in a sleep instruction. Instead include a sleep instruction with a shorter sleep time in a for loop. Long sleep times are more vulnerable to interruptions.

Wrong way: sleep(300)

Right way: for i=0,i<30,i=i+1
sleep(10)
### ssm and sss

The `ssm` and `sss` instructions are used to assign elapsed time, seconds since midnight (`ssm`), and seconds since Sunday (`sss`) to a declared variable. Figure 1.6.26 shows the format for the `ssm` and `sss` instructions.

#### SYNTAX
The general format is:

\[
\begin{align*}
  a &= \text{ssm} \\
  a &= \text{sss}
\end{align*}
\]

where `a` is a declared variable.

- `ssm` is the time of day as seconds since midnight, 86400 total.
- `sss` is the time of day as seconds since Sunday, 604800 total.

#### SCRIPT TYPES
`ssm` and `sss` can be used in the following script types:

- PROC LIBR FORM
- RBLC START UNIT

#### SAMPLE INSTRUCTION

```
time1=ssm
```

#### DEMONSTRATION OF FUNCTION

The example below demonstrates the `ssm` instruction. The `ssm` instruction is used to measure the time it takes to perform two task instructions. The two times are compared and, if the time required to complete `proc1` is greater than the time required to complete `proc2`, an alarm message is generated.

```
shared local time1, time2
.
.
.
time1=ssm
chain(batch,proc1); "chain to file batch, script proc1"
time1=ssm−time1; "time required to complete task1"
time2=ssm
chain(batch,proc2); "chain to file batch, script proc2"
time2=ssm−time2; "time required to complete task2"
if time1>time2
  alarm(3,2,"proc1 faster than proc2")
```

---

Figure 1.6.26. `ssm` and `sss` Instruction Format
**status_task**

The `status_task` instruction checks the status of a task from a script run in another task.

**status_task format**

Figure 1.6.27 shows the format of the `status_task` instruction.

```
SYNTAX
The general format is:
   x=status_task("y")
   where x is a variable used to check the status of a specified task:
   y is the Batch Run screen tag or PeerWay address of a batch task for which the task status is checked. You can use either a character string or string variable for the task name or address. Character strings must be enclosed in quotes.
   The Peer Way address consists of the ControlFile and task address:
   ControlFile address 77
   (If omitted, status_task uses current ControlFile address.)
   Task address 02
   (Must have two digits or the status_task instruction will fail.)

SCRIPT TYPES
status_task can be used in the following script types:
   PROC      START      UNIT
   LIBR      FORM

DEMONSTRATION OF FUNCTION
private local rs
rs=status_task("12903")
```

**Figure 1.6.27. started_by Instruction Format**
status_task Status Values

The status_task instruction returns numerical status values that can be checked to determine the status of another task. Table 1.6.1 describes status values for the status_task instruction.

NOTE:
- The terms parent and child tasks refer to tasks used to run recipes. The Parent task 1 to 32 runs the recipe and child tasks 33 to 64 run library and start scripts used by the recipe.
- The task may perform some of these activities so quickly that the status might not be recorded. Status values that might not be recorded include 32, 39, and 43.

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>The task is not configured.</td>
</tr>
<tr>
<td>-7</td>
<td>A download to the task failed.</td>
</tr>
<tr>
<td>-4</td>
<td>The task is in the Finished state.</td>
</tr>
<tr>
<td>-3</td>
<td>The task cannot execute a specific link.</td>
</tr>
<tr>
<td>-2</td>
<td>The task incurred a fatal error and cannot be restarted.</td>
</tr>
<tr>
<td>-1</td>
<td>The task incurred a soft error and can be restarted.</td>
</tr>
<tr>
<td>0</td>
<td>The task is in the Idle state.</td>
</tr>
<tr>
<td>1</td>
<td>The task is executing a unit script.</td>
</tr>
<tr>
<td>2</td>
<td>The task is executing a formula script.</td>
</tr>
<tr>
<td>3</td>
<td>The task is executing a procedure script.</td>
</tr>
<tr>
<td>6</td>
<td>The task is executing an RBL type script.</td>
</tr>
<tr>
<td>11</td>
<td>The parent task is loading a Control Recipe from disk.</td>
</tr>
<tr>
<td>12</td>
<td>The parent task is loading a Process icon.</td>
</tr>
<tr>
<td>13</td>
<td>The child task is starting a unit library script.</td>
</tr>
<tr>
<td>14</td>
<td>The parent task is waiting for a unit library script to complete.</td>
</tr>
<tr>
<td>16</td>
<td>The parent task is waiting for a unit library script to complete.</td>
</tr>
<tr>
<td>17</td>
<td>The parent task is in the Working Recipe control state.</td>
</tr>
<tr>
<td>18</td>
<td>The parent task is waiting for the Working Recipe to complete.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>The child task is running a unit start script.</td>
</tr>
<tr>
<td>21</td>
<td>The child task has completed execution of the start script.</td>
</tr>
<tr>
<td>22</td>
<td>The child task running the start script is waiting for the recipe to complete.</td>
</tr>
<tr>
<td>23</td>
<td>A unit library script for the child task is loaded and running.</td>
</tr>
<tr>
<td>24</td>
<td>The child task has completed execution of the unit library script.</td>
</tr>
<tr>
<td>26</td>
<td>The child task is loading an operation library script.</td>
</tr>
<tr>
<td>27</td>
<td>The parent task is executing an O-Icon.</td>
</tr>
<tr>
<td>28</td>
<td>The parent task is completing execution of an O-Icon.</td>
</tr>
<tr>
<td>29</td>
<td>The child task is loading a material library script.</td>
</tr>
<tr>
<td>30</td>
<td>The child task is running a material library script.</td>
</tr>
<tr>
<td>31</td>
<td>A material library script for the child task is loaded and running.</td>
</tr>
<tr>
<td>32</td>
<td>The parent task has completed execution of a Material icon.</td>
</tr>
<tr>
<td>34</td>
<td>The parent task is executing a Process icon.</td>
</tr>
<tr>
<td>35</td>
<td>The CP is running a new Process icon in a new parent task.</td>
</tr>
<tr>
<td>37</td>
<td>The parent task has completed execution of an icon and is waiting for the next icon.</td>
</tr>
<tr>
<td>38</td>
<td>The child task is waiting for a batch wipe.</td>
</tr>
<tr>
<td>39</td>
<td>The CP is performing a batch wipe of the child task.</td>
</tr>
<tr>
<td>40</td>
<td>The parent task is loading a Control Recipe from disk.</td>
</tr>
<tr>
<td>42</td>
<td>Parent task has completed execution of an icon and is waiting for the child task to release the library script.</td>
</tr>
<tr>
<td>43</td>
<td>The child script is executing a null unit start script.</td>
</tr>
</tbody>
</table>
stop

Triggers the execution of an *on stop* trap. If there is no *on stop* trap, the *stop* instruction will not affect execution of the batch script. Figure 1.6.28 shows the format of the *stop* instruction.

**SYNTAX AND SAMPLE INSTRUCTION**

The general format is:

```
stop
```

**SCRIPT TYPES**

*stop* can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>LIBR</th>
<th>FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBLC</td>
<td>START</td>
<td>UNIT</td>
</tr>
</tbody>
</table>

Figure 1.6.28. stop Instruction Format
stop_task

The stop_task instruction stops the execution of a task. The stop_task instruction is often useful to stop a task temporarily in response to a specified process condition.

Figure 1.6.29 shows the format for the stop_task instruction.

**SYNTAX**
The general format is:

```
stop_task("x")
```

where `x` is the Batch Run screen tag or PeerWay address of a batch task. You can use a character string or string variable for the task name (for example, `a$`).

The PeerWay address consists of the ControlFile address and task address:

- **ControlFile address 77**
  - (If omitted, stop_task uses current ControlFile address.)
- **Task address 02**
  - (Must have two digits or the instruction will fail.)

**SCRIPT TYPES**

`stop_task` can be used in the following script types:

- PROC  START  UNIT
- LIBR  FORM

**SAMPLE INSTRUCTION**

Using the Batch Run screen tag:

```
stop_task("PROCESS")
```

Using the PeerWay address:

```
stop_task("7702")
```

Using a string variable:

```
a$="7702"  or  a$="PROCESS"
stop_task(a$)
```

Figure 1.6.29. stop_task Instruction Format
The `sync` instruction synchronizes the execution speed of a `while` loop. The `sync` instruction specifies the time required to complete a single `while` loop. The `sync` instruction temporarily suspends execution of the loop long enough to achieve the time specified.

Figure 1.6.30 shows the format of the `stop_task` instruction.

**SYNTAX**
The general format is:

```
sync(x)
```

where `x` is the desired execution speed of the `while` loop.

**SCRIPT TYPES**
`stop_task` can be used in the following script types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Proc</th>
<th>Start</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib</td>
<td></td>
<td></td>
<td>form</td>
</tr>
</tbody>
</table>

**SAMPLE INSTRUCTION**
The `while` loop will execute for two seconds, regardless of changes in the execution rate of the task.

```
while total > 5
    instruction aaa
    sync(2)
    instruction bbb
```

*Figure 1.6.30. stop_task Instruction Format*
time

The *time* instruction assigns a time value, as known by the node where this program is running, to a declared variable. Figure 1.6.31 shows the format for the *time n* instruction. Table 1.6.2 provides a list of the available values for the *time* instruction.

**SYNTAX**

The general format is:

```
a=time(n)
```

where 

- `a` is a declared variable.
- `n` is the time value assigned to the variable.

**SCRIPT TYPES**

*time* can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

**SAMPLE INSTRUCTION**

```
until time1==30
```

**DEMONSTRATION OF FUNCTION**

```
shared local t1
.
.
.
t1=0; "set time to 0"
until t1==30; "will wait until the clock shows 30 seconds"
t1=time 1; "elapsed time"
```

*Figure 1.6.31. time Instruction Format*
### Table 1.6.2: Time Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Minutes since 1 January, 1980</td>
<td>7</td>
<td>Day of the week (Sunday=1)</td>
</tr>
<tr>
<td>1</td>
<td>Seconds</td>
<td>15</td>
<td>Number of days in this month</td>
</tr>
<tr>
<td>2</td>
<td>Minutes</td>
<td>16</td>
<td>Number of days in this year</td>
</tr>
<tr>
<td>3</td>
<td>Hours</td>
<td>21</td>
<td>Current time -- 14:21:16 = 142116</td>
</tr>
<tr>
<td>4</td>
<td>Day of the Month</td>
<td>22</td>
<td>Date (27-Jul-88 = 880727)</td>
</tr>
<tr>
<td>5</td>
<td>Month of the year (May = 5)</td>
<td>23</td>
<td>Date-European standard (27-Jul-88 = 72788)</td>
</tr>
<tr>
<td>6</td>
<td>Year (1988)</td>
<td>30</td>
<td>System time in hundredths of seconds since the CP last started.</td>
</tr>
</tbody>
</table>
Section 7:
Recipe Management Instructions

Recipe Management instructions are used to interface with or control batch recipes or data in the underlying batch configuration database. Some of these instructions can only be used in library and start scripts; others can be used in procedure scripts configured and run from a Batch Run screen.
align_on and align_wait

The **align_on** and **align_wait** instructions are used to synchronize the execution of the Main Recipe and Unit Recipe or two Unit Recipes. Because the Main Recipe and Unit Recipes execute in parallel, **align_on** instructions are used in pairs. The **align_on** instructions use character strings to coordinate execution. Neither recipe can advance beyond the **align_on** instruction until **align_on** instructions for both recipes match character strings.

The **align_on** instructions are used as expressions in **while** instructions. The while instruction executes the **align_on** instruction repeatedly until the other recipe executes an **align_on** instruction with a matching character string.

The **align_wait** instruction is used to suspend execution of the **while** loop for a specified time interval. Its function is similar to a **sleep** instruction, except that it can be interrupted when the **align_on** instructions in both subrecipes execute.

Comm-Op icon pairs are used to execute **align_on** instructions. The operation assigned to the Comm-Op icon names a script containing the **align_on** instruction.

**NOTE:**

- You can use up to five **align_on** instructions (each with a unique character string) per script. Each **align_on** instruction in the script must have a unique character string. Otherwise, the CP will ignore any subsequent use of the string and not execute that **align_on** instruction.
- You cannot nest an **align_on** while looping in another **while**, **for**, or **until** loop.
- The **align_on** instruction cannot execute more than once every 10 seconds. Therefore, 10 seconds is the minimum execution time of the while loop.
- You can use the **align_wait** instruction in the while loop to slow down and reduce unnecessary execution of the **align_on** instruction. Execution of the **align_on** instruction creates a demand on CP processor.
Figure 1.7.1 shows the *align_on* and *align_wait* format.

**SYNTAX**

The general format is:

```plaintext
while ((rs=align_on("a"))<0)
    align_wait(x)
```

where `rs` is a variable used to check the instruction status. If the status is 1, the instruction was successful. If the status is -1, the instruction was unsuccessful. If the status is -2, the instruction is attempting to execute more than once every 10 seconds. If the status is -3, the instruction does not have a unique character string.

`a` is a character string or string variable used to match *align_on* instructions in two recipes.

`x` is a user-defined time interval, in seconds.

**NOTE:** The syntax must have double parenthesis before `rs` and after "a". The syntax `while (rs=align_on("a")<0)` will not work.

**SCRIPT TYPES**

*align_on* can be used in the following script types:

- LIBR

**DEMONSTRATION OF FUNCTION**

```plaintext
private local rs
private string a$

a$="XYZ"
while((rs=align_on(a$))<0)
    align_wait(10)
```

Figure 1.7.1. *align_on* and *align_wait* Instruction Format
The `badunit` instruction returns the plant unit number for a ControlBlock if the following conditions are true:

- A Working Recipe attempts to write to a ControlBlock with a plant unit other than zero (from 1 to 255), and,
- The Batch Configuration screen has been configured to enforce plant units, and,
- The Working Recipe does not own the batch plant unit.

These conditions will also cause an `on unowned_unit` trap to execute. (For information on use of plant units and batch plant units in ABC Batch, see BA: 5-2).

Figure 1.7.2 shows the format of the `badunit` instruction.

**SYNTAX**

The general format is:

\[
x = \text{badunit}
\]

where \( x \) is a variable that receives the plant unit number from the `badunit` instruction.

**SCRIPT TYPES**

`badunit` can be used in the following script types:

- START
- LIBR

**DEMONSTRATION OF FUNCTION**

shared local \( x \)

..

on unowned_unit

\[
x = \text{badunit}
\]

until ownunit(x); “Wait here until unit is free”

sleep(2)

return

Figure 1.7.2. badunit Instruction Format
**begin_recipe**

The `begin_recipe` instruction is used to begin the execution of a Working Recipe. The `begin_recipe` instruction selects a Control Recipe, writes startup values to the Control Recipe, and begins execution of the Working Recipe.

Both the `begin_recipe` and `run_recipe` instructions let you start a Control Recipe. However, the `begin_recipe` includes several additional features that the `run_recipe` instruction does not have. The `begin_recipe` instruction lets you to write startup values for formulas, batch unit sets, and recipe locals.

Figure 1.7.3 shows the format for the `begin_recipe` instruction. Figure 1.7.4 demonstrates the `begin_recipe` instruction. Table 1.7.1 describes the status values for the `begin_recipe` instruction.

### SYNTAX

The general format for the `begin_recipe` instructions is:

```plaintext
u=begin_recipe("v","w","x",y,z_1...z_10)
```

- **u** is a variable used to check the instruction status. Use is optional though recommended. See Table 1.7.1 for status values.
- **v** is the name of the Control Recipe. The Control Recipe name can be either a character string or a string variable. Enclose character strings in quotes.
- **w** is the Batch ID for the Control Recipe. The Batch ID can be either a character string or a string variable. Enclose character strings in quotes. **NOTE:** if you use `waitbid` and `sendbid` to assign a Batch ID, enter an empty character string “”.
- **x** is the name of a formula (optional). A formula is a set of values assigned to operation parameters in a recipe. The formula can be either a character string or a string variable. Enclose character strings in quotes. If you do not want to use a formula, enter an empty character string “” (For more information on formulas, see BA: 2-6).
- **y** is the index number of a batch unit set (optional). A batch unit set is an ordered set of batch units numbered from 1 to 30. If you do not want to use a unit set, enter 0 as the unit set value. (For more information on unit sets, see BA: 2-6).
- **z** are recipe local values (optional). Recipe local values are specified on recipes and used by the `recipe_local` instruction. (For more information on recipe local values, see BA: 03.) **NOTE:** You must assign a Formulas Table to the Control Recipe before you can select a formula.

### SCRIPT TYPES

`begin_recipe` can be used in the following script types:

- PROC  START  UNIT
- LIBR  FORM

Figure 1.7.3. `begin_recipe` Instruction Format
**DEMONSTRATION OF FUNCTION**

RECS="Cooling -C"; "Recipe Name"
IDs=print$("CL%TIME"); "Create a unique Batch ID"
FRM$="Formula1"; "Formula for batch"
IDX=5; "Index for batch unit set"
R0=5; "Value for recipe_local(0)"
R1=3; "Value for recipe_local(1)"
R2=6; "Value for recipe_local(2)"

gosub BEGIN_REC

BEGIN_REC:
retrys=0; "Reset retry variable"

again:
if retrys>6
  print(1,"After 6 tries, could not start %s, Skip BEGIN_REC",RECS); X=X+1
  return
rs=begin_recipe(RECS,IDs,FRM$,IDX,R0,R1,R2)
if (rs>=0)
  retrys=0
  R$(X)=print$("^tRecipe %s started, rs=%d",b$,rs); X=X+1
  return
else
  if(rs==14)
    a$=print$("Recipe not started rs=14, Batch unit set invalid number")
  elseif(rs==13)
    a$=print$("Recipe not started rs=13, Write to recipe locals failed")
  elseif(rs==12)
    a$=print$("Recipe not started rs=12, Batch ID out of range")
  elseif(rs==11)
    a$=print$("Recipe not started rs=11, Software version incompatible")
  elseif(rs==10)
    a$=print$("Recipe not started rs=10, No NVMEM")
  elseif(rs==9)
    a$=print$("Recipe not started rs=9, Recipe not validated")
  elseif(rs==8)
    a$=print$("Recipe not started rs=8, Fatal internal error")
  elseif(rs==7)
    a$=print$("Recipe not started rs=7, Recipe missing parameter")
  elseif(rs==6)
    a$=print$("Recipe not started rs=6, Request message failed")
  elseif(rs==5)
    a$=print$("Recipe not started rs=5, Volume access failed")
  elseif(rs==4)
    a$=print$("Recipe not started rs=4, Unit access failed")
  elseif(rs==3)
    a$=print$("Recipe not started rs=3, Recipe access failed")
  elseif(rs==2)
    a$=print$("Recipe not started rs=2, Bad tag or ID being used")
  elseif(rs==1)
    a$=print$("Recipe not started rs=1, Slot being used")
  else
    a$=print$("invalid begin_recipe( ) return status code %d, rs")
error=error+1
retrys=retrys+1
  goto _again

Figure 1.7.4. begin_recipe Instruction Demonstration of Function
<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-20</td>
<td>Parameter out of range.</td>
</tr>
<tr>
<td>-17</td>
<td>Unidentified formula error.</td>
</tr>
<tr>
<td>-16</td>
<td>The formula is not in the Formulas Table.</td>
</tr>
<tr>
<td>-15</td>
<td>Cannot open the Formulas Table.</td>
</tr>
<tr>
<td>-14</td>
<td>Either an invalid number for the batch unit set or unidentified problem.</td>
</tr>
<tr>
<td>-13</td>
<td>Write to recipe locals failed.</td>
</tr>
<tr>
<td>-12</td>
<td>Batch ID is out of range. Check the syntax of the Batch ID.</td>
</tr>
<tr>
<td>-11</td>
<td>Recipe software version is incompatible with CP software version.</td>
</tr>
<tr>
<td>-10</td>
<td>There is not enough volatile memory to run the recipe.</td>
</tr>
<tr>
<td>-9</td>
<td>Control Recipe failed validation. Recipe header might not match the batch configuration. Update the Master Recipe and recreate the Control Recipe.</td>
</tr>
<tr>
<td>-8</td>
<td>Fatal or internal error encountered while executing <code>begin_recipe</code>.</td>
</tr>
<tr>
<td>-7</td>
<td>Recipe is missing a parameter.</td>
</tr>
<tr>
<td>-6</td>
<td>Request message failed due to internal software error.</td>
</tr>
<tr>
<td>-5</td>
<td>Cannot access the disk volume.</td>
</tr>
<tr>
<td>-4</td>
<td>Cannot access the batch unit.</td>
</tr>
<tr>
<td>-3</td>
<td>Task cannot access the recipe on the disk.</td>
</tr>
<tr>
<td>-2</td>
<td>Recipe name or Batch ID is bad. Another recipe might be using the Batch ID.</td>
</tr>
<tr>
<td>-1</td>
<td>Task specified in Batch Units Table is being used.</td>
</tr>
<tr>
<td>1</td>
<td>Recipe started successfully.</td>
</tr>
<tr>
<td>2</td>
<td>Recipe started successfully from a backup volume. The recipe could not start from the primary volume.</td>
</tr>
</tbody>
</table>
**display and undisplay**

The *display* instruction is used to display up to 16 variables, aliases, and dim arrays and text on the Working Recipe Info Menu, Batch Monitor screen, the Batch Recipe Viewing screen, and the Process Graphic screen. The *undisplay* instruction is used to remove the *display* instruction items to free display space for other variables.

**NOTE:** When using the *display* instruction:

- String arrays, virtual arrays, constants, and dot operators cannot be displayed with the *display* instruction. You must first assign values from these symbols to a variable to display them.

- The *display* instruction can also include the *opremain* and *phremain* functions. Values for *opremain* and *phremain* can be displayed on the Batch Monitor screen, BFACE graphic object, and Working Recipe Info Menu.

- On the Batch Monitor screen, *display* instruction variables replace other variables displayed at the bottom of the screen.

- On a BFACE graphic, an exclamation point (!) is used to mark fields for display variables. If several fields are marked for display variables, variables are displayed in the BFACE in the order that *display* instructions are executed in the script.

- On the BFACE graphic, single column fields cannot display more than 10 characters. However, multiple columns can display larger character strings. Enter the exclamation point (!) in the first column in a row and leave subsequent columns blank.
Figure 1.7.5 shows the format of the *display* and *undisplay* instructions.

**SYNTAX**
The general format is:

```
display("x",y)
```

where
- `x` is a line of text that can be displayed on the batch screen.
- `y` is a variable that can be displayed on the batch screen.

The general format is:

```
undisplay(y)
```

where `y` is the variable to be removed from the display list.

**SCRIPT TYPES**
`display` and `undisplay` can be used in the following script types:
- PROC START UNIT
- LIBR FORM

**SAMPLE INSTRUCTIONS**
```
display("Feed rate",feedrate)
```
```
undisplay(feedrate)
```
The `exit` instruction terminates the execution of a script. It can also be used to select between one of 16 possible alternative paths introduced by a Decision icon in the recipe flow chart. When the recipe executes a Decision icon, it will choose the path specified by the `exit` instruction.

Figure 1.7.6 shows the format of the `exit` instruction. Figure 1.7.7 demonstrates the use of the `exit` instruction.

**SYNTAX**

The general format is:

```plaintext
exit(x)
```

where `x` is the number of a path selected by a Decision icon. The following rules apply:

- A Decision icon can choose between a maximum of 16 alternative paths.
- Paths on the flow chart are represented by numbers 1 to 16 in ascending order from left to right, with the left-most path represented by the number 1.
- If you enter a 0 or a number higher than the actual number of paths, the decision step will select the right-most path.
- The path number can be represented by a variable or alias value.

**SCRIPT TYPES**

`exit` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**SAMPLE INSTRUCTION**

`exit(1)`

*Figure 1.7.6. exit Instruction Format*
DEMONSTRATION OF FUNCTION

The exit instruction is used to choose one of four cooling tanks introduced by a Decision icon. The Decision icon branches to the first cooling tank that is not busy.

“Operational script EXIT_OP for Decision icon”

Cooling:

if (TEMP > .5)
    exit(5); “Bypass cooling tanks”
elseif(~(Tank1_Busy))
    exit(1)
elseif(~(Tank2_Busy))
    exit(2)
elseif(~(Tank3_Busy))
    exit(3)
elseif(~(Tank4_Busy))
    exit(4)
else
    sleep(60)
    goto Cooling

Figure 1.7.7. exit Instruction Format
fetch and fetch$

The Main Recipe and Unit Recipes can use *fetch* and *fetch*$ instructions to read variables in another subrecipe.

The Comm-Op icons are used in pairs to execute the *fetch* or *fetch*$ instruction. An operation assigned to the Comm-Op icon names a script containing the *fetch* or *fetch*$ instruction. An operation assigned to a matching Comm-Op icon in another subrecipe names a script containing the variables read by the *fetch* or *fetch*$ instruction.

The *fetch* and *fetch*$ instructions specify the symbol type and name of the variable it reads, a variable that receives the value read, and a status variable. The *fetch* instruction reads decimal variables. The *fetch*$ instruction reads string variables.

**NOTE:**

- Execution time might vary for the *fetch* and *fetch*$ instructions. If you want to use *fetch* or *fetch*$ instructions to read variables at a specific point in the process, use *align_on* instructions to suspend execution of the script read.
- If there is no matching Comm-Op icon for the Comm-Op icon that executes the *fetch* or *fetch*$ instruction, *fetch* or *fetch*$ will fail.
- If you want to make sure that both the Main Recipe and Unit Recipes are running, use the *fetch* or *fetch*$ instruction after an *align_on* instruction.
- To prevent excessive PeerWay messages and excessive use of processor memory, the *fetch* and *fetch*$ instructions cannot execute more than once every 10 seconds.

Figure 1.7.8 shows the format for the *fetch* instruction. Table 1.7.2 describes the status values for the *fetch* instruction.
SYNTAX
The general format is:
\[
x = \text{fetch}("y","z",rs) \\
x$ = \text{fetch$}("y","z\$,rs)
\]
where \( rs \) is a variable used to check the instruction status. See Table 1.7.2.
\( x \) is a variable that receives the value returned from the specified variable. Use a local variable for \( \text{fetch} \). Use a string variable for \( \text{fetch$} \).
\( y \) is the symbol type of variable read, either "shared" or "private".
\( z \) is the name of the variable read. \( \text{fetch} \) reads a local variable. \( \text{fetch$} \) reads a string variable.

SCRIPT TYPES
\( \text{fetch} \) can be used in the following script types:
LIBR

DEMONSTRATION OF FUNCTION
private local rs, getvalue
private string c$
for , 
getvalue=\text{fetch}("shared","svalue",rs) 
\text{c$}=\text{fetch$}("shared","s$",rs)
if (rs>0); break
\text{sleep}(5) 

getvalue=\text{fetch}("shared","svalue",rs) 
while(rs<0)
\text{sleep}(10) 
getvalue=\text{fetch}("shared","svalue",rs) 

Figure 1.7.8. \text{fetch} Instruction Format
### Table 1.7.2. fetch status values

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>Failed: Unknown error.</td>
</tr>
<tr>
<td>-6</td>
<td>Failed: Incompatible hardware and software.</td>
</tr>
<tr>
<td>-5</td>
<td>Failed: Variable not found.</td>
</tr>
<tr>
<td>-4</td>
<td>Failed: Other task not found.</td>
</tr>
<tr>
<td>-3</td>
<td>Failed: Other unit recipe does not have the same Batch ID.</td>
</tr>
<tr>
<td>-2</td>
<td>Failed: Unit recipe not found.</td>
</tr>
<tr>
<td>-1</td>
<td>Failed: Communication channel not found.</td>
</tr>
<tr>
<td>1</td>
<td>Variable returned successfully.</td>
</tr>
</tbody>
</table>
The `filerr` instruction is used as an expression in a `while` instruction to test the execution status of an `on file_n_err` trap. The `on file_n_err` traps are typically used in library scripts to direct execution if a `print` instruction cannot write to the report file and in the start script to open that report file using an `open` instruction.

The `on file_n_err` trap in the start script activates the `filerr` instruction in the library script. As long as the library script cannot write the report file, the `filerr` instruction suspends execution of the library script. When the report file successfully opens, the `filerr` instruction clears, terminating the `while` loop.

Only the `on file_n_err` trap in the start script should attempt to open the report file if the `print` instruction fails. Library scripts should only use the `on file_n_err` trap to wait on the `filerr` instruction until the start script successfully opens the report. Using the start script to open the report prevents multiple library scripts for parallel operations from attempting to open the report file at the same time.

Figure 1.7.9 demonstrates the use of the `filerr` instruction. If a `print` instruction attempts to write to a report file that is closed, `on file_n_err` traps in both the library script and the start script will trip. In the start script, the `on file_n_err` trap includes an `open` instruction to open the report file. The `while` instruction executes the `open` instruction repeatedly until the report file is successfully opened. In the library script, another `while` instruction containing the `filerr` instruction suspends execution of the library script until the `on file_n_err` trap completes execution. Script execution then returns to the `print` instruction that previously failed.

For more information on the `filerr` instruction, see *Maintaining Report Redundancy* in BA: Appendix D.

**WARNING**

Whenever `print` instructions are included in library scripts, the library and start scripts should include `on file_1_err` traps configured as shown in Figure 1.7.9. If you do not use `on file_1_err` traps, the script will halt on the `print` instruction if the print instruction cannot write to a report file.
SYNTAX
The general format is:

```
filerr
```

SCRIPT TYPES

`filerr` can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

DEMONSTRATION OF FUNCTION

Start Script

```
“Open report file”
on file_1_err
  while (status=open(1,disk,file,0))<0
    sleep(10)
  return
```

Library Script

```
“Wait until report file opened”
on file_1_err
  while (filerr)
    return
  .
  .
  .
  print(1,”Message”); “Print to report file”
```

Figure 1.7.9. filerr Instruction Format
getmaterial and putmaterial

The getmaterial instruction reads the property values of a material on the Batch Materials Table. The putmaterial instruction writes property values to a material on the Batch Materials Table. Figure 1.7.10 demonstrates the format of the getmaterial and putmaterial instructions.

**SYNTAX**

The general format is:

```
rs = getmaterial(x$, y$, z)
rs = putmaterial(x$, y$, z)
```

where rs is a variable that is used to check the instruction status. Values include:

- 0 instruction was successful.
- -1 value not found, system or disk problems.
- -2 material name not found.
- -3 property name not found.

x$ is the name of the material listed in the Batch Materials Table. Use either a string variable or a character string enclosed in quotes.

y$ is the name of the material property. You can use either a string variable or a character string enclosed in quotes.

z is a variable containing the property sent to or received from the Batch Materials Table.

**SCRIPT TYPES**

getmaterial and putmaterial can be used in the following script types:

LIBR

**DEMONSTRATION OF FUNCTION**

```
private local status, density
private string material$, property$, msg$
property$="density"
material$=MAT$; "Obtain the name of the material name for the M-Icon that is executing"
"Obtain the material properties for the specified material"
status=getmaterial(material$, property$, density)
if status<0; "If getmaterial fails"
    msg$=print$("HELP: %s %s not available", material$, property$)
    almsg(8,1,msg$)
    print(1,msg$)
    print(1,"Material %s, Property %s is %d", material$, property$, density)
```

Figure 1.7.10. getmaterial and putmaterial Instruction Format
getmatlim

The *getmatlim* instruction reads the property value limits of a material on the Batch Materials Table. Figure 1.7.11 shows the format of the *getmatlim* instruction.

SYNTAX
The general format for the *x* instructions is:
```
rs=getmatlim(v$,w$,x,y,z)
```
where
- *rs* is a variable that is used to check the instruction status. Values include:
  - 0 instruction was successful.
  - -1 value not found, system or disk problem.
  - -2 material name not found.
  - -3 property name not found.
- *v$* is the name of the material listed in the Batch Materials Table.
- *w$* is the name of the material property.
- *x* is a variable used to get a property value.
- *y* is a variable used to get a high limit for a property value.
- *z* is a variable used to get a low limit for a property value.

SCRIPT TYPES
```
file_rename
```
can be used in the following script types:
```
LIBR
```

DEMONSTRATION OF FUNCTION
```
“Get material values from 0.0 to 1.0”
rs=getmatlim(MAT$,PROP$,mat_value,Up_Lim,Low_Lim)
if (rs==0)
    decimal_val=(mat_value-Low_Lim)/(Up_Lim-Low_Lim)
```

Figure 1.7.11. *getmatlim* Instruction Format
mapunit

The `mapunit` instruction assigns a batch unit set to the Working Recipe. A batch unit set is an ordered set of unit entries. By changing batch unit sets, you can change units assigned to Unit-Process icons.

A unit set file can have up to 30 batch unit sets. In order to use `mapunit` to assign a batch unit set to a recipe, the following conditions must be true:

- A Unit Set File is assigned to the Master Recipe or Control Recipe, \textit{and},
- The number for the batch unit set matches a configured batch unit set in the Unit Set File, \textit{and},
- The appropriate Unit-Process icons are created on the Control Recipe, \textit{and},
- Element identification numbers for units in the batch unit set are assigned to the Unit-Process icons, \textit{and},
- Units affected are defined in the Batch Units Table, \textit{and},
- In the Batch Units Table, all units in the batch unit set are configured for the same PeerWay node, \textit{and},
- The `mapunit` instruction is executed before the Unit-Process icon it affects, \textit{and},
- The Control Recipe has been validated for each batch unit set you intend to use.

Figure 1.7.12 shows the format for the `mapunit` instruction.
SYNTAX
The general format for the x instructions is:

\[ x=\text{mapunit}(y) \]

where \( x \) is a variable used to check the instruction status. Values include:
- 0 Instruction was successful.
- -1 Invalid number for the batch unit set. The number must be between 1 and 30.
- -2 Batch unit set failed recipe validation.
- -3 Internal error while executing mapunit.
- -4 Volume search failed. Could not find unit set file or batch unit set.

\( y \) is the number (from 1 to 30) of a batch unit set.

SCRIPT TYPES
mapunit can be used in the following script types:

<table>
<thead>
<tr>
<th>Script Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC</td>
<td>START UNIT</td>
</tr>
<tr>
<td>LIBR</td>
<td>FORM</td>
</tr>
</tbody>
</table>

DEMONSTRATION OF FUNCTION

```plaintext
private local status
private constant AVAILABLE:1
private alias Leach1:=1A-30/a; “CB signal that Leacher1 is available”
private alias Leach2:=1A-30/b; “CB signal that Leacher2 is available”

AGAIN
while 1
if Leach1==AVAILABLE
    status=mapunit(1); “Use batch unit set 1 for recipe”
    if status<0
        almsg(8,1,”FATAL ERROR”)
        while 1
            retry AGAIN
        break
    break
if Leach2==AVAILABLE
    status=mapunit(2); “Use batch unit set 2 for recipe”
    if status<0
        almsg(8,1,”FATAL ERROR”)
        while 1
            retry AGAIN
        break
    break
sleep(3)
```

Figure 1.7.12. mapunit Instruction Format
The `materialdesc` instruction reads the material description of a specified material in the Batch Materials Table and writes the description to a character string variable. Material descriptions are defined for materials on the Material Properties screen.

Figure 1.7.13 demonstrates the format of the `materialdesc` instruction.

### SYNTAX

The general format is:

```
rs=materialdesc(x$,y$)
```

where `rs` is a variable that is used to check the instruction status. Values include:

- 0 instruction was successful.
- -1 value not found, system or disk problems.
- -2 material name not found.
- -3 property name not found.

`x$` is a character string variable that specifies the name of a material listed in the Batch Materials Table.

`y$` is a character string variable that receives the text description of the material (that is specified by `x$`).

### SCRIPT TYPES

`materialdesc` can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

### DEMONSTRATION OF FUNCTION

```plaintext
private local status
private string material$, desc$, msg$
material$=MAT$; "Obtain the name of the material name for the M-Icon that is executing"
"Obtain the description of the specified material"
status=materialdesc(material$,desc$)
if status<0; "If materialdesc fails"
   msg$=print$("HELP: %s not available",material$)
almsg(8,1,msg$)
   print(1,msg$)
   print(1,"Material %s , Description %s",material$,desc$)
```

Figure 1.7.13. `materialdesc` Instruction Format
The `materialunits` instruction reads the engineering units of measure of a specified material property in the Batch Materials Property Table and writes the units of measure to a character string variable. Engineering units of measure are defined for material properties on the Material Properties screen. Figure 1.7.14 shows the format of the `materialunits` instruction.
SYNTAX
The general format is:

\[
rs = \text{materialunits}(x$,y$,z$)
\]

where \( rs \) is a variable that is used to check the instruction status. Values include:

- 0 instruction was successful.
- -1 value not found, system or disk problems.
- -2 material name not found.
- -3 property name not found.

\( x$ \) is a character string variable that specifies the name of a material listed in the Batch Materials Table.

\( y$ \) is a character string variable that specifies the name of a material property listed in the Batch Material Properties Table.

\( z$ \) is a character string variable that receives the engineering units of measure of the material property (that is specified by \( x$ \) and \( y$ \)).

SCRIPT TYPES
\textit{materialunits} can be used in the following script types:

- PROC \quad START \quad UNIT
- LIBR \quad FORM

DEMONSTRATION OF FUNCTION

private local status
private string property$, material$, units$, msg$
property$="Density"
material$=\text{MAT}$; "Obtain the name of the material name for the M-Icon that is executing"
"Obtain the Engineering units of measure for the material property"
status=\text{materialunits}(\text{material$\prime$,property$\prime$,units$\prime$})
if status$\leq$0; "If materialunits fails"
\hspace{1em}msg$=\text{print}(\text{"HELP:%s %s not available"},\text{material$\prime$,property$\prime$})$
\hspace{1em}almsg(8,1,msg$
\hspace{1em}print(1,msg$
\hspace{1em}print(1,\text{"Material %s, Property %s, Units %s"},\text{material$\prime$, property$\prime$, units$\prime$})

\textbf{Figure 1.7.14. materialunits Instruction Format}
on unowned_unit

The on unowned_unit instruction executes only if all of the following conditions are true:

- A Working Recipe attempts to write to a ControlBlock with a plant unit other than zero (from 1 to 255), and,
- The recipe has been configured to enforce plant units (on the Batch Configuration screen), and,
- The Working Recipe does not own the batch plant unit, and,
- The plant unit is not specified for the batch unit on the Batch Unit Table.

(For information on use of plant units and batch plant units in ABC Batch, see BA: Appendix C.)

Figure 1.7.15 shows the format of the on unowned_unit instruction.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td>on unowned_unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>on unowned_unit can be used in the following script types:</td>
</tr>
<tr>
<td>START</td>
</tr>
<tr>
<td>LIBR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared local x</td>
</tr>
<tr>
<td>on unowned_unit</td>
</tr>
<tr>
<td>x=badunit</td>
</tr>
<tr>
<td>until ownunit(x); “Wait here until unit is free”</td>
</tr>
<tr>
<td>sleep(5)</td>
</tr>
<tr>
<td>return</td>
</tr>
</tbody>
</table>

Figure 1.7.15. on unowned_unit Instruction Format
ownunit and disownunit

The ownunit instruction assigns ownership of a specified plant unit to a Working Recipe. Only one recipe at a time can own a plant unit. The ownunit instruction can return a status value that can be checked in the script to see if the instruction was successful.

The disownunit instruction terminates the Working Recipe ownership of a plant unit. You should disown a plant unit after the recipe has finished using it so another recipe can use it. If the recipe does not own the plant unit, the disownunit instruction will not affect recipe execution.

You can check the ownership status of plant units on the the Batch Plant Unit Status screen. The tag of the Working Recipe that owns the plant unit is displayed in the “Ownership” field next to the batch plant unit.

☐ To call up the Batch Plant Unit Status screen, type:

BAP  [RETURN]

NOTE:

- If a plant unit has been assigned (on the Batch Unit Table) to the recipe batch unit, the Working Recipe does not need to execute ownunit to write to a block with the same plant unit.

- Any exception logic that terminates the recipe must also terminate recipe ownership of batch plant units.

- If ownership of a plant unit on the Batch Plant Unit Status screen is incorrect, you can delete the incorrect owner. An incorrect owner might be displayed if the recipe terminates execution without terminating ownership of the plant unit. An incorrect owner might appear as N002S36.

For more information on batch plant units, see BA: Appendix C.

Figure 1.7.16 shows the format of the ownunit and disownunit instructions.
SYNTAX
The general format is:

\[ x = \text{ownunit}(y) \]
\[ \text{disownunit}(y) \]

where \( x \) is a variable used to check instruction status. Use is optional though recommended. Values include:

- 1 Instruction was successful.
- 0 Instruction was unsuccessful.

\( y \) is the number of a plant unit (between 1 to 255) or a variable that represents the plant unit number.

SCRIPT TYPES
\textit{ownunit} and \textit{disownunit} can be used in the following script types:

\begin{itemize}
  \item START
  \item LIBR
\end{itemize}

DEMONSTRATION OF FUNCTION
on unowned_unit

\begin{verbatim}
status=0
while (status<1)
  status=ownunit(badunit)
sleep(10)
return
\end{verbatim}

Figure 1.7.16. ownunit and disownunit Instruction Format
The recipe_local instruction is used to read or write to recipe local values in the Working Recipe. On the recipe, recipe local values are defined on the “Local” menu before starting or during execution of the recipe. You can also use a begin_recipe instruction to assign recipe local values to a recipe at start time.

NOTE:

- Recipe locals are negatively offset by one integer in the script. For example, the index number 1 for a recipe local in the recipe is represented by 0 in the script, the index number 2 by 1, and so on.
- Recipe locals are intended to be used in scripts that execute a begin_recipe instruction. The script that executes begin_recipe should only read recipe locals, not write to them. The recipe locals are saved on disk as part of the Working Recipe.

Figure 1.7.17 shows the format of the recipe_local instruction.

**SYNTAX**

The general format for the recipe_local instructions is:

```plaintext
recipe_local(x)
```

where x is an index number from 0 to 9 for a recipe local value.

**SCRIPT TYPES**

recipe_local can be used in the following script types:

LIBR START

**DEMONSTRATION OF FUNCTION**

shared local Var

R0=3, R1=15, R2=323.5; “assign values for recipe locals to variables”

“Begin recipe and write values to recipe locals”

rs=begin_recipe(RECS,IDS,FORMS,INDEX,R0,R1,R2)

“Assign recipe local to a variable”

Var=recipe_local(1); “Var equals 15”

“Change recipe local”

recipe_local(2)=450; “recipe local(2) changed from 323.5 to 450.”

Figure 1.7.17. recipe_local Instruction Format
run_recipe

The `run_recipe` instruction starts the execution of a configured Control Recipe. The `run_recipe` instruction is useful for starting one recipe from another recipe that is already running or from a standard batch procedure script. Figure 1.7.18 shows the format of the `run_recipe` instruction.

**NOTE:** if the recipe uses batch unit sets or formulas, use the `begin_recipe` instruction instead of the `run_recipe` instruction.

Figure 1.7.18 shows the format of the `run_recipe` instruction. Table 1.7.3 describes the status values for the `run_recipe` instruction.

### SYNTAX

The general format is:

```
w=run_recipe("x","y", z)
```

where `w` is a variable used to check the instruction status. Values include:

- `x` is the name of the Control Recipe. You can enter the Control Recipe name in either a character string or a string variable. Character strings must be enclosed in quotes.
- `y` is an field for an alphanumeric Batch ID for the Control Recipe. A Batch ID can include up to 32 characters. It must begin with an alphabetical letter. You can enter the Control Recipe name in either a character string or a string variable.

**NOTE:** If a Batch ID is automatically generated with a `wait_bid` and `send_bid` instruction, this field should not have a Batch ID. However, the field must have an entry. Enter a blank character string (for example, " ").

- `z` is a field reserved for future use. Enter 0 in this field.

### SCRIPT TYPES

`run_recipe` can be used in the following script types:

```
PROC START UNIT
LIBR FORM
```
DEMONSTRATION OF FUNCTION

REC$="Cooling - C"; "Recipe Name"
ID$=print$("CL%TIME"); "Create a unique Batch ID"

gosub RUN_REC

RUN_REC:
retrys=0; "Reset retry variable"
    again:
        if retrys>6
            print(1,"After 6 tries, could not start %s, Skip RUN_REC",REC$); X=X+1
            return
        rs=run_recipe(REC$,ID$,0)
        if (rs>=0)
            retrys=0
            R$(X)=print$("^tRecipe %s started, rs=%d",b$,rs); X=X+1
            return
        else
            if(rs==-12)
                a$=print$("Recipe not started rs=-12, Batch ID out of range")
            elseif(rs==-11)
                a$=print$("Recipe not started rs=-11, Software version incompatible")
            elseif(rs==-10)
                a$=print$("Recipe not started rs=-10, No NVMEM")
            elseif(rs==-9)
                a$=print$("Recipe not started rs=-9, Recipe not validated")
            elseif(rs==-8)
                a$=print$("Recipe not started rs=-8, Fatal internal error")
            elseif(rs==-7)
                a$=print$("Recipe not started rs=-7, Recipe missing parameter")
            elseif(rs==-6)
                a$=print$("Recipe not started rs=-6, Request message failed")
            elseif(rs==-5)
                a$=print$("Recipe not started rs=-5, Volume access failed")
            elseif(rs==-4)
                a$=print$("Recipe not started rs=-4, Unit access failed")
            elseif(rs==-3)
                a$=print$("Recipe not started rs=-3, Recipe access failed")
            elseif(rs==-2)
                a$=print$("Recipe not started rs=-2, Bad tag or ID being used")
            elseif(rs==-1)
                a$=print$("Recipe not started rs=-1, Slot being used ")
            else
                a$=print$("invalid run_recipe ( ) return status code %d, rs")
            error=error+1
            retrys=retrys+1
            print(1,"Retrying %s , rs=%d",b$,rs)
            goto _again

Figure 1.7.19. begin_recipe Instruction Demonstration of Function
### Table 1.7.3. Status Values for run_recipe Instruction

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-19</td>
<td>run_recipe cannot use batch unit sets.</td>
</tr>
<tr>
<td>-18</td>
<td>run_recipe cannot use formulas.</td>
</tr>
<tr>
<td>-12</td>
<td>Batch ID is out of range. Check the syntax of the Batch ID.</td>
</tr>
<tr>
<td>-11</td>
<td>Recipe software version is incompatible with CP software version.</td>
</tr>
<tr>
<td>-10</td>
<td>There is not enough volatile memory to run the recipe.</td>
</tr>
<tr>
<td>-9</td>
<td>Control Recipe failed validation. Recipe header might not match the batch configuration. Update the Master Recipe and recreate the Control Recipe.</td>
</tr>
<tr>
<td>-8</td>
<td>Fatal or internal error encountered while executing <code>run_recipe</code>.</td>
</tr>
<tr>
<td>-7</td>
<td>Recipe is missing a parameter.</td>
</tr>
<tr>
<td>-6</td>
<td>Request message failed due to internal software error.</td>
</tr>
<tr>
<td>-5</td>
<td>Cannot access the disk volume.</td>
</tr>
<tr>
<td>-4</td>
<td>Cannot access the batch unit.</td>
</tr>
<tr>
<td>-3</td>
<td>Task cannot access the recipe on the disk.</td>
</tr>
<tr>
<td>-2</td>
<td>Recipe file name or Batch ID is bad. Another recipe might the same file name or Batch ID.</td>
</tr>
<tr>
<td>-1</td>
<td>Task specified in Batch Units Table is being used.</td>
</tr>
<tr>
<td>1</td>
<td>Recipe started successfully.</td>
</tr>
<tr>
<td>2</td>
<td>Recipe started successfully from a backup volume. The recipe could not start from the primary volume.</td>
</tr>
</tbody>
</table>
The \textit{started\_by} instruction returns information on how the recipe was started. A numeric value indicates whether the recipe was started by an operator or another recipe; a text value identifies the key ID of the user or the tag name of the other recipe that started the recipe. This information is useful because parameter assignments might have to be performed differently, depending on whether the recipe is started automatically from another recipe or manually by an operator.

Figure 1.7.20 shows the format of the \textit{started\_by} instruction.
SYNTAX
The general format is:
\[ x = \text{started\_by}(y$) \]
where \( x \) is a variable that can be checked in the script to determine how the recipe was started. Values include:
- 1 Recipe started by operator
- 2 Recipe started by another recipe
\( y$ \) is a string variable that receives the identification of the user or recipe that starts the recipe. The strings can return the following values:

<table>
<thead>
<tr>
<th>String Content</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Key ID</td>
<td>CONFIG 2</td>
</tr>
<tr>
<td>Recipe Tag</td>
<td>BA092493</td>
</tr>
</tbody>
</table>

SCRIPT TYPES

\textit{started\_by} can be used in the following script types:
- START
- LIBR

DEMONSTRATION OF FUNCTION

"Check how Control Recipe BAT\_A was started"
shared local start
private string rec$, evt$
start=started\_by(rec$)
if start==2
    evt$=print$("Started by Recipe %s",rec$)
event(10,0,evt$)

Figure 1.7.20. \textit{started\_by} Instruction Format
usegraphic

The *usegraphic* instruction assigns a process graphic file to the recipe. After the *usegraphic* instruction has executed, you can call up the process graphic as follows:

- Press [SELECT] on the “Graphic” field on the Batch Working Recipe screen.
- Press [SELECT] on a recipe task on the Batch Overview screen after the *usegraphic* instruction has been executed.
- If a batch task generates an active alarm, press the [ACTIVE ALARM] key to call up the process graphic designated by the *usegraphic* instruction.

**NOTE:**

- The *usegraphic* instruction is typically included in the library script for the unit on the Batch Units Table. Once executed, the graphic named by the *usegraphic* instruction is available to all tasks with the same Batch ID.
- If the graphic is not available, an error message in a yellow banner will appear, and the screen will call up the Working Recipe.
- You cannot use the *usegraphic* instruction in a procedure script (type “PROC”). In order to use graphics with batch processes run from a procedure script, enter the name of the graphic in the “Graphic” field on the Batch Run Screen.

You can specify the number of an optional anchor object. Anchors are used to center a location on the process graphic on the screen and are useful for viewing a graphic when a graphic is larger than one screen. When the process graphic is called up, the screen scrolls to the location identified by the anchor object with the anchor object at its center. (For more information on creating an anchor object, see CC: 2-2.)

A second *usegraphic* instruction will assign a new graphic to the recipe. Figure 1.7.21 shows the format of the *usegraphic* instruction.
SYNTAX
The general format is:

\texttt{usegraphic("x","y",z)}

where \( x \) is the disk volume on which the associated graphic resides. The default volume is the volume on which the Batch Configuration Database resides.

\( y \) is the name of the graphic file.

\( z \) is the number of an anchor object on the process graphic. This field is only functional if an anchor object has been configured on the process graphic. If there is no anchor on the process graphic, enter the number 0 in this field.

SAMPLE INSTRUCTION

\texttt{usegraphic("DISK1","GR1",3)}

\textbf{Figure 1.7.21. usegraphic Instruction Format}
wait_bid and send_bid

The `wait_bid` and `send_bid` instructions are used to generate Batch IDs for Control Recipes. The `wait_bid` instruction waits for a request from a Control Recipe for a Batch ID. The `send_bid` instruction sends an alphanumeric batch identification code to the Specific Recipe.

A Batch ID is an alphanumeric identification code that identifies a recipe. The `send_bid` instruction can send Batch IDs of up to 32 characters. The recipe uses a tag mask configured on the Batch Configuration screen to select up to eight characters from the Batch ID for the task tag. For information on generating batch task tags, see BA: 1-3. For more information on masking a Batch ID, see BA: 2-1.

A procedure script is used to execute `wait_bid` and `send_bid`. If so, on the Batch Unit Table, enter the tag name of the Batch Run screen in the “Task ID” field for the batch unit.

Figure 1.7.22 shows the format of the `wait_bid` and `send_bid` instructions.

**NOTE:** When the ID task is running in a node that is also executing a begin or run recipe, the execution of the ID task may be blocked. By doing a backup instruction immediately before the begin/run recipe, the chances of the problem occurring are greatly reduced.
SYNTAX
The general format is:

wait_bid(x,y,z)

where 
- x returns the number of the CPIV node in which the recipe requesting the Batch ID is run. The node number is assigned to the recipe in the Batch Unit Table.
- y returns the number of the batch task in which the recipe requesting the Batch ID is run. The task number is assigned to the recipe in the Batch Unit Table.
- z is the length of the Batch ID. The length is determined by the “Max Batch ID Length” field on the Batch Configuration Database.

The general format is:

send_bid(a$)

where 
a$ is a character string used by the Control Recipe as a Batch ID. The Batch ID can be a maximum of 32 characters. A masking feature defined on the Batch Configuration Database constructs a Batch tag by selecting characters (maximum of 8) from the Batch ID.

SCRIPT TYPES
wait_bid can be used in the following script types:
- PROC START UNIT
- LIBR FORM

DEMONSTRATION OF FUNCTION
shared local node, slot, bid_max
shared local rs
private string id$, grade$
private alias GRADE:=3E-34/A; “Contains the grade of product being made”
private alias LOT:=3E-34/B; “Contains the grade of product being made.”
private alias BATCH:=3E-34/C; “Contains the grade of product being made.

“Generate ID in the form of:”
“ NOTE: Tag must start with an alphanumeric character”
“ Example ID: ARLOT012BATCH394_D022394_T130134”
“ Tag MASK: XX____XXX_____XXX_____________”
“Generate tag of AR012394”

while 1
priority(100); “Lower the priority to reduce load”
rs=wait_bid(node,slot,bid_max); “Wait for the next ID request”
priority(1000); “Increase the priority to speed response”
if GRADE==1; grade$="A"; “Generate the correct grade character.”
elseif GRADE==2; grade$="B"
else; grade$="C"
id$=print$(“%sRL%3dBATCH%3d_D%DATE_T%TIME”,grade$,LOT,BATCH)
send_bid(id$); “Send the ID generated to the correct task”
flog("IDtask","ID %s from Nd %d;%d",node,slot); “Log the message”
BATCH=BATCH+1; “Increment the BATCH storage block”

if BATCH>100
BATCH=1
LOT=LOT+1

Figure 1.7.22. wait_bid and send_bid Instruction Format
Section 8:  
On Trap Instructions

This section describes on instructions. The on instruction sets a trap to direct execution of the script when a specified event occurs. If the event occurs, the on instruction is executed along with any indented lines under the on instruction. If a return instruction is indented under the on instruction, it returns the script to the program line that was executing when the on instruction executed.
On Trap Overview

The following characteristics apply to on instructions:

- Instructions configured before the first phase name of a script are always active during execution of the script. These on instructions are often called global on instructions.

- An on instruction configured within a phase is only active during execution of that phase.

- If a gosub instruction is used to direct the script to another phase, on instructions in the original phase, the new phase, and the global environment are all active.

- An on instruction that specifies a ControlBlock alarm or event only executes when the alarm or event occurs; it does not execute simply because the condition exists. For example, the instruction on =2A-2/a checks for logic step a of ControlBlock =2A-2 to go into alarm. If the logic step is already in alarm when the on instruction is activated, then the on instruction does not see the alarm occur and does not execute.

- The batch script environment determines which on traps are active. For example, on traps in the phase local environment and in the global environment of the script are always active. The on traps may be active in multiple phases that are connected by gosub instructions.

- The on trap order of execution is determined either by the script environment or by a priority designation (alarm traps only).

- While scripts are linked by chain instructions, only the on traps in the child script are active (see Section 5.)

- The alarm on trap will not execute unless the “Report” and “When” fields on the ControlBlock discrete diagram display are configured. These fields must be configured in order for the ControlBlock to generate an alarm. (For rules on configuring the “when” field, see CB: 6.)

- The on traps should include a return instruction.
On Trap Execution Priority

Alarm *on* traps are executed in response to ControlBlock alarms or events. System *on* traps are executed in response to system errors in the batch or RBLC hardware and software.

In typical operations, the Coordinator Processor or System Resource Unit monitors all alarms and system activity on the PeerWay. If an active *on* trap matches an alarm or system error, that trap is executed. However, two conditions can occur that can affect the response of *on* traps in a script:

- An alarm or system error occurs when there is more than one *on* trap for the same ControlBlock address or system error condition.
- Alarms or system errors occur for one or more active *on* traps while an *on* trap is executing.

More than One On Trap per ControlBlock Address

If the script contains more than one *on* trap for the same ControlBlock address or system condition, only one *on* trap can be active in memory at a time. The following conditions determine active priority:

- *on* traps in the lowest nesting level will be active. (This is only true for alarm traps that do not have priority designations.)
- The *on* traps in the local environment of a script have execution priority over *on* traps in the global environment (unless the global alarm trap has a higher priority designation).
- The *on* traps in a subphase accessed by a *gosub* instruction have execution priority over higher level on traps in the local environment. For example, the on trap “on =1A-1/a” is active in “SUBPHASE_1” but not in “PHASE_1” or the global environment.
- If two on traps in the same nesting level have the same type or address, the first on trap will not execute.

**NOTE:** On alarm traps that have priority designations are an exception to these rules. The alarm trap with highest priority designation will be active, regardless of nesting level.

Figure 1.8.1 shows how nesting levels can establish execution priority for *on* traps that do not have a priority designation.
The indicated *on* traps are active in memory. When there are more than one *on* trap for the same condition, only the *on* trap in the lowest nesting level is active.

Figure 1.8.1. More than One on Trap per Control Block Address
Execution Order in Response to Multiple Alarms or System Conditions

If an on trap is executing when an alarm or system condition occurs for another active on trap, the executing on trap is interrupted and the next on trap is executed. When the next on trap finishes executing, the first on trap resumes execution. The script can jump from one on trap to another for a maximum of 40 nesting levels. Figure 1.8.2 shows how nesting levels determine the order of on trap execution when more than one alarm is generated for active on traps.

Figure 1.8.2. Execution Order for on Traps in Response to Multiple Alarms
NOTE:

- A *return* instruction returns the script to the line that was executing when the *on* trap executed.

- A *start* instruction directs the script to a phase name. But it also clears the *on* traps in all nesting levels. If an *on* trap executes a *start* instruction, all other traps that have been interrupted will not finish executing.

- You can assign a priority designation to an *on* alarm trap that will prevent *on* alarm traps with lower priorities from interrupting it.

- Exiting an *on* trap with a *gosub* to a phase instead of a *return* instruction clears nesting levels for all *on* traps.
Alarm Traps

An alarm on trap directs the script to on x instructions in response to an alarm condition or an event specified by a ControlBlock logic step. Table 1.8.1 describes the alarm on trap format.

**NOTE:** Alarm on traps can only be used in batch tasks and not in RBLC applications.

<table>
<thead>
<tr>
<th>on Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on x</td>
<td>Executes if a ControlBlock logic step, alias, or continuous output (x) goes into alarm or causes an event alarm.</td>
</tr>
<tr>
<td>where x is a Control Block variable address or an alias.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** Aliases can use dot operators to indicate registers.

**Alarm Traps with Priority Designation**

You can assign an on alarm trap one of three numerical priority designations to define its execution priority:

- on1 tag or {address}  highest priority
- on2 tag or {address}  midlevel priority
- on3 tag or {address}  lowest priority

If an on alarm trap is executing when an alarm for another active on alarm trap occurs, the priority designation can prevent the on trap for that alarm from interrupting the currently executing on alarm trap.

An on alarm trap with a low priority cannot interrupt one with a higher priority. This priority applies only to on alarm traps. System on traps can still interrupt on alarm traps, regardless of the on alarm trap priority.

**NOTE:** on alarm traps must have a return instruction to return the script to the interrupted on alarm trap.
on Alarm Traps without Priority Designation

Alarm traps without a priority designation can interrupt each other in response to alarm conditions. If *on* alarm traps with and without priority designations are both used in the same script, the *on* alarm traps with priority designations can interrupt *on* traps without priority designations.
System Traps

System traps direct execution of the script in response to error conditions that result from malfunctions in the Coordinator Processor, System Resource Unit, nonvolatile memory, or run time errors (run, hold, continue, abort, stop, and kill). Table 1.8.2 describes system on traps that can be used in batch and RBLC scripts. Table 1.8.3 describes system on traps that can be used only in batch scripts.

Table 1.8.2. on Instruction Format for Batch and RBLC System Traps

<table>
<thead>
<tr>
<th>on Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| on abort              | Executes in response to an “ABORT” command (on a Batch Monitor screen or BFACE object), an abort instruction, or a recabort instruction.  
**NOTE:** If the task is in the abort state when the Coordinator Processor reboots, dies, or switches to a redundant support card, the on power_up trap will not execute until you continue the task. Call up the Batch Run screen BAR and select the “cont” field. |
| on hold               | The on hold instruction executes in response to a “HOLD” command on a BFACE graphic object, the Batch Monitor screen, or from a hold instruction in a batch script. If the hold command is enabled, it sets an internal holding flag for the task that triggers the on hold instruction. The on hold trap (which includes the on hold instruction and indented lines) will then direct execution of the script.  
**NOTE:** If a “hold” command is disabled by the dishold instruction, the on hold instruction will not execute. However, if the script later executes an enhold instruction, the on hold instruction will execute at that time. |
| on unowned_unit       | Executes if a Working Recipe attempts to write to a ControlBlock with a plant unit other than zero (from 1 to 255). For more information on this instruction, see Section 7.  
**NOTE:** Include the on unowned_unit trap in both the library and start scripts. |
| on stop               | Executes in response to a “STOP” command (on a Batch Monitor screen or BFACE object), a stop instruction, or a recstop instruction. |
| on file_n_err         | Executes if an attempt to print report number n fails, where n ranges from 1 to 10. If the report has been successfully opened but a print instruction fails, this on instruction is executed.  
**NOTE:** Include the on file_n_err trap in both the library and start scripts. |
<p>| on link_err           | Executes if an error occurs when the script is accessing a ControlBlock. When this on instruction is executed, the automatic once-per-second retry of communication with the ControlBlock is suspended. |
| on power_up           | Executes in response to a restart of the Coordinator Processor, RBL Controller, or a load switch to a redundant Coordinator Processor card. |
| on nest_err           | Executes if outstanding nesting instructions exceed nesting limits of more than 40 active nesting instructions or if a return is executed when no gosub has been declared. To clear the nested stack, execute a start instruction. |</p>
<table>
<thead>
<tr>
<th>Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on acquire_err</td>
<td>Executes in response to a faulty acquire condition. An acquire fault can occur if the master task cannot log an acquire request to the acquire queue of the slave task. This should not occur under normal operating conditions. (For more information on on acquire_err, see Section 10.)</td>
</tr>
<tr>
<td>on chain_err</td>
<td>Executes if the script named by a chain instruction does not exist on either the primary volume or backup volume when a chain command is issued. If either of these checks fail, chaining results in a “Chain Error” condition and processing is directed to the trap. <strong>NOTE:</strong> on chain_err only parses in script types UNIT, FORM, and PROC.</td>
</tr>
<tr>
<td>on input_err</td>
<td>Executes in response to an incomplete execution of an operator input instruction. For example, the trap executes if no console is available to an input or print instruction.</td>
</tr>
</tbody>
</table>
| on lost_master     | Executes if the slave task loses its master task. The slave task can use on lost_master to release itself from a lost master task. (For details on terminating a master and slave relationship, see Section 10.) A master task is considered to be lost prematurely if it is terminated before it can execute an unacquire instruction or the slave task can execute a release instruction. A master task may be lost because:  
  • Master/Slave task relationship is terminated at the Batch Acquire Queues Screen.  
  • Master task is killed, finishes, or is put in “Idle” state.  
  • Node on which master task resides is wiped manually or wiped by a bad nonvolatile memory board during a reboot. **NOTE:** You cannot use on lost_master to remove an entry for a pending master task from the slave task acquire queue. |
| on lost_slave      | Executes if the master task loses its slave task. The master task can use on lost_slave to unacquire a lost slave task or remove an entry for a lost slave task from its acquire queue. (For details on terminating a master and slave relationship, see Section 10.) A slave task is considered to be lost prematurely if it is terminated before it can execute a release instruction or if the master task can execute an unacquire instruction. A slave task may be lost because:  
  • Master/Slave task relationship is terminated at the Batch Acquire Queues Screen.  
  • Slave task is killed, finishes, or is put in “Idle” state.  
  • Node on which slave task resides is wiped manually or wiped by a bad nonvolatile memory board during a reboot. |
| on no_backup       | Executes if the batch task is not being backed up to the nonvolatile memory. This emergency results from a low power supply battery for the nonvolatile memory. The system will suspend backup if AC power to the nonvolatile memory falls below 20 volts. Batch tasks in the Coordinator Processor stop until the nonvolatile memory returns. You can use a backup instruction in the on no_backup trap to suspend the task until a backup can be performed. |

(continued on next page)
Table 1.8.3. on Instruction Format for Batch Only System Traps (continued)

<table>
<thead>
<tr>
<th>Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on task_retry</td>
<td>Executes if a batch task cannot be accessed on the PeerWay.</td>
</tr>
<tr>
<td>on unacquire</td>
<td>Executes if the master task terminates a master/slave relationship. Use this trap only in the slave task. If there is no on unacquire trap in the slave task, a fatal error will result if the master task fails. (For more information on on unacquire, see Section 10.)</td>
</tr>
</tbody>
</table>
| on varray_err      | The on varray_err trap executes if a file error prevents the script from either reading or writing to a virtual array.  
**NOTE:** The on varray_err trap will not execute in response to errors that occur during a read or write to a virtual string array ($vstringdim$), or redundant virtual string arrays ($rvdim$ or $rvstringdim$). |
clear_alarm_queue

The `clear_alarm_queue` instruction clears the batch alarm queue of alarms for `on` alarm traps. These alarms are stored in the queue until the appropriate `on` alarm trap executes. The order in which `on` alarm traps execute in response to the alarms depends on the time the alarm was generated and the priority of the `on` alarm trap. The oldest alarms or alarms for the highest priority `on` alarm traps are executed first. The alarms are normally cleared from the queue when the `on` alarm trap executes.

**NOTE:** The queue filters alarms to prevent queuing of an alarm if an identical alarm is already in the queue. The filter prevents `on` alarm traps from repeatedly executing in response to the same alarm. The filter only works for `on` alarm traps with priority designations 1, 2, or 3.

The size of the alarm trap filter is limited by the available volatile memory space for the batch CP. Each queued alarm uses approximately 10 bytes.

Figure 1.8.3 shows the `clear_alarm_queue` instruction format.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td><code>clear_alarm_queue</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clear_alarm_queue</code> can be used in the following script types:</td>
</tr>
<tr>
<td>PROC</td>
</tr>
<tr>
<td>LIBR</td>
</tr>
</tbody>
</table>

**Figure 1.8.3. clear_alarm_queue Instruction Format**
Alarm, log, and report instructions are used to generate alarm messages, log messages, and reports. They are also used to communicate with terminals, printers, and other devices from a script.
alarm, almsg, and clear

The *alarm* instruction generates alarms. The *almsg* instruction generates a cleared alarm. The *clear* instruction clears alarms from the alarm list. Figure 1.9.1 shows the format for the *alarm*, *almsg*, and *clear* instructions. Table 1.9.1 lists the types of alarms.

When you acknowledge an alarm, it is stored in the alarm list. When you acknowledge a cleared alarm, it is stored in the cleared alarm list.

**SYNTAX**
The general format is:

- `alarm(a,b,c)`
- `almsg(a,b,c)`
- `clear(a,b,c)`

where
- `a` is a numeric value designating the alarm type to be generated. (See Table 1.9.1).
- `b` is a numeric value (0-15) designating the alarm priority (0 for highest priority). A variable can represent the value.
- `c` is the string expression (up to 21 characters) that is to appear in the alarm message. String expressions over 21 characters are concatenated to 21 characters.

**NOTE:** The clear instruction must have the same parameter values as the *alarm* instruction it clears.

**SCRIPT TYPES**

*alarm* can be used in the following script types:

- PROC LIBR FORM
- RBLC START UNIT

**DEMONSTRATION OF FUNCTION**

An alarm instruction with a text string:

- `alarm(3,1,"Cooling Failure")`

An alarm instruction with a string variable:

- `badcool$="Cooling Failure"
  alarm(3,1,badcool$)`

An alarm instruction with a formatted variable:

- `alias temp
  badcool$=print$("Cooling Failure %f degrees",temp)
  alarm(3,1,badcool$)`

**Figure 1.9.1. alarm Instruction Format**
### Table 1.9.1. Alarm Traps

<table>
<thead>
<tr>
<th>Value</th>
<th>Alarm Type</th>
<th>Value</th>
<th>Alarm Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Advisory alarm</td>
<td>5</td>
<td>Disk Event List alarm</td>
</tr>
<tr>
<td>2</td>
<td>Critical alarm</td>
<td>6</td>
<td>Undefined</td>
</tr>
<tr>
<td>3</td>
<td>Hardware alarm</td>
<td>7</td>
<td>Undefined</td>
</tr>
<tr>
<td>4</td>
<td>System alarm</td>
<td>8</td>
<td>Batch alarm</td>
</tr>
</tbody>
</table>
The *event* instruction creates an entry in an event list. Figure 1.9.2 shows the format for the *event* instruction.

**SYNTAX**

The general format is:

```
event(a,b,c)
```

where

- `a` is a numeric value (0–255) designating the event type.
- `b` is a numeric value (0–15) designating the alarm priority. This field is not useful at this time. Enter 0 in this field.
- `c` is the string variable or the entire string (up to 21 characters enclosed within quotation marks) that is to appear in the alarm message.

**SCRIPT TYPES**

*event* can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

**DEMONSTRATION OF FUNCTION**

An event instruction with a text string:

```
event(82,1,"Mix completed")
```

An event instruction with a string variable:

```
heat$="Mix heat cycle begun"
alarm(82,1,heat$)
```

*Figure 1.9.2. event Function Format*
The *flog* (file log) instruction is used to enter log messages to batch log files on the disk. The *flog* instruction can also be used to create a new batch log file. Log files are stored in the Log folder. The *flog* instruction is normally used for equipment maintenance records on plant units.

The *flog* instruction is sent to all consoles on the data highway. Each time the *flog* instruction is executed, one entry is added to a specified batch log file on each console node that is configured to receive messages from the CP. (Console nodes are configured on the Plant Units Configuration screen.)

The entry is one line of text that is time-stamped. Entries are displayed on the Batch Log screen in chronological order with the most current entry at the bottom of the list. A new file is created at any console owning the CP if the *flog* instruction is used with a file name that does not currently exist within the log file folder.

The *flog* instruction can be used to log entries to any log file including the $$BATCH Log File. The $$BATCH Log File is the system default file that automatically records status, alarm, and error conditions. The default size of a batch log file is 1000 lines. (For more information on the $$BATCH Log File, see BA: 3-1.)

**NOTE:** If you are using Multitube pair or triple consoles, all consoles must be listed in the Plant Status or Plant Unit Configuration screen as owning the task. This insures that if the disk master changes between the consoles, the other console can still log the *flog* instruction entries to the disk.

Figure 1.9.3 shows the format for the *flog* instruction.
SYNTAX
The general format is:

\[ \text{flog}("x", y, a) \]

where

- \( x \) is the batch log file name.
- \( y \) is the print message that is to be copied to the log file and the optional format requirements for that message.
- \( a \) is an optional variable containing a value that is to be converted to a string value and included in the print message at the % designator.

SCRIPT TYPES

The \textit{flog} command can be used in the following script types:

- \text{PROC} \quad \text{START} \quad \text{UNIT}
- \text{LIBR} \quad \text{FORM}

SAMPLE INSTRUCTION

\[ \text{flog}(\text{$$BATCH$"}, \text{"Chained %d times"}, \text{count}) \]

DEMONSTRATION OF FUNCTION

shared local number
shared string \( a\$

\[
\text{number} = 46 \\
\text{a$} = \text{print$}("\text{Variable is %d"}, \text{number}) \\
\text{flog}(\text{$$BATCH$"}, \text{"%s"}, \text{a$}) \\
\text{"Log file $$BATCH contains this line: 7/15 17:49:46 Variable is 46"}
\]

Figure 1.9.3. \textit{flog} Instruction Format
open and close

The *open* and *close* instructions are used to open and close report files and generate reports. Figure 1.9.4 shows the format for the *open* and *close* instructions.

The *open* instruction can open the following types of report files:

- **Existing report files for reports generated by ControlBlock alarms and events.** These report files are configured on the Report Configuration screen. The *open* instruction begins another report generation. If the report configuration has “Autoprint ⇒ yes” specified, the report is automatically printed when the report file is closed or when the batch task finishes.

- **New report files created by RBL scripts.** If the *open* instruction is used to open a report file that does not exist, the file is then created and put into the Reports folder. The report can be viewed and printed from the Report Read Directory like other reports.

- **Existing report files created by RBL scripts.** If the *open* instruction is used to open a report file that has been created by an RBL script, another generation of the report is created. Each time that the *open* instruction is used, another generation of the report file is created. Each RBL-created report file can have a maximum of ten report file generations.

**NOTE:**

- The *open* instruction append option can be used to open a report file that has been previously closed without creating a new generation of the report file. This option is useful for reopening a report file that is inadvertently closed if the Coordinator Processor or console should reboot or fail during processing of the script, or if a Multitube console should switch disk drives.

- In a Working Recipe, if a report file is closed before the execution of a new Unit-Process icon, any subsequent attempt to append to the report file will create a new report. This occurs because the Unit-Process icon changes the batch task that runs the recipe. If you want to continue to write to the same report after changing Unit-Process icons, do not close the report before Unit-Process icons are switched.

For information on using redundant reports, see BA: Appendix D.
SYNTAX
The general format of an open instruction is:
open(w,"x","y",z)
where
  w is an arbitrary number from 1 to 10 specifying one of 10 active report file names.
  x is the volume name of the disk (y may be defaulted or entered as node:drive).
  y is the report file name.
  z is an optional append action performed by the open instructions. This field can include the following actions:

  0 = Open a new report file. It is the default action and is performed if no action is specified.
  1 = Append a report file. It enables an open instruction to open a report file that has been previously closed without creating a new generation of the report file.

NOTE: Three conditions are required in order for a report file to be reopened with an append option:
• The closed report must exist in the specified volume and file.
• The report must have been created by the same batch task.
• The report must have same number (w) as specified in the open instructions.

The general format of a close instruction is
close(x)
where
  x is a report reference number (created by an open instruction) from 1 to 10.

SCRIPT TYPES
open and close can be used in the following script types:

PROC    LIBR    FORM
RBLC    START    UNIT

SAMPLE INSTRUCTION
open(1,"CCBOOT30","PAINT"); “Create a new report file”
close(1); “Close report file”

Figure 1.9.4. open and close Instruction Format
(continued on next page)
DEMONSTRATION OF FUNCTION

If the Coordinator Processor (CP) should fail or reboot during processing of a batch script, the batch console automatically closes the report file. When the script resumes processing, a subsequent print instruction directs processing to the on file_n_err trap to reopen the report file. The append option (1) in the open statement reopens the last generation of the report file. The return instruction directs processing back to the print instruction.

```bash
shared local status
on file_n_err
    while ((status=open(1,"CCBOOT30","PAINT",1))<0); "reopen report file"
    return
open(1,"CCBOOT30","PAINT",1)
print(1,"This is my report entry"); "CP reboots at this point"
```

Figure 1.9.4. open and close Instruction Format (continued)

open and close Status Values

The open and close instructions return numerical status values that can be checked in the script to see if the instruction was successful. Table 1.9.2 describes the status values for open and close instructions.

Table 1.9.2. Status Values for open and close Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>1</td>
<td>The open instruction successfully opened a report file.</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>The open instruction failed to open a report file.</td>
</tr>
<tr>
<td>close</td>
<td>1</td>
<td>The close instruction successfully closed a report file.</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>The close instruction failed to close a report file.</td>
</tr>
</tbody>
</table>
The `print` and `print$` instructions determine the format of information in the generated report.

**print**

The `print` instruction makes an entry to an opened report file or the Batch Input screen. These entries record batch or RBLC processes.

Figure 1.9.5 shows the format for the `print` instruction. Sample instructions demonstrate the use of the `print` instruction to print report entries and generate entries to the Batch Input screen.
SYNTAX
The general format is:
print(x,"y",z)
where
for a report file:

- x is a number (1–10) referring to the active report file (created by the open instruction)
- y includes the print message that is to be copied to the report file and the print option characters, if required, for that message.
- z is an optional variable which includes the value that is to be converted to a string variable. A format option must be used with the z variable to specify the format type of the variable. For example, a %f format option indicates that a floating point variable is to be converted to a character string variable.

For the Batch Input screen:

- x is an asterisk (*) that designates the print message is to be sent to the Batch Input screen. (For batch input this statement is used with a prompt instruction.)
- y includes the print message that is to be copied to the report file and the print option characters, if required, for that message.

SCRIPT TYPES
print can be used in the following script types:

- PROC
- LIBR
- FORM
- RBLC
- START
- UNIT

DEMONSTRATION OF FUNCTION
print(1,"\nStart ON procedures"); “Print report to report file”
print(*,"\nEnd ON procedures"); “Print report to Batch Input screen”
print(1,"\nCurrent task ID is TASK %ID"); “%ID marks where ID value appears in string”
shared local x
x=4.5
print(1,"\nAdd %.1f Gallons",x); “%.1f marks where x floating point value appears in string”
“1 field width left of decimal, 1 decimal place for floating point”
Output reads:
Start ON procedures
End ON procedures
Current task ID is TASK 4
Add 4.5 Gallons

Figure 1.9.5. print Instruction Format
The `print$` instruction converts a variable value, print message, or string format modifier (for example, %TAG) to a character string value and assigns the character string value to a string variable. When converting a numeric variable value to a string value, the `print$` instruction must include a format option to specify the format type of the variable. Figure 1.9.6 shows the `print$` instruction format.

**SYNTAX**

The general format is:

```
x$=print$("y",z)
```

where

- `x$` is a string variable to which the `print$` instruction writes a character string message.
- `y` is the print message that is to be converted to a string variable and the print option characters, if required, for that message.
- `z` is an optional variable that includes the value to be converted to a string variable. A format option must be used with the `z` variable to specify the format type of the variable. For example, a %f format option indicates that a floating point variable is to be converted to a character string variable.

**SAMPLE INSTRUCTION**

```
TASK$=print$("%TAG")
TIME1$=print$("%s",state$)
```

**DEMONSTRATION OF FUNCTION**

In the example below, the `print$` instruction converts a decimal signed integer to a string variable.

```plaintext
x=1.1
a$=print$("variable is %1.1f",x); "% 1.1f marks where x floating point value appears in string"
  "1 field width left of decimal, 1 decimal place for floating point"
print(1,"%s",a$); "print a$ to report file"
alarm(2,1,a$); "Generate a$ as alarm"
```

Figure 1.9.6. `print$` Instruction Format
Print Options

Print options can be used with *print* instructions to designate report values and formats.

Options are designated by special characters in the *print* instruction character string. The character can specify specific format requirements or designate where converted variables will appear in the character string.

Figure 1.9.7 shows the characteristics of print options.

Figure 1.9.7. Characteristics of Print Options
Format Control Options

Format control options specify particular printing options. Table 1.9.3 lists and describes the format control options available.

Example: print(*,“^n ^t ^1r print message”)

<table>
<thead>
<tr>
<th>Form Control Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>A required character to designate the beginning of a format control specification.</td>
</tr>
<tr>
<td>n</td>
<td>Designates printing to begin on a new line.</td>
</tr>
<tr>
<td>t</td>
<td>Designates a horizontal tab (8 columns per tab).</td>
</tr>
<tr>
<td>x</td>
<td>Specifies the color of the print statement, where x is a name of the color (white, red, etc.) or the number of the color (1n, 1r, etc.) displayed on the Configuration Color Palette screen.</td>
</tr>
</tbody>
</table>
Constant Conversion Options

Constant conversion options convert system defined values into character string output. The % designator option marks where the converted value will appear in the string. Table 1.9.4 lists and describes the constant conversion options available.

Example: DISK$=print$("%VOL")
        RECS=print$("%RTAG")
        status=open(1,DISK$,RECS$,0)
        print(1,"Running Recipe %RTAG")

<table>
<thead>
<tr>
<th>Format Instruction</th>
<th>Data Entered</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Percent sign, no conversion</td>
</tr>
<tr>
<td>%DATE</td>
<td>Current date</td>
</tr>
<tr>
<td>%FILE</td>
<td>File name</td>
</tr>
<tr>
<td>%ID</td>
<td>Batch ID string or RBLC block address</td>
</tr>
<tr>
<td>%MAT</td>
<td>Material name of current M-Icon</td>
</tr>
<tr>
<td>%NODE</td>
<td>Current batch node number</td>
</tr>
<tr>
<td>%OPER</td>
<td>Current operation name</td>
</tr>
<tr>
<td>%RTAG</td>
<td>Working Recipe Tag</td>
</tr>
<tr>
<td>%SCRIPT</td>
<td>Script name</td>
</tr>
<tr>
<td>%TAG</td>
<td>Batch tag</td>
</tr>
<tr>
<td>%TASK</td>
<td>Current batch task number</td>
</tr>
<tr>
<td>%TIME</td>
<td>Current time</td>
</tr>
<tr>
<td>%VOL</td>
<td>Script Volume</td>
</tr>
</tbody>
</table>
Decimal Conversion Options

Decimal conversion options specify the format of character string decimals. Table 1.9.5 lists and describes the decimal conversion options available.

Syntax: \texttt{print("\% -0nn.dd.x",y)}

Example: \texttt{Script Configuration}
val=1010/3
print(1,"\n Right Justify \%04.3f",val)
print(1,"\n Left Justify \%- -4.3f",val)

\texttt{Report Print Out}
Right Justify 0336.667
Left Justify 336.667

Table 1.9.5. String Format Conversion Specifications

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left justification of the string (default is right justification).</td>
</tr>
<tr>
<td>0</td>
<td>Zeros are used like spaces to pad out any unused places in the designated field width. For example, if the field width is 3 digits and the current parameter value is 12, the parameter is displayed as 012.</td>
</tr>
<tr>
<td>\texttt{nn}</td>
<td>Digits that specify the minimum width of the field to the left of the decimal. The field width will automatically default to a larger width if necessary. \textbf{NOTE:} If you assign less spaces to \texttt{nn} than required by the actual number, the decimal conversion will take decimal places from the right of the decimal point (\texttt{dd}) and add them to the left of the decimal point (\texttt{nn}) in order for \texttt{nn} to default to a larger width.</td>
</tr>
<tr>
<td>.\texttt{dd}</td>
<td>Digits that specify precision in addition to \texttt{nn}. The \texttt{dd} digits are preceded by a period(\texttt{.}).</td>
</tr>
</tbody>
</table>
Variable Conversion Options

Variable conversion options convert variable values into character string output. There must be one % designator option for each variable converted in the string. The % designator option marks where the converted value will appear in the string. Table 1.9.6 lists and describes the variable conversion options available.

Example: print(1,"error number: %d %s",VALUE,A$

Table 1.9.6. Format Instructions

<table>
<thead>
<tr>
<th>Format Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Decimal signed integer</td>
</tr>
<tr>
<td>%u</td>
<td>Decimal unsigned integer</td>
</tr>
<tr>
<td>%o</td>
<td>Octal unsigned integer</td>
</tr>
<tr>
<td>%b</td>
<td>Binary unsigned integer</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal unsigned integer</td>
</tr>
<tr>
<td>%e</td>
<td>Floating point number in exponential notation</td>
</tr>
<tr>
<td>%f</td>
<td>Floating point number</td>
</tr>
<tr>
<td>%s</td>
<td>Character string</td>
</tr>
</tbody>
</table>
System String Options

System strings are strings whose values are already defined. These strings can be used to print particular types of data. They are similar to constant conversion options except that they cannot be included directly within the character string. A % designator option must mark where the converted value will appear in the string. Table 1.9.7 lists and describes the system string options available.

Example: REC$=print$("R_%s",RTAG$)
status=open(1,DISK$,REC$,0)
print(1,"Running Recipe %s",RTAG$)

a$="Date of run is"
data$=strcat$(a$,DATE$)
print(1,"^n %s",data$)

Table 1.9.7. String Formats (Fixed)

<table>
<thead>
<tr>
<th>Format Instruction</th>
<th>Data Entered</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKUP$</td>
<td>Backup volume specified on the Batch Configuration screen</td>
</tr>
<tr>
<td>DATE$</td>
<td>Current date</td>
</tr>
<tr>
<td>FILE$</td>
<td>File name</td>
</tr>
<tr>
<td>ID$</td>
<td>Batch ID string or RBLC block address</td>
</tr>
<tr>
<td>MAT$</td>
<td>Material name of current M-Icon</td>
</tr>
<tr>
<td>NODE$</td>
<td>Current batch node number</td>
</tr>
<tr>
<td>OPER$</td>
<td>Current operation name</td>
</tr>
<tr>
<td>PRIMARY$</td>
<td>Primary volume specified on the Batch Configuration screen</td>
</tr>
<tr>
<td>RTAG$</td>
<td>Working Recipe tag</td>
</tr>
<tr>
<td>SCRIPT$</td>
<td>Script name</td>
</tr>
<tr>
<td>TAG$</td>
<td>Batch tag</td>
</tr>
<tr>
<td>TASK$</td>
<td>Current batch task number</td>
</tr>
<tr>
<td>TIME$</td>
<td>Current time</td>
</tr>
</tbody>
</table>
quietlog and loudlog

The *quietlog* and *loudlog* instructions enable the processor to filter messages sent from the batch task to the BATCH LOG on the console disk. The *quietlog* instruction filters messages so that only messages that report run time errors are sent to the Batch Log. Messages are filtered to prevent batch processing messages from overloading the BATCH LOG and consequently losing messages. The *loudlog* instruction allows all messages to be sent to the BATCH LOG.

In extreme cases, the batch messages may overwhelm the console disk and interfere with normal processing of batch tasks. If messages are lost, the processor generates the following alarm message:

```
System alarm input log queue folder message lost
```

As a precaution, *quietlog* instructions should be included in those task responsible for generating high levels of batch messages. If the condition persists, the batch log may have to be disabled from the Console Configuration screen. (For more information on configuring the Console Configuration screen, see CC: 1-1.) The choice of solution depends on the level of message traffic entering the Batch Log; some experimentation may be required to achieve an effective level of control.

Figure 1.9.8 shows the format of the *quietlog* and *loudlog* instructions.
SYNTAX
The general format is

<table>
<thead>
<tr>
<th>loudlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>quietlog</td>
</tr>
</tbody>
</table>

**NOTE:** loudlog is the default condition of all tasks.

SCRIPT TYPES

*quietlog* and *loudlog* can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>LIBR</th>
<th>FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>UNIT</td>
<td></td>
</tr>
</tbody>
</table>

DEMONSTRATION OF FUNCTION

shared local T1
T1=0; “Set time to 0”
until time==30
quietlog
  T1=time 1; “read elapsed time in seconds”
loudlog

Figure 1.9.8. quietlog and loudlog Instruction Format
Section 10: 
Acquire/Release Instructions

This section describes the use of acquire/release instructions. This section shows a sample use of acquiring a task, instruction status conditions, and the instruction formats. The Batch Acquire Queues screen displays acquire/release information for batch tasks. (For information on using the Batch Acquire Queues screen, see BA: 3.)
Overview

The acquire/release instructions are used to control one batch task from one or more batch tasks. They are useful when more than one task share a common task (see Figure 1.10.1). When the common task is acquired by another task, it is called the slave task. The task acquiring the slave task is called the master task. A slave task may be owned by only one master task at a time. A master task may acquire any number of slave tasks.

In order for a master task to use a slave task, it must first acquire the slave task. The master task issues an acquire request by executing an acquire instruction. The request is entered into the request queue of the common task. If the common task is currently a slave of another task, the master task must wait to acquire the common task. The slave task processes acquire requests in the order in which it receives them (the one exception is a request issued by an acqfirst instruction, which is always processed first). When the slave task has finished processing a request, it releases itself from control of the master and processes the next acquire request in its queue, accepting the task that issued that request as its new master.

NOTE: The master and slave tasks can be on different nodes on the PeerWay.
Example of Acquiring a Task

The acquire/release instructions are used to control one batch task from one or more batch tasks. Figure 1.10.1 shows a sample use of an acquire/release application. In Figure 1.10.1, reactor 1 and reactor 2 are used to make paint. The batch process of making paint in reactor 1 is configured in task 4. The batch process of making paint in reactor 2 is configured in task 6.

Both task 4 and task 6 require that the reactor be drained. Instead of duplicating program lines in task 4 and task 6, task 12 has been configured to drain the reactor tanks. Both task 4 and task 6 use acquire/release instructions to use task 12. Only one task can use task 12 at any one time.

![Diagram of reactors and task configurations](image)

Figure 1.10.1. Performing Multiple Batch Tasks
Acquire/Release Concept

The task that is acquiring another task is called the master task. The task that is being acquired is called the slave task. Figure 1.10.2 shows the relationship between a master task and a slave task and how the acquire/release instructions are used.

Figure 1.10.2. Batch acquire/release Instruction Concept
Acquire/Release Status Values

The acquire/release instructions return numerical status values that can be assigned to a variable and checked in the script to see if the instruction was successful. The status value is useful for identifying logic errors in the slave and master tasks and when acquire/release instructions are not properly synchronized between scripts. In addition, status values for acquire and acqfirst can monitor the timeout status of an acquire request in the queue of the slave task. Table 1.10.1 describes the status values for acquire/release instructions.

Table 1.10.1. Status Values for acquire/release Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire</td>
<td>1</td>
<td>The master task acquired a slave task that was suspended by an acqwait instruction.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The master task currently owns the slave task.</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>The acquire request was successfully issued and is currently waiting in the queue of the slave task.</td>
</tr>
<tr>
<td></td>
<td>-2</td>
<td>The acquire request was issued by a previous acquire instruction and is waiting in the queue of the slave task. The time set for the acquire request has not yet expired.</td>
</tr>
<tr>
<td></td>
<td>-3</td>
<td>The time set for the acquire request has expired. The acquire request is still in the queue of the slave task.</td>
</tr>
<tr>
<td></td>
<td>-4</td>
<td>The slave task is unavailable. The acquire request was not placed in the queue of the slave task. (This only applies when an asterisk is used in the time field, for example, acquire(t,*).)</td>
</tr>
<tr>
<td>acqwait</td>
<td>1</td>
<td>The slave task waited until it was acquired by the master task.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The slave task had already been acquired by a master when acqwait was executed. No suspension of the task occurred.</td>
</tr>
<tr>
<td>Instruction</td>
<td>Status</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>dequeue</td>
<td>1</td>
<td>The master task had an acquire request waiting in the request queue of the slave task when dequeue was issued. The waiting acquire request was removed.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The task did not have an acquire request waiting in the request queue of the slave task when dequeue was issued. No action was taken.</td>
</tr>
<tr>
<td>masterof</td>
<td>1</td>
<td>The task is currently master of the slave task.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The task is not currently master of the specified task.</td>
</tr>
<tr>
<td>release</td>
<td>1</td>
<td>The slave task had a master when release was issued. The slave task successfully released from the master task.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The slave task did not have a master when release was issued. No action was taken.</td>
</tr>
<tr>
<td>relwait</td>
<td>1</td>
<td>The task was the master of the slave task when relwait was issued. The slave completed processing and released from the master.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The task was not the master of the slave task when relwait was issued. No action was taken.</td>
</tr>
<tr>
<td>unacquire</td>
<td>1</td>
<td>The task was master of the slave task when the unacquire was issued. The master and slave relationship was terminated.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>The task was not master of the slave task when the unacquire was issued. No action was taken.</td>
</tr>
</tbody>
</table>
acquire and acqfirst

The acquire instruction is used by the master task to acquire control of a slave task. The slave task can be any task accessible from the PeerWay. The acquire instruction places the master request into the queue of the slave task. The acquire instruction also contains a time expression that registers a timeout status if the slave task is not acquired within the specified amount of time. This error condition can be used in the master task to recognize timing problems so that corrective action can be initiated (see to the discussion on instruction status in this section).

The acqfirst instruction performs the same function as the acquire instruction. However, it places the master’s request at the top of the slave queue, giving the acqfirst request priority over all other acquire requests in the slave queue that are issued by other master tasks.

Figure 1.10.3 shows the format for the acquire and acqfirst instructions.

**SYNTAX**
The general format is:

```
x=acquire("y",z)
x=acqfirst("y",z)
```

where x is a variable used to check instruction status. Use is optional though recommended.

y is the batch tag of the task to be acquired. You can use either a text string or a string variable. Text strings must be enclosed in quotes.

z is an integer value for the queue time assigned to the acquire request. Queue time cannot exceed 21 days. This field can include the following time units:

- * = no queue wait time
- 0 = no queue timeout status returned
- S = seconds
- M = minutes
- H = hours
- D = days

**NOTE:** The timer rules are as follows:

- If time units (S,M,H, or D) are not specified, seconds are assumed.
- If the asterisk (*) is used, the acquire request checks whether the slave task is suspended by an acqwait instruction and immediately available for use. If the slave task is not available, the master request is not entered into the slave queue.
- If a queue time of 0 is used, the request is placed in the queue with no timeout attached. The acquire instruction will not return a timeout status.

Figure 1.10.3. acquire and acqfirst Instruction Format
**SCRIPT TYPES**

*acquire* and *acqfirst* can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>START</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBR</td>
<td>FORM</td>
<td></td>
</tr>
</tbody>
</table>

**SAMPLE INSTRUCTIONS**

status=acquire("TT-1",*); “Acquire the slave task if it is currently available.”
status=acquire("TT-1",500S); “The acquire request has a queue time of 500 seconds”
status=acquire("TT-1",3D12H4M15S)
“The acquire request has a queue time of 3 days, 12 hours, 4 minutes, and 15 seconds”
status=acqfirst("TT-1",500S)
“The acquire request is placed at the top of the slave queue ahead of all other requests”

**DEMONSTRATION OF FUNCTION**

on {=2b-12/a}; “if block goes into alarm”
unacquire("UNLOADER"); “unacquire slave task”

_shared local status; “Variable used to check instruction status”
if ((status=acquire("UNLOADER",60M))>=0)
  relwait; “wait here for slave release”
else; “some other task has acquired the slave task”
  sleep(600)
  goto _UNLOAD; “try to acquire slave again”

**Figure 1.10.3. acquire and acqfirst Instruction Format (continued)**

**NOTE:** A slave task will only allow one acquire request at a time from a given master task in its acquire queue. Until the waiting request is cleared from the queue, the slave task will ignore subsequent acquire requests from that task. However, subsequent executions of the *acquire* instruction are useful for checking the status of the original acquire request in the queue.
The *acqwait* instruction causes the slave task to wait until a master task issues an *acquire* instruction for the slave. Figure 1.10.4 shows the format for the *acqwait* instruction. The *acqwait* instruction can return a status value that can be checked to see if the instruction was successful.

**SYNTAX AND SAMPLE INSTRUCTION**
The general format is:

```plaintext
x=acqwait
```

where `x` is a variable used to check instruction status. Use is optional though recommended.

**SCRIPT TYPES**
*acqwait* can be used in the following script types:

```plaintext
PROC START UNIT
LIBR FORM
```

**DEMONSTRATION OF FUNCTION**

- shared local status; “Variable used to check instruction status”
- `status=acqwait; “Suspend slave task and wait to be acquired by master task”`

*Figure 1.10.4. acqwait Instruction Format*
**dequeue**

The *dequeue* instruction is executed by the master task to remove an acquire request from the request queue of the slave task. It may be useful to terminate a request, for example, if a master task aborts processing. Figure 1.10.5 shows the format for the *dequeue* instruction.

**NOTE:** The master task must check the status value of the *dequeue* instruction to make sure that the master request has been removed from the queue of the slave task. This precaution is necessary because the slave task may grant control to the master task before the master task can execute the *dequeue* instruction. Read Task Synchronization later in this Section thoroughly before using the *dequeue* instruction.
DEMONSTRATION OF FUNCTION

The following procedure waits for an acquire to happen. If the request timer runs out of time (times out) before the slave task becomes available, the dequeue attempts to remove the request.

NOTE: A status check must always be included to avoid timing problems.

shared local status; “Variable used to check instruction status.”

ABORT_PHASE: “Phase uses timeout indication.”

while ((status=acquire("SLAVE",3M))<0); “while acquire request in queue, check status”

if status == - -3; “If acquire request has timeout but still in queue”

status=dequeue("TT- -1"); “Delete request from queue.”

if (status==0);

“The dequeue did not delete request. This task is now the master.”

“The slave task must be unacquired.”

unacquire("SLAVE")

else; “dequeue worked, redirect script execution”

start SLAVE_DELAYED

sleep(1)

SLAVE_DELAYED:

“Perform phase activities”

SYNTAX

The general format is:

x=dequeue("y")

where x is a variable used to check instruction status. Use is required.

y is the tag of the slave task for which an acquire request has been issued. You can use either a text string or a string variable. Text strings must be enclosed in quotes.

SCRIPT TYPES

decue can be used in the following script types:

PROC START UNIT
LIBR FORM

Figure 1.10.5. dequeue Instruction Format
**masterof**

The `masterof` instruction checks whether the task is the master of a specified slave task. The `masterof` instruction returns a status value that can be checked in the script to determine ownership:

- **TRUE (1)** The task is the master of the specified task
- **FALSE (0)** The task is not the master of the specified task.

The `masterof` instruction is useful in logic instructions (*if, while* and others) for initiating actions if the `masterof` status value is true. Figure 1.10.6 shows the format for the `masterof` instruction.

**SYNTAX**

The general format is:

```plaintext
x=masterof("y")
```

where `x` is a variable used to check the instruction status. A status variable is required. `y` is the tag of the slave task being checked. You can use either a text string or a string variable. Text strings must be enclosed in quotes.

**SCRIPT TYPES**

`masterof` can be used in the following script types:

- `PROC` START UNIT
- `LIBR` FORM

**DEMONSTRATION OF FUNCTION**

```plaintext
shared local status; "Variable used to check instruction status."

if ((status=masterof("slave1"))=1)
    start SLAVE1_PROC
else if ((status=masterof("slave2"))=1)
    start SLAVE2_PROC
else
    start ERROR
```

Figure 1.10.6. `masterof` Instruction Format
The `on acquire_err` instruction executes in response to a faulty acquire condition that should not occur under normal operating conditions. An `on acquire_err` trap can be used in both the master task and the slave task. An acquire fault can occur if the master task cannot log an acquire request to the acquire queue of the slave task or the master task is killed.

**NOTE:** The `on acquire_err` trap can be used to detect if either the master task or slave task is killed. However, the use of the `on lost_master` and `on lost_slave` is preferred, because they can detect the loss of a task more accurately than `on acquire_err`.

Figure 1.10.7 shows the format of the `on acquire_err` instruction.

### SYNTAX
The general format is:

```plaintext
on acquire_err
```

### SCRIPT TYPES

`on acquire_err` can be used in the following script types:

- `PROC` `START` `UNIT`
- `LIBR` `FORM`

### DEMONSTRATION OF FUNCTION

```plaintext
on acquire_err

"Error. This trap should never be executed"
msg$=print$("%TAG acquire error")
alarm(8,1,msg$); "Send batch alarm"
hold; "Take necessary corrective action"
```

Figure 1.10.7. `on acquire_err` Instruction Format
on lost_master

The *on lost_master* instruction is an *on* trap that executes if the slave task prematurely loses its master task. The slave task can use *on lost_master* to release itself from a lost master task.

A master task is considered to be lost prematurely if it is terminated before it can execute an *unacquire* instruction or before the slave task can execute a *release* instruction. A master task may be lost because:

- The Master/Slave task relationship is terminated at the Batch Acquire Queues Screen.
- The Master task is killed, finishes, or is put in Idle state.
- The node on which the master task resides is wiped manually or wiped by a bad nonvolatile memory board during a reboot.

**NOTE:** When using *on lost_master*, the following rules apply:

- If the master task is lost, the slave task will execute the *on lost_master* trap only if it is the active slave of the lost master task. Therefore, the slave task cannot use an *on lost_master* trap to remove a master task entry that is waiting in its acquire queue. You must use the Batch Acquire Queue screen (B A Q) to remove the waiting entry from the slave task acquire queue manually.
- An *on acquire_err* trap can also be used to direct execution if the master task is killed. If both an *on acquire_err* trap and an *on lost_master* trap are used in the same task, only the *on lost_master* task will execute if the task is killed.

Figure 1.10.8 shows the format of the *on lost_master* instruction.
**SYNTAX**
The general format is:

```
on lost_master
```

**SCRIPT TYPES**

`:on lost_master` can be used in the following script types:

```
PROC START UNIT
LIBR FORM
```

**DEMONSTRATION OF FUNCTION**

If an active master task is lost, the slave task can use an `on lost_master` trap to release itself from the master task. The `release` instruction releases the slave task from control of the lost master task.

```
shared local status; "Variable used to check instruction status"

on lost_master
    status=release; "release slave from control of master"
    msg$=print$("%TAG lost master")
    alarm(8,1,msg$)
    hold; "Take necessary corrective action"
```

---

**Figure 1.10.8. on lost_master Instruction Format**
The *on lost_slave* instruction is an *on* trap that executes if the master task loses its slave task. The master task can use *on lost_slave* to unacquire a lost slave task or remove an entry for a lost slave task from its acquire queue.

A slave task is considered to be lost prematurely if it is terminated before it can execute a *release* instruction or before the master task can execute an *unacquire* instruction. A slave task may be lost because:

- The Master/Slave task relationship is terminated at the Batch Acquire Queues Screen.
- The Slave task is killed, finishes, or is put in the Idle state.
- The node on which the slave task resides is wiped manually or wiped by a bad nonvolatile memory board during a reboot.
Conditions for Execution

If the master task loses a slave task, the on lost_slave trap will execute only if the following conditions are true:

- The master task is the active master when the slave task is lost.
- The master task attempts to check the status of the acquire request in the slave task acquire queue after the slave task is lost. For example:
  - The master task executes an acquire instruction and returns a status value of -1, indicating that the acquire request has been entered into the slave task status queue.
  - The slave task is then lost.
  - If the master task executes the acquire instruction again to determine the status of the acquire request, the on lost_slave trap will execute.

The on lost_slave trap will not execute in the following situations:

- The master task attempts to acquire for the first time a task that has been wiped or does not exist.
- The acquire instruction uses the no timeout value (*) as follows: acquire(myslave,*)

In either case, the master task will not execute the on lost_slave trap, because there is no record of the acquire request in the slave task acquire queue. The master task will execute the acquire instruction repeatedly, causing the task to stay in a Retry mode until the slave reappears. To prevent this condition, you can include an on task_retry trap in the master task script.

NOTE: An on acquire_err trap can also be used to direct execution if the slave task is killed. If both an on acquire_err trap and a on lost_master trap are used for this purpose in the same task, only the on lost_master task will execute in response to this condition.

Figure 1.10.9 shows the format for the on lost_slave instruction.
SYNTAX
The general format is:

on lost_slave

SCRIPT TYPES

on lost_slave can be used in the following script types:

PROC START UNIT
LIBR FORM

DEMONSTRATION OF FUNCTION
If the slave task MYSLAVE is lost, a master task can use an on lost_slave trap to unacquire a slave task or remove an entry for a pending slave task from its acquire queue.

- If the status value of unacquire equals 1, the master task owns the slave task MYSLAVE and it will unacquire MYSLAVE.
- If the status value of dequeue equals 0, the master task request for the slave task is still waiting in its acquire queue. The master task will remove the slave task entry from its acquire queue.

```plaintext
shared string msg$
shared local status1,status2; "Variables used to check instruction status"

on lost_slave
   "master task lost slave task"
   if ((status1=unacquire("MYSLAVE$"))===1); "if task is an active master"
      msg$=print$("%TAG unacquired slave!")
      alarm(8,1,msg$)
      hold; "Take necessary corrective action"
   if ((status2=dequeue("MYSLAVE"))===0); "If task is waiting in slave task master queue"
      msg$=print$("%TAG dequeued slave")
      alarm(8,1,msg$)
      hold; "Take necessary corrective action"
   hold; "master Task is neither active nor waiting in slave task acquire queue"
```

Figure 1.10.9. on lost_slave Instruction Format
on unacquire

The on unacquire instruction redirects execution of the slave task script when the master task terminates a master and slave relationship with an unacquire instruction or when an operator terminates a master and slave relationship from the Batch Acquire Queues screen.

NOTE: The on unacquire instruction can be used to detect if a master and slave task relationship is terminated from the Batch Acquire Queues screen. However, on lost master and on lost_slave perform this function more effectively than on unacquire. If on unacquire is included in the same script with either the on lost master or on lost_slave trap, only the on lost master or on lost_slave trap will execute if either the master or slave task is terminated.

Figure 1.10.10 shows the format for the on unacquire instruction.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td>on unacquire</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCRIPT TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>on unacquire can be used in the following script types:</td>
</tr>
<tr>
<td>PROC START UNIT</td>
</tr>
<tr>
<td>LIBR FORM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>on unacquire</td>
</tr>
<tr>
<td>“Master has unacquired the slave task.”</td>
</tr>
<tr>
<td>“Terminate relationship with master task.”</td>
</tr>
<tr>
<td>“Wait to be acquired by the next master task in the queue.”</td>
</tr>
</tbody>
</table>

Figure 1.10.10. on unacquire Instruction Format
The *release* instruction releases a slave task from control of a master task. In addition, if the master task has suspended processing by executing a *relwait* instruction, the *release* instruction will allow the master task to continue processing. The *release* instruction returns a status value that can be checked to see if the slave task was successfully released from control of the master task. Figure 1.10.11 shows the format for the *release* instruction.

**SYNTAX**

The general format is:

\[ x = \text{release} \]

where \( x \) is a variable used to check the instruction status.

**SCRIPT TYPES**

*release* can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**DEMONSTRATION OF FUNCTION**

shared local status; “Variable used to check instruction status”

shared string msg$

\[ \]

ACQ: “phase subroutine”

while \((\text{status}=\text{acqwait})<1\); “Wait to be acquired by master”

\[ \]

status=release; “Release master task”

if \((\text{status}._=1)\); “if master task did not own slave task, determine why”

\[
\text{msg$} = \text{print$}(%\text{TAG \%d=release!}, \text{status})
\]

\[
\text{alarm}(8,1,\text{msg$})
\]

\[
\text{hold}
\]

“Return to acqwait and wait for next master”

start ACQ

![Figure 1.10.11. release Instruction Format](image-url)
The `relwait` instruction suspends execution of the master task until the slave task executes a `release` instruction. Figure 1.10.12 shows the format for the `relwait` instruction.

**SYNTAX**
The general format is:

\[ x=\text{relwait}("y") \]

where

- `x` is a variable used to check the instruction status.
- `y` is the tag of the slave task. You can use either a text string or a string variable. Text strings must be enclosed in quotes.

**DEMONSTRATION OF FUNCTION**

- shared local status; “Variable used to check instruction status”
- shared string `msg$
  
  .
  .
  .

  status=relwait(MYSLAVE$); “Wait here for slave release”

  “When operation is complete, slave task releases master task”

  if (status\_\_=1); “If master task did not own slave task, determine why”

  msg$=print$("%TAG %d=relwait!",status)

  alarm(8,1,msg$)

**SCRIPT TYPES**

`relwait` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**Figure 1.10.12. relwait Instruction Format**
The \textit{unacquire} instruction allows the master task to terminate the master and slave relationship. The \textit{unacquire} instruction redirects processing of the slave task to an \textit{on unacquire} instruction in response to a change in the process status. The \textit{unacquire} instruction returns a status value that can be checked to see if the master/slave relationship was terminated. Figure 1.10.13 shows the format for the \textit{unacquire} instruction.

**SYNTAX**

The general format is:

\begin{verbatim}
x=unacquire("y")
\end{verbatim}

where \( x \) is a variable used to check the instruction status.\( \quad y \) is the tag of the batch task to be unacquired.

**SCRIPT TYPES**

\textit{unacquire} can be used in the following script types:

<table>
<thead>
<tr>
<th>PROC</th>
<th>START</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBR</td>
<td>FORM</td>
<td></td>
</tr>
</tbody>
</table>

**DEMONSTRATION OF FUNCTION**

\begin{verbatim}
shared local status1; "Variable used to check instruction status"
shared string msg$
.
.
.on lost_slave
"Master task lost slave task"
"Master task active or waiting in queue when slave task lost"
if ((status1=unacquire(MYSLAVE$))=1); "If task is a master, unacquire slave task"
  msg$=print$("%TAG unacquired slave!")
  alarm(8,1,msg$)
  hold "Take necessary corrective action"
\end{verbatim}

\textbf{Figure 1.10.13. unacquire Instruction Format}
Hints for Using Acquire/Release Instructions

The following procedures explain several applications of acquire/release instructions.

Task Synchronization

Unless carefully configured, the master and slave tasks can develop timing problems in synchronizing activities. This is because the master and the slave tasks process instructions independently of each other and at independent rates.

The design of RBL insures that acquire/release instructions in the master and the slave are processed one at a time, even if the tasks are processing on different nodes. This prevents the tasks from unknowingly cancelling each other’s actions during processing. However, since either task can initiate a change in the acquire queue, timing problems can still result. Synchronization between tasks can be maintained by using logic instructions (if, while, and others) to check the status values returned by the acquire/release instructions and to direct the execution of subsequent instructions based on the status values.

For example, the slave task may reach an acqwait instruction and grant control to the master task before the master task can issue the dequeue instruction to remove the acquire request from the slave queue. The slave task will then assume that it has a master and will continue processing its script after the acqwait instruction. If this occurs, then the dequeue instruction executed by the master task will return a status of 0, indicating that the acquire request was not found, and therefore was not removed. The acquire request was not found because it has fallen out of the acqwait and has begun processing subsequent script instructions.

NOTE: Do not rely on the status value returned by the masterof or the acquire instruction to determine whether the dequeue instruction should terminate a master request. The masterof status value is unreliable because the slave task may grant control to the master task at some point between the execution of the masterof and the dequeue instructions or between the acquire and the dequeue instructions. Figure 1.10.14 demonstrates two cases that are prone to a timing problem:
DEMONSTRATION OF POTENTIAL TIMING PROBLEM

if masterof(SLAVE1)==0
dequeue("SLAVE1")

while ((rs=acquire("SLAVE1",time))<0) & (rs=-3)
if rs=-3
dequeue("SLAVE1")

Figure 1.10.14. Timing Problems Associated with Terminating a Slave Request

The dequeue status value should be used to determine whether the acquire request should be terminated. If the dequeue instruction returns a 0, the acquire request cannot be terminated because there was no waiting request in the slave queue or the slave task may now be processing its script. If the dequeue instruction returns a value of 1, the acquire request is in the queue of the slave task and can be deleted. Figure 1.10.15 and Figure 1.10.16 demonstrate two actions the master task can take if dequeue returns a status value of 1:

- Recheck the masterof status and if it has become TRUE (1), continue processing master routines as configured in the master script.

DEMONSTRATION OF VERIFYING SYNCHRONIZATION

if masterof("SLAVE1")==0
if dequeue("SLAVE1")==0
if masterof("SLAVE1")
start MASTERPHASE
else start SLAVE_MISSED

while ((rs=acquire("SLAVE1",time))<0) & (rs=-3)
if rs=-3
if dequeue("SLAVE1")==0
if masterof("SLAVE1")
start MASTERPHASE
else start SLAVE_DELAYED

Figure 1.10.15. Verify Synchronization and Continue Processing

- Issue an unacquire instruction to disconnect the master task from the slave task as follows:
DEMONSTRATION OF VERIFYING SYNCHRONIZATION

if masterof("SLAVE1")==0
  if dequeue ("SLAVE1")==0
    unacquire("SLAVE1")
  else start SLAVE_MISSED

while ((rs=acquire(SLAVE1,time))<0 )&( rs=- -3)
  if rs==-3
    if dequeue("SLAVE1")==0
      unacquire("SLAVE1")
    else start SLAVE_DELAYED

Figure 1.10.16. Verify Synchronization and Disconnect Master and Slave Tasks

The unacquire instruction transfers processing of the slave task to the on unacquire instruction, if one is included in the slave script. The unacquire instruction returns a status value of 1 if an active master and slave relationship is terminated.
Responding to Loss of a Master Task or Slave Task

Under certain circumstances, a master and slave relationship between two tasks may be prematurely terminated if either the master task or slave task is lost. These circumstances include:

- The master task or slave task is killed, finishes, or is put in Idle mode.
- The node on which master task or slave task resides is wiped manually or wiped by a bad nonvolatile memory board during a reboot.

If either the master or slave task is lost, you can use the `on lost_slave`, `on lost_master`, or the Batch Acquire Queues screen ([B] [A] [Q]) to terminate the master and slave relationship.
Effects of a Lost Master Task or Lost Slave Task

If ignored, a lost task can cause the following problems:

- If the active master task is lost, the slave task will continue to operate as if the master and slave relationship is still in effect. If the slave task executes a *release* instruction, it will release itself from the lost master and accept the next waiting task in its acquire queue (if one is waiting) as its master. The Batch Acquire Queues screen, Batch Monitor screen, and Batch Overview screen will display the following message in the the task "ID" field:

  -- LOSTMSTR --

- If the slave task is lost, the master task will wait indefinitely for the lost slave task on the currently executing line. If another task tries to acquire the missing slave task, that task will repeatedly execute the *acquire* instruction every second, causing a “Task Re-try” error. The Batch Acquire Queues screen, Batch Monitor screen, and Batch Overview screen will display the following message in the the task “Status” field:

  (hold) Lost Slave

  The lost slave task will continue execution when restarted.

**NOTE:** A master task and slave task become active tasks when the master task actually acquires the slave task. The tasks are considered as waiting until the master task acquires the slave task.

**WARNING**

If *on lost_slave* and *on lost_master* are used to terminate a master and slave relationship, you might need to specify other corrective actions in the batch script to maintain the integrity of the batch process, depending on the requirements of your process.
Sample Use of on lost_slave and on lost_master

In the following example, a master task and a slave task use on lost_slave and on lost_master traps to protect the batch process in the event that either task is lost.

- If the slave task is lost, a master task can use an on lost_slave trap to unacquire an active slave task or remove an entry for a waiting slave task from its acquire queue.

- If an active master task is lost, the slave task can use an on lost_master trap to release itself from the active master task.

To begin the operation, the master task executes an acquire instruction to acquire the slave task. The request is logged in the slave task acquire queue. The master task will then check the status of the acquire request at specified intervals to see whether it has acquired the slave task.

Figure 1.10.17 shows a flow chart representation of this program.
**Figure 1.10.17. Master Task and Acquire Task Flow Chart**

- **Master Task**
  - slave task is lost
  - execute acquire to acquire task or check status
  - if master task acquires slave task
    - if acquire request in queue and no timeout
      - issue an alarm
    - if acquire request in queue and timeout expires
      - hold
    - set on acquire_err trap
  - issue alarm, hold
  - continue task

- **Slave Task**
  - set or execute on lost_slave trap
  - if master of slave task, unacquire slave
  - release master task
  - set on unacquire trap

- **Flow Chart**
  - DRAIN: perform drain activities
  - execute relwait to wait for slave task release
  - if master task did not own slave task
    - issue alarm
  - if master task did own slave at release
    - perform ACQ phase activities
    - release master task
    - set on lost_master trap
  - start ACQ
  - if waiting for slave task, delete acquire request
  - if master task is lost
    - set on lost_master trap
  - start DRAIN

- **Instructions**
  - set on acquire_err trap
  - set on unacquire trap
  - if acquire request in queue and no timeout
  - if acquire request in queue and timeout expires
  - if master task acquires slave task
  - if master task did not own slave task
  - issue alarm
  - continue task
shared local status, status1, status2
shared string msg$, myslave$

on acquire_err
   "Error. This trap should never execute"
   msg$=print$("%TAG acquire error")
   alarm(8,1,msg$), "Send batch alarm"
   hold "Take necessary corrective action"

on lost_slave
   "Master task lost slave task"
   "Master task active or waiting in queue when slave task lost"
   if ((status1=unacquire(myslave$))==1); "If task is a master"
       msg$=print$("%TAG unacquired slave!")
       alarm(8,1,msg$)
       hold "Take necessary corrective action"
   if ((status2=dequeue(myslave$))==1); "If task is waiting in slave task master"
       msg$=print$("%TAG dequeued slave")
       alarm(8,1,msg$)
       hold "Take necessary corrective action"
   hold "Master task is neither active or waiting in slave task acquire queue"
       "This condition should not happen"

ACQ:
status=acquire(myslave$,15M); "Issue acquire request for the slave task"
if ((status==0)|(status==1)); "If master task acquired slave task or currently"
   "owns the slave task"
   start DRAIN
if ((status==-1)|(status==-2)); "If first time or subsequent time request in"
   "acquire queue"
   sleep(15)
   start ACQ; "Check status again"
if (status==-3); "If master task request timeout has expired"
   msg$=print$("%TAG time out")
   alarm(8,1,msg$)
   sleep(300)
   start ACQ; "Check status again"
   msg$=print$("%TAG acquire problem"); "Error. Status does match specified"
   "values"
   "Program should never execute this line"
   alarm(8,1,msg$)
   hold

DRAIN:
   "Master task successfully acquired slave task"
   "Master task is suspended while the slave task executes"
   "Slave task performs drain activities during master/slave relationship"
status=relwait(myslave$); "wait here for slave release"
   "When operation is complete, slave task releases master task"
if (status=-1); "If master task did not own slave task, determine why"
   msg$=print$("%TAG %d=relwait!", status
   alarm(8,1,msg$)

start PHASE2; "Go to next phase"
shared local status
shared string msg$

on unacquire
   "The master task should never unacquire the slave task"
   "The slave task releases the master task"
   msg$=print$("%TAG unacquired!")
   alarm(8,1,msg$)
   hold

on acquire_err
   "Master task waiting in slave queue. Remove manually on BAQ screen"
   msg$=print$("%TAG acq err")
   alarm(8,1,msg$)
   hold

on lost_master
   status=release; "get master out of my queue"
   msg$=print$("%TAG lost master")
   alarm(8,1,msg$)
   hold

ACQ:
while ((status=acqwait)<1);
   "Wait to be acquired by master"
   "Master task has acquired slave task. Master task remains suspended by a"
   "relwait instruction"
   "Perform a drain activity"
   "verify tank level"
   "open drain valve"
   "close drain valve"
   status=release; "Release master task"
   if (status=-1);
      "if master task did not own slave task, determine why"
      msg$=print$("%TAG %d=release!",status)
      alarm(8,1,msg$)
      hold
   "Return to acqwait and wait for next master"
   start ACQ
Removing Master Request From Slave Task Acquire Queue

If a master task is lost while its request is waiting, you will have to remove the entry for that master request from the slave task acquire queue manually on the Batch Acquire Queues screen ([B] [A] [Q]). An on lost_master trap cannot be used to remove a waiting master request.

The Batch Acquire Queues screen provides information about the status of master tasks and slave tasks. If a master task or slave task is terminated prematurely for any reason, the message

-- LOST! --

will appear highlighted in red in one of the following acquire queue fields for that task:

- My Master
- My Slave
- Tasks Waiting To Become My Slave
- Tasks Waiting To Become My Master

To remove a lost task from an acquire queue:

1. Put the lost master task (designated as -- LOST! --) and the slave task in the Hold mode. You can execute a hold command from the Batch Monitor screen for each task.
2. Move the cursor to the lost master task and press [CTRL] [D] to delete the entry for the master task from the acquire queue.

WARNING

Manually terminating a lost task from an active or pending acquire queue on the Batch Acquire Queues screen should be done only as a last resort, if the task cannot be terminated automatically in the script.

For more information on the Batch Acquire Queues screen, see BA: 3.

Passive and Active Masters and Slaves

Both the master task and slave task can take either a passive or active role in the master and slave relationship. The active task will perform some predefined activity; the passive task will wait in a suspended state until the active task completes the activity.
Section 11: Interactive Instructions

The interactive instructions enable you to interact directly with the batch tasks. They generate requests for information and then input the information to variables in the script. The operator enters the requested information from an interface screen: either the Batch Input screen or a process graphic BAINPUTW object.

The prompt and notify instructions access the interface screen by issuing an alarm. This alarm halts program execution until the interface screen for that task is brought up on the console. The screen then proceeds to display lines that are indented under the prompt or notify instruction, such as input and print instructions. An input or input$ instruction stops the program until the operator enters a response at the prompt on the interface screen.

The reprompt and renotify instructions perform the same function as the prompt and notify instructions, except that they do not issue the “Batch waiting for operator input” alarm while the interface screen is displayed.

For more information on the process graphic BAINPUTW object, see BA: 4. For more information on the Batch Input screen, see BA: 3.

NOTE: The input and input$ instructions are not interchangeable.
The *notify* instruction is used in conjunction with a *usegraphic* instruction to initiate communication between a library script and a BAINPUTW graphic object. The BAINPUTW graphic object displays a batch input window on the process graphic.

When executed, the *notify* instruction issues an alarm. The *usegraphic* instruction specifies the graphic file in which the BAINPUTW graphic object appears. The script must execute the *usegraphic* instruction first or the *notify* instruction will fail.

The *renotify* instruction performs the same function as the *notify* instruction except that it does not issue an alarm while the process graphic is displayed on the screen.

**NOTE:**

- *notify* and *renotify* are similar to the *prompt* and *reprompt* instructions. The *notify* and *renotify* instructions can access the Batch Input screen. However, the *prompt* and *reprompt* instructions cannot access the BAINPUTW graphic object.

- In ABC Batch, the Working Recipe must own any non-zero plant unit specified for the batch unit (on the Batch Unit Table). Otherwise, the *notify* instruction will fail.

- A blank line must follow the last line in a sequence of indented lines.

Figure 1.11.1 shows the format of the *notify* and *renotify* instructions.
### SYNTAX
The general format is:

```
notify
renotify
```

### SCRIPT TYPES
notify can be used in the following script types:

- PROC START UNIT
- LIBR FORM

### DEMONSTRATION OF FUNCTION
shared local runs, newrun
shared string opnam$
usegraphic(WINI-C,Graphic,0)
notify
  print(*,”%d Batch runs are complete.”,runs)
  input(“^n Enter Next Run Sequence Number:”,newrun)
  opnam$=input$(“Enter operation name:”)

---

**Figure 1.11.1.** *notify and renotify* Instruction Format
prompt, reprompt, and input

The *input* instruction allows you to enter numeric data from the Batch Input screen to a local, global, or array variable. The *input* instruction is indented under a *prompt* or *reprompt* instruction.

The *prompt* or *reprompt* instruction accesses the interface screen by issuing an alarm. The *reprompt* instruction performs the same function as the *prompt* instruction, except that it does not issue an alarm while the Batch Input screen is displayed.

The *input* instruction can copy a message string to the Batch Input screen. The *input* instruction designates a variable to receive the numeric value input by the operator. The possible entries also include Y(es), or N(o).

**NOTE:** A blank line must follow the last line in a sequence of indented lines.

Figure 1.11.2 shows the prompt, *reprompt* and *input* instruction formats.
SYNTAX
The general format for the prompt, reprompt and input instruction is:

```
prompt
reprompt
input("y",z)
```

where y is the message that is copied to the Batch Input screen and the optional format requirements for that message. The message can be a maximum of 22 characters.

z is a numeric variable that defines the operator-enterable field on the Batch Input screen. The variable also accepts "Y" and "N" as inputs. A value of 1 is returned for "Y" (or equivalent), and a value of 0 is returned for "N" (or equivalent).

NOTE: The input instruction is always indented 2 spaces under the prompt or reprompt instruction.

SCRIPT TYPES
prompt, reprompt, and input can be used in the following script types:

```
PROC       LIBR       FORM
START       UNIT
```

DEMONSTRATION OF FUNCTION
Sample script:
```
shared local runs,stflow,rerun,opshift
prompt; "Access Batch Run screen"
prompt; print(*,"%d Batch runs are complete.",runs)
prompt; print(*,"stflow=%f GPM.",stflow)
prompt; input("Do you want to run another Batch? (Y or N)",rerun)
prompt; input("Operator shift number?",opshift)
if rerun
   "Log operator shift number to report"
prompt; open(1,"CCBOOST","SHIFTRPRT")
prompt; print(1,"Operator shift number %d started Batch.",opshift)
```

Messages that appear on the Batch Input screen showing operator response:

5 Batch runs are complete.
stflow=75.3 GPM
Do you want to run another Batch> Yes
Operator shift number?>2

NOTE: A * symbol appears in the Batch Input screen where the manual entry should be made.
prompt, reprompt, and input$

The $input$ instruction allows you to enter character data from the Batch Input screen to a string variable. The $input$ instruction is indented under a prompt or reprompt instruction. The prompt or reprompt instruction accesses the interface screen by issuing an alarm. The $input$ can copy a message string to the Batch Input screen. The $input$ instruction designates a variable to receive the character string input by the operator.

**NOTE:** A blank line must follow the last line in a sequence of indented lines.

Figure 1.11.3 shows the prompt, reprompt, and $input$ instruction formats.
## SYNTAX

The general format for the `prompt`, `reprompt`, and `input$` instructions is:

```plaintext
prompt
reprompt
  y$ = input$("z")

where
  y$ is a string variable that receives data from the operator-enterable field on the Batch Input screen.
  z is the message that is to be copied to the Batch Input screen and the optional format requirements for that message (maximum length of 22 characters).
```

**NOTE:** The `input$` instruction is always indented 2 spaces from the `prompt` or `reprompt` instruction.

## SCRIPT TYPES

`prompt`, `reprompt`, and `input$` can be used in the following script types:

- `PROC`  `START`  `UNIT`
- `LIBR`  `FORM`

## DEMONSTRATION OF FUNCTION

Sample script:

```plaintext
shared string comp$.

prompt; "Access Batch Run screen"
  comp$ = input$("Component 1")
  "Log Component number to report"
  open(1,"CCBOOT","SHIFTRPRT")
  print(1,"%s now entering process:"comp$ )
```

**NOTE:** A " symbol is printed in the Batch Input screen where the manual entry should be made.

---

**Figure 1.11.3.** prompt, reprompt, and input$ Formats
Section 12:
Scaling Instructions

Scaling instructions are used in a batch script to represent analog input and output values in the internal scaling of the ControlBlock or in engineering units of measure for field devices in a batch process. These scales include the following:

**Plant Scaling**
The range of units of a field device that is being used to measure a substance. For example, a reactor tank with a volume of 1000 gallons has a plant scaling range of 0–1000 GAL.

**Internal Block Scaling**
A range from 0–1 that is used internally by the ControlBlock and for performing calculations and functions. For example, with an Eng Zero of .00 and an Eng Max of 1000, the value 500 is internally represented as .5 by the ControlBlock.

**Display Block Scaling**
A range that you configure for displaying internal ControlBlock values in terms of the plant scaling units on the faceplates and other console screens. For example, an internal block value of .5 is displayed as 500 GAL on a faceplate. You configure display block scaling using the “Eng Zero” and “Eng Max” fields on a Continuous Links screen.

For more information about scaling types, see CB: 3–2.
Table 1.12.1 shows some examples of different scaling used for field device values in a batch process.

<table>
<thead>
<tr>
<th>Field Device values</th>
<th>Plant Scaling</th>
<th>Internal Block Scaling</th>
<th>Display Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.00 mA</td>
<td>1000 GPM</td>
<td>1.0</td>
<td>1000 GPM</td>
</tr>
<tr>
<td>12.00 mA</td>
<td>500 GPM</td>
<td>0.5</td>
<td>500 GPM</td>
</tr>
<tr>
<td>5.6 mA</td>
<td>100 GPM</td>
<td>0.1</td>
<td>100 GPM</td>
</tr>
<tr>
<td>4.00 mA</td>
<td>0 GPM</td>
<td>0.0</td>
<td>0 GPM</td>
</tr>
</tbody>
</table>

**NOTE:** Scaling instructions cannot be linked directly to Input/Output blocks. To access an Input/Output block with a scaling instruction, link the Input/Output block address to a ControlBlock. The scaling instruction can then read or write to the Input/Output block through the ControlBlock.
**scdp**

The `scdp` instruction is used to assign the number of significant decimal places of a continuous input or output value in display block scaling to a declared variable.

Figure 1.12.1 shows the format for the `scdp` instruction.

---

### SYNTAX
The general format is:

```
a = scdp(x)
```

where
- `a` is a declared variable that receives the number of significant decimal places.
- `x` is the ControlBlock register of a continuous input or output value.

### SCRIPT TYPES
`scdp` can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

### SAMPLE INSTRUCTION
```
dec = scdp(in)
```

### DEMONSTRATION OF FUNCTION
```
shared local dec
shared alias in: TEMP/A
dec = scdp(in)
```

If the continuous input `A` of ControlBlock TEMP is 100.00, the value of `dec` is 2.

---

*Figure 1.12.1. scdp Instruction Format*
### schi and sclo

The `schi` instruction is used to assign the “Eng Max” of a ControlBlock input or output to a declared variable. The `sclo` instruction is used to assign the “Eng Zero” of a ControlBlock input or output to a declared variable.

Figure 1.12.2 shows the format for the `schi` and `sclo` instruction.

#### SYNTAX

The general format is:

```plaintext
a = schi(x)
b = sclo(x)
```

where `a` and `b` are declared variables that receive the “Eng Max” and “Eng Zero” values.

`x` is a ControlBlock register of a continuous input or output value.

#### SCRIPT TYPES

`schi` and `sclo` can be used in the following script types:

```plaintext
PROC   START   UNIT
LIBR   FORM
```

#### SAMPLE INSTRUCTION

```plaintext
upper = schi({TEMP/A})
lower = sclo({TEMP/A})
```

#### DEMONSTRATION OF FUNCTION

shared local upper, lower

```plaintext
upper = schi({TEMP/A})
" If the “Eng Max” value of continuous input A of ControlBlock TEMP is 200.00"
" the value of max is 200.0"
lower = sclo({TEMP/A})
" If the “Eng Zero” value of continuous input A of ControlBlock TEMP is -50.00"
" the value of min is -50.0"
```

---

**Figure 1.12.2. schi and sclo Instruction Formats**
sclike and unsc

The `sclike` and `unsc` instructions are used to provide ControlBlock information in a format readily understandable to the user. ControlBlock continuous values used in RBL scripts are scaled from 0 to 1. However, if the values are to be used in user interfaces such as reports and operator prompts, it is desirable to display the values with scaling that is familiar to the user (`sclike` instruction). If any values are entered by the user for use in the RBL script, the values must be converted to 0 to 1 internal block scaling (`unsc` instruction).

The `sclike` instruction is used to convert a variable value expressed in internal Block scaling (0 to 1) to display scaling, as configured on the Continuous Links screen. The `unsc` instruction is used to convert a variable value in plant scaling to Internal Block scaling (0 to 1).

Figure 1.12.3 shows the format for the `sclike` and `unsc` instructions.
**SYNTAX**

The general format is:

\[ a = \text{sclike}(x,y) \]
\[ b = \text{unsc}(x,y) \]

where \( a \) and \( b \) are declared variables that receive a converted value in either continuous or ControlBlock scaling.

\( x \) is a continuous input or output of a ControlBlock.

\( y \) is a value to be scaled.

**SCRIPT TYPES**

`sclike` and `unsc` can be used in the following script types:

- PROC  START  UNIT
- LIBR  FORM

**SAMPLE INSTRUCTION**

```plaintext
scaledrate = \text{sclike}({SP1/B}, flowrate)

" the value of variable \textit{scaledrate} is now scaled according to SP1 display scaling"

" (for example, 0 to 300 GPM)"

flowrate = \text{unsc}({SP1/B}, scaledrate)

" the value or variable \textit{scaledrate} is now scaled according to SP1 Internal Block Scaling"

" (for example, 0 to 1)"
```

**DEMONSTRATION OF FUNCTION**

A `sclike` instruction is used to scale the flowrate value like the ControlBlock pumpsp in the variable `scaledrate`, which is used in the `print` instruction to tell the user the current pump setpoint.

The `unsc` instruction is used to convert the new operator-entered setpoint to RBL script scaling, which is 0 to 1, so that it can be used by the script.

"pump represents setpoint for block SP1"

"SP1 Internal Block scaling range is 0 to 1"

"Display scaling range is 0 to 300 GPM"

shared alias:pump:SP1/B

shared local flowrate,scaledrate

alias pump;

flowrate=pump; "SP1 now pumping 150 GPM, flowrate equals block value 0.5"

scaledrate=sclike(pump,flowrate); "scaledrate is now 150"

prompt

print(*,"Pump setpoint is %d GPM",scaledrate)

input("Please enter new setpoint ",$p, scaledrate)

flowrate=unsc(pumpsp,scaledrate); "Flowrate is now scaled back to 0 to 1"

pump=flowrate; "Write new setpoint value to ControlBlock"

**Figure 1.12.3. sclike and unsc Instruction Formats**
Section 13:
Timestamp Functions

This section discusses Rosemount Basic Language timestamping functions. Timestamps can be converted to character string values and written to reports or log files, or be converted to arithmetic values and used in arithmetic operations.
The `getssm` function converts a realtime value into seconds since midnight (ssm). Once converted, the value can be used with standard batch numeric operators and functions.

Figure 1.13.1 shows the format for the `getssm` function.

**SYNTAX**
The general format is:

\[
a = \text{getssm}(x)
\]

where

- `a` is a variable or array element that receives a converted realtime value in seconds since midnight.
- `x` is a variable or array storage location that contains a real time value.

**SCRIPT TYPES**
`getssm` can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

**SAMPLE INSTRUCTION**

“Convert realtime value in variable rt to seconds since midnight”
- “and writes value to variable TIME1”
- shared local lssm, rt
  
  \[
  \text{lssm} = \text{getssm}(\text{rt})
  \]

Figure 1.13.1. `getssm` Function Format
The `getsss` function converts a realtime value into seconds since Sunday (sss). Once converted, the value can be used with other numeric operators and functions (see Section 4 in this manual).

Figure 1.13.2 shows the format for the `getsss` function.

### SYNTAX

The general format is:

```plaintext
a=getsss(x)
```

where
- **a** is a variable or array element that receives a converted realtime value in seconds since Sunday.
- **x** is a variable or array storage location that contains a real time value.

### SCRIPT TYPES

`getsss` can be used in the following script types:

- PROC `START` `UNIT`
- LIBR `FORM`

### SAMPLE INSTRUCTION

"Convert realtime value in variable rt to seconds since Sunday and write value to variable lsss."

```
shared local lsss, rt
lsss=getsss(rt)
```

---

**Figure 1.13.2. getsss Function Format**
gettime

The `gettime` function converts a realtime value into arithmetic values for year, month, day, hour, minutes, and seconds. Once it has been converted, the value can be used with standard batch numeric operators and functions.

Figure 1.13.3 shows the format for the `gettime` function.

**SYNTAX**

The general format is:

```
gettime(rt,n1,n2,n3,n4,n5,n6)
```

- `rt` is the variable or array storage location that contains a realtime value.
- `n1`, `n2` are variable or array elements that receive a converted realtime value. Date
- `n3`, `n4`, and time values are assigned in the following order: year, month, day, hour,
- `n5`, `n6` minute, second. Variables or array storage locations are required for all six values.

The value for year is represented as 90, not 1990.

**SCRIPT TYPES**

`gettime` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**DEMONSTRATION OF FUNCTION**

"Write realtime values in variable time1 to variables."

```
shared local time1,yr,mo,day,hr,mn,sec
getime(time1,yr,mo,day,hr,mn,sec)
```

"Write realtime values in array track(i,j) to array data array elements"

"date(1,0) through date(1,5)"

```
getime(track(i,j),date(1,0),date(1,1),date(1,2),date(1,3),date(1,4),date(1,5))
```

---

**Figure 1.13.3. gettime Function Format**
The `gettrend` function returns data points from an SRU trend file and writes these data points in a data array. Data point values are returned in engineering units of measurement. For example, a block value of 50 GPM, with an “Eng Min” of 0 and an “Eng Max” of 50, will be returned as 50 GPM, not 0.50. The data points in the data array correspond by row and column to timestamps in a tracking array.

Figure 1.13.4 shows the format for the `gettrend` function.
**SYNTAX**
The general format is:
```
a=gettrend(b,c,d,e,f)
```
where
- `a` is a variable that indicates the number of data points actually returned by `gettrend`. The value of variable `a` can be checked in the script using logic instructions or relational operators.
- `b` is the ControlBlock link for a trend file from which data points are returned.
- `c` is the PeerWay node number on which the trend file resides.
- `d` is the number of data points returned from the requested trend file (maximum of 255). The tracking and data array must have as many columns as indicated by this value. Otherwise, the number of points returned from the trend file will be trimmed to the column dimensions specified by the array.
- `e` is the two dimensional tracking array in which time stamps are stored. Specify the row and column numbers for the tracking array. Column dimensions are optional.
- `f` is the data array in which trending data points are stored. Specify the row and column numbers for the data array. Column dimensions are optional.

**DEMONSTRATION OF FUNCTION**

"Return data point from trendfile and write time stamp to 'mhtrack' and"
"data point value to 'mydata'."

```
shared local numget,srunode,numpoints,row,cal
shared dim mydata(10,300)
shared vdim mhtrack(10,3000):mytrack1
numpoints=100;srunode=12
numget=gettrend({FIC101/A},srunode,numpoints,mhtrack(2,200),mydata(0,200))
```

**SCRIPT TYPES**

`gettrend` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**Figure 1.13.4. gettrend Function Format**
gettrend Status Values

The `gettrend` function returns numerical status values that can be assigned to a variable and checked in the script to see if the instruction was successful, as shown in Figure 1.13.5.

**Demonstration of Function:**

```plaintext
shared local status
.
.
while (status=gettrend({DVC-3B/C},2,480,track(0),data(0))<0)
    if status==-2
        flog("logger","get data trend failed; check disk!")
    else
        flog("logger","retry get data trend in 30 seconds")
        sleep(30)
    if status<480
      "Too few trend points were returned"
      alarm(i,j,"Too few trend points!")
    hold
```

Figure 1.13.5. Configuring a Tracking Array

Table 1.13.1 describes the status values for the `gettrend` function.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gettrend</code></td>
<td>1</td>
<td>The <code>gettrend</code> instruction successfully returned trend data from the trend file.</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>The <code>gettrend</code> instruction failed to return trend data because of configuration errors in the trend file.</td>
</tr>
<tr>
<td></td>
<td>-2</td>
<td>The <code>gettrend</code> instruction failed to return trend data for reasons other than a configuration error in the trend file.</td>
</tr>
</tbody>
</table>
The `nil` function assigns a blank value to a data array element. Since each row of the data array must correspond to a row of a tracking array, the “nil” value indicates that there is no corresponding data associated with the timestamp in the tracking array. A “nil” value is used instead of a zero value because a zero is a meaningful value in mathematical calculations or in logic expressions. The “nil” value cannot be used with batch numeric functions or operators.

Figure 1.13.6 shows the format for the `nil` function.

**SYNTAX**

The general format is:

```plaintext
a = nil
```

where `x` is the variable or array element that is assigned the nil value.

**SCRIPT TYPES**

`nil` parses in the following script types:

- PROC START UNIT
- EDIT LIBR FORM

**SAMPLE INSTRUCTION**

```plaintext
track(i,j) = nil; "assign nil value to the array storage location track (i,j)."
```

```plaintext
time1 = nil; "assign nil value to the variable time1."
```

Figure 1.13.6. `nil` Function Format
puttime

The puttime function converts time values for year, month, day, hour, minutes, and seconds into a realtime value. Once converted, the realtime value can be used by the material history displays to plot data. The puttime function is the only way you can define timestamps, other than with the current realtime value.

Figure 1.13.7 shows the format for the puttime function.

**SYNTAX**

The general format is:

\[jt=\text{puttime}(rt,n1,n2,n3,n4,n5,n6)\]

where \(jt\) is an optional variable or array storage location to which puttime can assign the realtime value.

**NOTE:** puttime cannot use virtual arrays as arguments for \(jt\).

\(rt\) is the variable or array storage location that receives the realtime value.

\(n1,n2\) are variable or array elements that contain time values to be converted to realtime value. Date and time values are assigned in the following order: year, month, day, hour, minute, second. Variables or array storage locations are required for all six values. The value for year is represented as 90, not 1990.

**SCRIPT TYPES**

puttime can be used in the following script types:

- PROC START UNIT
- LIBR FORM

**SAMPLE INSTRUCTION**

“Convert time values in variables \(n1\) through \(n6\) to a realtime value and write to variable \(time1\)”

shared local rt,jt,yr,mo,day,hr,mn,sec

\[yr=90; mo=2; day=21; hr=22; mn=38; sec=0\]

\[jt=\text{puttime}((time1,yr,mo,day,hr,mn,sec)\]

“After executing puttime, both \(jt\) and \(TIME1\) contain the time stamp Feb 21, 1990, 10:30:00 PM”

“Convert time values in array elements date(1,0) through date(1,5) to a realtime value and”

“and write to tracking array track(i,j)”

puttime(track(i,j),date(1,0),date(1,1),date(1,2),date(1,3),date(1,4),date(1,5))

Figure 1.13.7. puttime Function Format
The `realtime` function returns the current time of day. The realtime value represents the actual time at which a batch event occurs. Time values returned by the `realtime` function cannot be used in arithmetic or logical operations (such as `==`, `<`, `>`, `+`, `-`, `*`, and `/`). This is because realtime values are represented in a notation that is incompatible with floating point notation used by other batch numeric values.

The `realtime` function can write the returned value to a variable or array element. The value can then be converted to a value that can be used in arithmetic operations. Realtime values that are written to a tracking array are called timestamps.

Figure 1.13.8 shows the format for the `realtime` function.

### SYNTAX

The general format is:

```
a = realtime
```

where `a` is the variable or array storage location that is assigned the realtime value.

### SCRIPT TYPES

`realtime` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

### SAMPLE INSTRUCTION

```
track(i,j)=realtime; “assign realtime to the array storage location track (i,j).”
time1=realtime; “assign realtime to the variable time1.”
```

Figure 1.13.8. realtime Function Format
%r, %rt, and %rd print Instruction Extensions

The %r, %rt and %rd print instruction extensions convert a realtime value into character string output. The % designator marks where the converted real time value will appear in the character string. A print instruction writes the character string with the converted value to a Batch Input screen or a batch report. A flog instruction can be used to write the character string to a batch log file.

Figure 1.13.9 shows the format for the %r, %rt, and %rd print instruction extensions.

SYNTAX

The general format is:

```
a$=print$(x,"y",z)
```

- `a$` is an optional string variable to which print can assign the print message.
- `x` is a number (1–10) referring to the active report file, or an asterisk (*) indicating that the print message should be copied to the Batch Input screen.
- `y` is a character string that contains one of the following print instruction extensions:
  - `%r` Converts the complete realtime value for the date and time-of-day into a character string value. For example, "08-Aug-89 08:40:00".
  - `%rt` Converts the realtime value for the time-of-day into a character string value: For example, "08:40:00".
  - `%rd` Converts the realtime value for the date into a character string value: For example, "08-Aug-89".
- `z` is a variable or array element that contains the realtime value.

DEMONSTRATION OF FUNCTION

```
shared local rtime; "Variable that receives the realtime value."
shared string TIMES$  
puttime(rtime,90,8,3,8,40,0); "realtime is assigned to the variable rtime."
print(1,"Date and Time: %r",rtime); "realtime value is converted to the string value."
    " 08-Aug-90 08:40:00' printed to report file 1."
print(1,"Time: %rt",rtime); "realtime value for time-of-day is converted to the"
    "string value. '08:40:00' is printed to report file 1."
TIMES$=print$("Date: %rd",rtime); "realtime value is converted to the string value."
    " '08-Aug-90' and assigned to the string variable TIMES$."
```

Figure 1.13.9. %r, %rt, and %rd print Instruction Extensions
Section 14:
Password Security

This section describes Rosemount Basic Language instructions that use password security to control access to scripts, monitor system users, or determine login status of consoles and system users.
The `alv2str$` instruction returns a text string value for the access level of the user at the console. The `alv2str$` instruction is used with the `inputp` instruction to control script access. Figure 1.14.1 shows the format for the `alv2str$` instruction. See Figure 1.14.2 for a demonstration of the `alv2str$` instruction with the `inputp` instruction.

**SYNTAX**

The general format for the `alv2str$` instructions is:

```
x$=alv2str$(y)
```

where

- `x$` is a string variable that receives a text string value returned by `alv2str`.
- `y` is a number for the user access level. The user permission is entered into the Batch Log. Enter one of the following numbers for the permission indicated:

<table>
<thead>
<tr>
<th>Number</th>
<th>User Permission</th>
<th>Text String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NONE</td>
<td>ANY</td>
</tr>
<tr>
<td>2</td>
<td>Operator</td>
<td>OPR</td>
</tr>
<tr>
<td>3</td>
<td>Supervisor</td>
<td>SUPV</td>
</tr>
<tr>
<td>4</td>
<td>Configuror</td>
<td>CNFG</td>
</tr>
<tr>
<td>5</td>
<td>Recipe Manager</td>
<td>RCPM</td>
</tr>
<tr>
<td>6</td>
<td>System Manager</td>
<td>SYSMGR</td>
</tr>
</tbody>
</table>

**SCRIPT TYPES**

`alv2str` can be used in the following script types:

- PROC
- START
- UNIT
- LIBR
- FORM

*Figure 1.14.1. alv2str$ Instruction Format*
The `inputp` instruction assigns permission to use subsequent code in the script. The Controller Processor (CP) logs the success or failure of the `inputp` instruction and the user permission in the Batch Log. If the `inputp` instruction fails, the CP will refuse to execute any script after the `inputp` instruction.

Figure 1.14.2 shows the format for the `inputp` instruction.

### SYNTAX

The general format for the `x` instructions is:

```
    w=inputp("x",y,z)
```

where

- `w` is a variable used to check the instruction status. Values include:
  - 1 Instruction was successful.
  - -1 Console does not have a password configured.
  - -2 User does not exist.
  - -3 There is a problem with the password file.
  - -4 The password is incorrect.
  - -5 The user does not have the required access permission.
- `x` is a login name. It can be either a character string or a string variable of up to 16 characters. Enclose character strings in quotes.
- `y` is the user permission level required. If the user enters the wrong permission level, the status variable “w” returns a failed status – -5. Authorization is valid if the permission level entered allows the same privileges as the permission level logged in. Enter one of the following numbers for the permission level indicated:

<table>
<thead>
<tr>
<th>Permission Level</th>
<th>Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ANY</td>
<td>Operator, Supervisor, Configuror, Recipe Manager, System Manager</td>
</tr>
<tr>
<td>2 Operator</td>
<td>Operator</td>
</tr>
<tr>
<td>3 Supervisor</td>
<td>Operator, Supervisor</td>
</tr>
<tr>
<td>4 Configuror</td>
<td>Supervisor, Recipe Manager, System Manager</td>
</tr>
<tr>
<td>5 Recipe Manager</td>
<td>Operator, System Manager</td>
</tr>
<tr>
<td>6 System Manager</td>
<td>Operator, Supervisor, Recipe Manager, Configuror, System Manager</td>
</tr>
</tbody>
</table>

- `z` Option is not available. Enter 0 in this parameter.

### SCRIPT TYPES

`inputp` can be used in the following script types:

- PROC START UNIT
- LIBR FORM

Figure 1.14.2. Inputp Instruction Format
DEMONSTRATION OF FUNCTION

local level, status, options
string logon$, name$, msg$, level$
alevel=4
options=0

level$=alv2str$(alevel)
_getperm:
msg$=strcat$("\n Waiting for sign-off by a: %s",level$)
prompt(msg$)
   logon$=input$("\nEnter logon name")
print(*,"Enter password")
   status=inputp(logon$,alevel,options)
if (status~= 1); "value of alevel determines status"
   print(*,"Permission denied")
sleep(3)
goto _getperm

Figure 1.14.2. inputp Instruction Format (continued)
whois

The `whois` instruction returns password information about a user at the selected console node: login name, user name, and permission level. The `whois` instruction includes variables that receive the password information returned. The variables can be used to make entries to reports and log files.

If the selected node is 0, then the `whois` instruction may be embedded in a `prompt` statement, and it will return password information about the user who responds to the prompt. Password information is configured on the User Name Detail screen (UND).

**NOTE:** The `whois` instruction assumes that the person at the console is the person logged in. To determine if the person at the console has the correct permission, use the `inputp` instruction instead of the `whois` instruction.

Figure 1.14.3 shows the format of the `whois` instruction. Figure 1.14.4 demonstrates the use of the `whois` instruction.
SYNTAX
The general format for the whois instructions is:
\[ v = \text{whois}(w, x$, y$, z) \]
where \( v \) is a variable used to check instruction status. Status value are:
- 0 Instruction was successful.
- -111 Node is not a console.
- -112 Node is bad.
\( w \) is the node number of the console. You can type number in this position or enter it as a variable. If you use zero as the console number, then imbed the whois instruction under a prompt or notify instruction to prompt the user for the console number on the Batch Input screen (BAI:).
\( x$\ is a variable that receives the login name of the user at the console. On a key system, it returns “Key System.”
\( y$\ is a variable that receives the user name of the user at the console. On a key system, it returns the key type.
\( z \) is a variable that returns an identification value for the permission levels of the user at the console. Permission levels are expressed as decimal or binary numbers:

<table>
<thead>
<tr>
<th>Permission</th>
<th>Privileges</th>
<th>Decimal Value</th>
<th>Binary Value</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO User</td>
<td>None</td>
<td>0</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>None</td>
<td>None</td>
<td>2</td>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>Operator</td>
<td>None, Operator</td>
<td>6</td>
<td>00000110</td>
<td>4</td>
</tr>
<tr>
<td>Supervisor</td>
<td>None, Operator,</td>
<td>14</td>
<td>00001110</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Supervisor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recipe Manager</td>
<td>None, Operator,</td>
<td>38</td>
<td>00100110</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Recipe Manager</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuror</td>
<td>None, Operator,</td>
<td>62</td>
<td>00111110</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Supervisor, Recipe Manager, Configuror</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Manager</td>
<td>None, Operator,</td>
<td>126</td>
<td>01111110</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>System Manager</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SCRIPT TYPES
whois can be used in the following script types:
- PROC
- START
- UNIT
- LIBR
- FORM

Figure 1.14.3. whois Instruction Format
private local rs, i, id_lvl
private string login$, name$
for i=2,i<97,i=i+2; “Counter incremented by 2 because consoles can only have even numbers.”
    rs=whois(i,login$,name$,id_lvl)
    if(rs==0)
        flog(“LOGIN”,“Node %d %s (%s) level %d”,i,login$,name$,id_lvl)

“whois assumes that the person operating the console is the person logged in.”
“To determine if the person at the console has the correct permission level”
“use inputp instead”

notify
    rs=whois(0,login$,name$,id_lvl); “Use zero to prompt for console number”
    if (rs==0)
        if id_lvl<38; “If access level not Recipe Manager or higher”
            print(“Insufficient Access level to proceed. Need RCPM or higher”)
        goto _NEED_ACCESS; “Operator and Supervisor key to exit”
        print(“%s has correct access level, continuing input”,name$)

_NEED_ACCESS:

Figure 1.14.4. whois Demonstration of Function
Section 15:
RBL Controller Instructions

This section describes RBLC instructions that enable the RBL Controller to communicate with external devices. These instructions can only be used in the “RBLC” script type.

(For information on using the RBLC Controller, see Chapter 2, Section 1 and Section 2 in this manual.)
The $bcc$ instruction performs a common checksum calculation on a character string. The $bcc$ instruction can use either of two checksum methods:

- Summation checksum method. This method calculates the sum of bytes, modulo 256.
- Cyclic Redundancy Check (CRC). This method calculates the remainder of a polynomial division, modulo 2.

Figure 1.15.1 shows the $bcc$ instruction format.
SYNTAX

The general format is:
\[ a$=bcc$(b$,c,d) \]

where
- \( a$ \) is the character string that stores the resulting checksum.
- \( b$ \) is the character string that is checksummed.
- \( c \) is the number of bytes to be checksummed. If \( numbyte \) is set to -1, the entire string is checksummed.
- \( d \) is a number for the checksum method used to calculate the checksum. Legal values for Summation Checksums are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Checksum is performed on 7 data bits.</td>
</tr>
<tr>
<td>8</td>
<td>Checksum is performed on 8 data bits.</td>
</tr>
<tr>
<td>16</td>
<td>Checksum is performed on 16 data bits, with the result being two bytes.</td>
</tr>
<tr>
<td>17</td>
<td>Checksum is performed on 16 data bits, with the result being one byte.</td>
</tr>
</tbody>
</table>

Legal values for CRC checksums are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>Modbus, crc-16</td>
</tr>
<tr>
<td>117</td>
<td>crc-16</td>
</tr>
<tr>
<td>118</td>
<td>ccitt</td>
</tr>
<tr>
<td>119</td>
<td>sdic</td>
</tr>
<tr>
<td>120</td>
<td>xmodem</td>
</tr>
</tbody>
</table>

SAMPLE INSTRUCTION

\[ a$=bcc$(b$,-1,8) \]

DEMONSTRATION OF FUNCTION

Cyclic Redundancy Check (CRC) Method

```plaintext
local hi, lo
string outp$, a$, msg$
outp$="The"
a$=bcc$(outp$, -1, 116)
hi=asc(a$(0,0))
lo=asc(a$(1,1))
msg$=print$("crc= %02x  %02x", hi, lo)
almsg(8,1,msg$
```

The hi and lo alarm message for each CRC protocol

<table>
<thead>
<tr>
<th>Value</th>
<th>hi</th>
<th>lo</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>e3</td>
<td>c7</td>
<td>Modbus, crc-16</td>
</tr>
<tr>
<td>117</td>
<td>23</td>
<td>b6</td>
<td>crc-16</td>
</tr>
<tr>
<td>118</td>
<td>7d</td>
<td>8d</td>
<td>ccitt</td>
</tr>
<tr>
<td>119</td>
<td>44</td>
<td>be</td>
<td>sd;c</td>
</tr>
<tr>
<td>120</td>
<td>1e</td>
<td>0a</td>
<td>xmodem</td>
</tr>
</tbody>
</table>

Figure 1.15.1. bcc$ Instruction Format
chflush

The chflush instruction clears the contents of the receive queue and transmit queue for the specified channel. This routine is normally called after a bad status has been received by one of the read or write routines. For example, if during a read operation, status indicates that a parity error occurred, the program clears out the receive queue to get rid of bad data. Then, if possible, a retry message might be performed. Figure 1.15.2 shows the format for the chflush instruction.

<table>
<thead>
<tr>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>The general format is:</td>
</tr>
<tr>
<td>chflush(a)</td>
</tr>
<tr>
<td>where a is the channel number (0-1) through which communication is established.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAMPLE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>chflush(0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEMONSTRATION OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>if commstat(1)&gt;1; “Test of comm status using ‘commstat’ function can be”</td>
</tr>
<tr>
<td>“used at any time in place of ‘status=function’,”</td>
</tr>
<tr>
<td>“A status of greater than ‘1’ indicates that there is some other error.”</td>
</tr>
<tr>
<td>num_error=num error+1; “increment the error counter by 1”</td>
</tr>
<tr>
<td>chflush(0); “Reset internal communications transmit and receive queue.”</td>
</tr>
<tr>
<td>goto COMMTEST; “Try the test again.”</td>
</tr>
</tbody>
</table>

Figure 1.15.2. chflush Instruction Format
The `chwait` instruction returns a status variable of 1 if all characters in the transmit queue have not been sent out. The status variable is equal to zero if there are no characters remaining in the transmit queue. Figure 1.15.3 shows the format for the `chwait` instruction.

**SYNTAX**
The general format is:
```
chwait(a)
```
where `a` is the channel number (0-1) through which communication is established.

**SAMPLE INSTRUCTION**
`chwait(1)`

**DEMONSTRATION OF FUNCTION**
```
while chwait(1); "While there are still characters being transmitted,"
sleep(1); "sleep. This is used more with slow baud rates."
```

Figure 1.15.3. chwait Instruction Format
The *commclose* instruction closes the communication channel so that another RBL task can use the channel. When a block goes into Idle mode or if the program is finished executing, the channel is automatically closed. When the channel is closed, all data currently in the buffers is cleared out. Figure 1.15.4 shows the format for the *commclose* instruction.

**SYNTAX**
The general format is:

```plaintext
commclose(a)
```

where `a` is the channel number (0-1) through which communication is established.

**SAMPLE INSTRUCTION**

```plaintext
x=commclose(0)
```

**DEMONSTRATION OF FUNCTION**

```plaintext
if num_attempts>100; "After 100 attempts"
x=commclose(1); "Close the channel"
if num_errors==0
    alarm(3,1,"Successful RBL Comm"); "Indicate successful test."
else
    alarmstrg$=print$("RBL Errors=%d",num_errors); "Build error alarm message."
    alarm(3,1,alarmstrg$); "Send alarm message."
```

*Figure 1.15.4. commclose Instruction Format*
The `commopen` instruction opens the communication channel, sets the various communication parameters, and clears out existing data in the communication buffers. The communication parameters include channel, baud rate, parity, data bits, stop bits, and mode.

Figure 1.15.5 shows the format for the `commopen` instruction. Table 1.15.1 lists the `commopen` parameter options. Table 1.15.2 lists the `commopen` instruction status returns.

**SYNTAX**

The general format is:

```
commopen(a,b,c,d,e,f)
```

where
- `a` is the channel number (0-1) through which communication is established.
- `b` is the baud or communication rate of the given channel.
- `c` is the parity of the given channel.
- `d` is the Nbits or data bits.
- `e` is the Sbits or stop bits.
- `f` is the channel mode.

**SAMPLE INSTRUCTION**

```
shared local status
status=commopen(0,11,0,3,7,3)
```

**DEMONSTRATION OF FUNCTION**

```
"Open channel 0 for 9600 baud, even parity, 8 data bits, 1 stop bit"
"and read/write access."
shared local status
.
.
status=commopen(0,11,0,3,7,3); "open the channel"

if status; "Check for non-zero return. If open failed"
    alarm(3,1"CommOpen Failed, RBL"); "generate an alarm."
```

**Figure 1.15.5. commopen Instruction Format**
<table>
<thead>
<tr>
<th>Value</th>
<th>Baud Rate</th>
<th>Parity</th>
<th>Data Bit Values (NBits)</th>
<th>Stop Bit Values (SBits)</th>
<th>Mode Values (Mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.0</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.938</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.875</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.813</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>9600</td>
<td>-</td>
<td>-</td>
<td>1.750</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>7200</td>
<td>-</td>
<td>-</td>
<td>1.688</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>4800</td>
<td>-</td>
<td>-</td>
<td>1.625</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>2400</td>
<td>-</td>
<td>-</td>
<td>1.563</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>1050</td>
<td>-</td>
<td>-</td>
<td>1.0</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1200</td>
<td>-</td>
<td>-</td>
<td>.938</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>600</td>
<td>No Parity</td>
<td>-</td>
<td>.875</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>300</td>
<td>No Parity</td>
<td>-</td>
<td>.813</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>Force Odd</td>
<td>8</td>
<td>.750</td>
<td>Transmit and receive.</td>
</tr>
<tr>
<td>2</td>
<td>134.5</td>
<td>Force Even</td>
<td>7</td>
<td>.688</td>
<td>Transmit only.</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>Odd</td>
<td>6</td>
<td>.625</td>
<td>Receive only.</td>
</tr>
<tr>
<td>0</td>
<td>50</td>
<td>Even</td>
<td>5</td>
<td>.563</td>
<td>Nothing transmitted or received.</td>
</tr>
</tbody>
</table>
### Table 1.15.2. Status Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>This channel is already open.</td>
</tr>
<tr>
<td>17</td>
<td>No EOT character encountered.</td>
</tr>
<tr>
<td>10</td>
<td>Wrong slot is accessing.</td>
</tr>
<tr>
<td>9</td>
<td>This channel is currently not open.</td>
</tr>
<tr>
<td>8</td>
<td>The character that is to be written cannot be found.</td>
</tr>
<tr>
<td>7</td>
<td>The transmit queue for this channel is currently full.</td>
</tr>
<tr>
<td>6</td>
<td>A break character was encountered.</td>
</tr>
<tr>
<td>5</td>
<td>A framing error was encountered.</td>
</tr>
<tr>
<td>4</td>
<td>A parity error was encountered.</td>
</tr>
<tr>
<td>3</td>
<td>An overrun error was encountered.</td>
</tr>
<tr>
<td>2</td>
<td>The receive queue has overflowed.</td>
</tr>
<tr>
<td>1</td>
<td>The receive queue is empty.</td>
</tr>
<tr>
<td>0</td>
<td>Good. No errors were encountered.</td>
</tr>
</tbody>
</table>
commstat

The *commstat* instruction gives the current status of the line. This routine can be called at any time to check the status of the serial line. When the status of the serial line is read, the status is reset to 0. Figure 1.15.6 shows the format for the *commstat* instruction.

**SYNTAX**
The general format is:

```
commstat(a)
```

where *a* is the channel number (0-1) through which communication is established.

**SAMPLE INSTRUCTION**

```
commstat(1)
```

**DEMONSTRATION OF FUNCTION**

shared local status

```
_ERETRY:
status=commstat(1)
if status=0
  alarm(3,1,"Function failed")
goto _ERETRY
```

**Figure 1.15.6. commstat Instruction Format**
The `dectoieee` instruction converts a floating point number which is in DEC format to the IEEE floating point format that is used by the RS3 control system. The conversion starts at the specified character within the `inbuf$` string. Each time a value is converted, 4 bytes are used. For example, if `inbuf$(0,0)` is the starting address, then characters `inbuf$(0,0)` through `inbuf$(3,3)` are used for the first conversion, `inbuf$(4,4)` through `inbuf$(7,7)` are used for the next conversion, etc.

The converted values are stored in an array. The first value is stored at the specified element and the rest of the values are stored from that point on in a contiguous manner. Therefore, the dimension size of the array must be large enough to handle the requested number of conversions. Figure 1.15.7 shows the format for the `dectoieee` instruction.

**SYNTAX**

The general format is:

```
dectoieee(inbuf$(0,0),cnvbuf(0),numflt)
```

where `inbuf$` is the character string that contains the DEC floating point values. The indexes specify the character position in the string. The `dectoieee` instruction only works when `inbuf$` has a middle-endian byte order. For example, the DEC representation of 0.0390625 must be placed into `inbuf$` as decimals 32, 62, 0, 0.

`cnvbuf` is an array where the values that are converted are stored.
`numflt` is the number of values to convert.

**SAMPLE INSTRUCTION**

```
dectoieee(inbuf$(0,0),cnvbuf(0),1)
```

**DEMONSTRATION OF FUNCTION**

```
shared local i,status
shared dim cnvbuf(10)
shared string inbuf$
inbuf$=getnbyte$(a,-1,status)
dectoieee(inbuf$(0,0),cnvbuf(0),1)
i=cnvbuf(0); "i now contains the IEEE format of the DEC value"
```

Figure 1.15.7. `dectoieee` Instruction Format
The `getbyte$` instruction obtains a single character from the input buffer for the specified channel. Figure 1.15.8 shows the format for the `getbyte$` instruction.

**SYNTAX**
The general format is:

```
c$=getbyte$(a,b)
```

where
- `c$` is a string variable, to receive the data.
- `a` is the channel number (0–1) through which communication is established.
- `b` is the local or global variable where the status of the variable is returned.

**SAMPLE INSTRUCTION**
```
PROCEDURE$=getbyte$(1,status)
```

**DEMONSTRATION OF FUNCTION**
```
shared local status
shared string inbuf$

_RETRY:
inbuf$=getbyte$(1,status); "Receive character from the input buffer"
if status
  alarm(3,1,“Function failed”)
goto _RETRY
```

Figure 1.15.8. `getbyte$` Instruction Format
The `lrc8$` instruction allows you to perform a byte longitudinal redundancy check on a character string. This instruction performs an exclusive-OR operation on a specified number of bytes. If the number of bytes to be checksummed is set to -1, the entire string is checksummed. Figure 1.15.9 shows the format for the `lrc8$` instruction.

**SYNTAX**

The general format is:

\[ a$=lrc8$(b$,c,d) \]

where

- \(a\$\) is the character string that stores the resultant checksum.
- \(b\$\) is the character string that is checksummed.
- \(c\) is the number of bytes to be checksummed. If numbyte is set to -1, the entire string is checksummed.
- \(d\) is the number of data bits used when calculating the checksum. Legal values are 7 or 8. If the input string is ASCII, 7 is used. If the input string is binary byte data, 8 is used.

**SAMPLE INSTRUCTION**

\[ a$=lrc8$(b$, -1, 8) \]

**DEMONSTRATION OF FUNCTION**

shared string \(a\$, b\$

shared local \(i\$

\(b\$=\text{"text message"} \)

\(a$=lrc8$(b$, -1, 8) \)

\(i=\text{asc}(a\$) \)

\(i=9310 \)

Figure 1.15.9. `lrc8$` Instruction Format
putbyte

The *putbyte* instruction takes as input the channel number and the single character that is to be written. The character is written to the given channel. Figure 1.15.10 shows the format for the *putbyte* instruction and provides a sample instruction and demonstration.

**SYNTAX**
The general format is:

```
a=putbyte(b,c$)
```

where

- **a** is a local or global variable where the status is returned. Use of status variables is optional.
- **b** is the channel number (0–1) through which communication is established.
- **c$** is the character that is to be written to the serial line. The character can be in a string or just a single character, but only the first character is sent out on the serial line.

**SAMPLE INSTRUCTION**
```
putbyte(1,outbuf$)
```

**DEMONSTRATION OF FUNCTION**
```
shared local status

string outbuf$
outbuf$="a"
status=putbyte(1,outbuf$); "Transmit the first output character, a, on comm line 1"
```

*Figure 1.15.10. putbyte Instruction Format*
**putnbyte**

The `putnbyte` instruction takes as input the channel number and the character or string that is to be written and transmits it on the data line. The difference between the `putnbyte` and `putbyte` instructions is that the `putnbyte` instruction can send out an entire string over the serial line while `putbyte` sends out a single character. Figure 1.15.11 shows the format for the `putnbyte` instruction.

**SYNTAX**

The general format is:

\[ a=\text{putnbyte}(b,c$) \]

where

- `a` is a local or global variable where the status is returned. Use of status variables is optional.
- `b` is the channel number (0–1) through which communication is established.
- `c$` is the character or entire string that is to be written (up to 270 characters).

**SAMPLE INSTRUCTION**

`putnbyte(1,OUTBUFS)`

**DEMONSTRATION OF FUNCTION**

- shared local status
- shared string `outbuf$`

```plaintext
outbuf$="abcdef"; "show string ending with an A"
```

```plaintext
status=putnbyte(1,outbuf$); "Transmit the output character string on comm line 1."
```

Figure 1.15.11. putnbyte Instruction Format
The `seteot` instruction defines the EOT (end of message) character for the specified channel. The EOT character is then used by the `getnbyte$` instruction to indicate an end of message.

**WARNING**

Do not declare the EOT character with `seteot` before a `commopen` instruction. The `commopen` instruction will clear any previously declared EOT character. The EOT is appropriately declared after the communication channel is opened with `commopen`.

The `getnbyte$` instruction reads characters from the communication channel until the EOT character is found, there are no more characters in the queue, or number of bytes is met. When the `getnbyte$` instruction finds the EOT character, it terminates the current read operation and returns. This instruction is more efficient than the `getbyte` instruction.

Figure 1.15.12 shows the format for the `seteot` and `getnbyte$` instructions.
SYNTAX
The general format of a seteot instruction is:

seteot(a,e)

where  
  a  is the channel number (0–1) through which communication is established.
  e  is the numeric decimal ASCII representation for the character used to indicate the end
      of a string.

The general format of a getnbyte instruction is:

c$=getnbyte$(a,b,d)

where  
  c$  is a string variable, to receive the data.
  a  is the channel number (0–3) through which communication is established.
  b  is the number of bytes. The maximum number of bytes is 270. If b = -1 all
      characters are read until either the EOT character is found or the queue is empty.
  d  is local or global variable where the status is returned. Use of status variables is
      optional.

DEMONSTRATION OF FUNCTION

shared local status
shared string inbuf$
.
.
.
  _RETRY
seteot(1,58); “Set the end of message character to a ‘:’.”
Inbuf$=getnbyte$(1,-1,status); “Get the receive string from the comm
  “buffer. The ‘-1’ for the number of bytes indicates to ignore the number of bytes”
  “inbuf$ could now contain ‘character string:’ ending with a ‘:’”
if status
  alarm(3,1,”Function failed”)
  goto RETRY

Figure 1.15.12. seteot and getnbyte Instruction Formats
Chapter 2:
RBL Controller

Section 1: Introduction .................................................. 2-1-1
   RBL Controller Hardware ........................................... 2-1-2
   RBL Controller Software ........................................... 2-1-3
   RBL Files .................................................................. 2-1-5

Section 2: Configuring the RBL ControlBlock ..................... 2-2-1
   RBL Continuous Faceplate Screen ............................... 2-2-2
   Writing Scripts ....................................................... 2-2-4
      RBL Monitor Screen ............................................... 2-2-4
      Using Checksums .................................................. 2-2-6
      Summation Checksum Method .................................. 2-2-6
      Longitudinal Redundancy Checksum Method ............... 2-2-7
   Scaling Conversions ................................................ 2-2-8
   Weigh Scale Sample Program .................................... 2-2-9
   RBL Loop Back Sample Program ................................. 2-2-13
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1</td>
<td>RBL Controller Interface Hardware Functional Overview</td>
<td>2-1-2</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Blocks to Scripts</td>
<td>2-1-3</td>
</tr>
<tr>
<td>2.1.3</td>
<td>RBL ControlBlock Communication with Ports</td>
<td>2-1-4</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Relationship Between Folders, Files, and Scripts</td>
<td>2-1-5</td>
</tr>
<tr>
<td>2.2.1</td>
<td>RBL ControlBlock Continuous Faceplate</td>
<td>2-2-2</td>
</tr>
<tr>
<td>2.2.2</td>
<td>RBL Monitor Screen</td>
<td>2-2-5</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Summation Checksum Method</td>
<td>2-2-6</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Longitudinal Redundancy Checksum Method</td>
<td>2-2-7</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Scaling Conversion Formulas</td>
<td>2-2-8</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Weigh Scale Communications Flow Chart</td>
<td>2-2-9</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Shorting Pins 2 and 3 on Cable Connector</td>
<td>2-2-13</td>
</tr>
<tr>
<td>2.2.8</td>
<td>Loop Back Test Flow Chart</td>
<td>2-2-14</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1 RBL Continuous Faceplate Screen Fields</td>
<td>2-2-3</td>
</tr>
</tbody>
</table>
Section 1: Introduction

This section describes the hardware and software components of the Rosemount Basic Language (RBL) Controller. RBL Controllers are used to execute programs that perform special tasks, such as communicating with external devices, calculating values, and performing nonstandard algorithms. You can configure and run RBL programs called scripts from the RBL ControlBlock Faceplate screen. A script defines the data format and communications protocols for communications with asynchronous field devices.

Interface between the RBL Controller and the field devices is routed through a Communications I/O FlexTerm. One I/O FlexTerm card cage can be shared by two RBL Controller Processor cards. The FlexTerm supports up to two ports for each RBL Controller Processor card. RS422 and RS222 serial link cable lines carry communications between the FlexTerm and field the devices.

RBL ControlBlocks can use standard ControlBlock Links and Faceplate screens. There are two exceptions: RBL cannot use the Continuous Diagram and Discrete Diagram screens.
Figure 2.1.1 shows that the RBL Controller consists of a Controller Processor card and a Communication FlexTerm. (For general information on Controller Processors, see OV: 1.) One FlexTerm can be shared by two RBL Controller cards. Communications between the RBL Controller card and a FlexTerm is carried by a RS422 cable. Communications between the FlexTerm and field devices is carried by either an RS232 or RS422 cable.

A FlexTerm has two ports (0 and 1) for each RBL Controller card. Each port consists of a Port I/O card. This card provides electrical isolation of the system and connection of the field devices to the system. The attributes of each port, such as the baud rate and the communication format, are configured in RBL programs called scripts.

There are eight port slots on the FlexTerm, but only four are used by the RBL Controller software. Slots 1 and 2 in the FlexTerm contain ports 0 and 1 for Controller Processor card 1; slots 5 and 6 contain ports 0 and 1 for Controller Processor card 2.

Two kinds of Port I/O cards are available: RS422/RS422C and RS422/RS232. Both of these cards are connected to the RBL Controller Processor card by an RS422 connection. The RS422/RS232C card provides a single RS232C cable connection to the field devices and the RS422/RS422 card provides a single RS422 cable connection to the field devices. The type of Port I/O card used depends on the kind of connector on the field device.

![Figure 2.1.1. RBL Controller Interface Hardware Functional Overview](image-url)
The RBL Controller uses two software structures: RBL ControlBlocks and RBL Language scripts. The RBL ControlBlock controls execution of the RBL scripts and establishes block links to other ControlBlocks. RBL scripts consist of sets of instructions written in the Rosemount Basic Language. RBL instructions can perform the following functions:

- Define the communications channel between the FlexTerm and field devices, such as I/O port number, baud rate, parity, channel mode.
- Apply mathematical formulas for processing data.
- Parse and decode data sent to and from a field device.
- Define block links with other ControlBlocks.
- Generate alarm and report data.

Up to six RBL ControlBlocks can be running simultaneously, two of which can be in communication with external devices. Communications is limited to two blocks because there are only two serial I/O communications ports to a FlexTerm for each RBL Controller Processor: port 0 and port 1. Each RBL ControlBlock can execute only one RBL script at a time, as shown in Figure 2.1.2.

When the batch script opens a communications channel to a field device, it sets the various communications parameters, and clears out existing data in the communications buffer for the selected port. The buffer holds data being sent to or received from the field device. By holding data until its needed, the buffer helps compensate for differences in data processing speeds of the RBL Controller and the field device.

Communications between blocks is accomplished either by block links or by statements in the script. Values transferred via read or write statements in the script do not count against the 40 link limitation for either the Controller Processor or the Coordinator Processor.

One-to-one relationship between RBL ControlBlocks and RBL scripts. Scripts can define block links between the RBL ControlBlocks.

Figure 2.1.2. Blocks to Scripts
Figure 2.1.3 shows that only one RBL ControlBlock at a time can use an I/O port. If one ControlBlock uses both ports, then only that block can communicate with external devices.

One block uses two ports.

Each block uses one port.

**NOTE:** The RBL ControlBlock does not function unless the controller contains the proper program image. The Nonvolatile Memory card stores the RBL program image and loads it to the processor on power-up.
Scripts reside in files in the RBL Files Folder on the console hard disk. Each file in the folder can contain multiple scripts, as shown in Figure 2.1.4.

Figure 2.1.4. Relationship Between Folders, Files, and Scripts
Section 2: Configuring the RBL ControlBlock

The RBL ControlBlock is software that is configured only on the RBL controller. The main function of the RBL Block is to allow you to define links to access different blocks throughout the system and to associate an RBL script with each block. The RBL controller can access any number of blocks with the only limitation being the time required to access large numbers of links. Each block has a user-enterable field that identifies the RBL program associated with that block.
RBL Continuous Faceplate Screen

The RBL Continuous Faceplate screen is used to configure an RBL ControlBlock and perform RBL operations.

To call up the RBL Continuous Faceplate screen:

- Type the following command at the command line in the “Function” field of the Block Continuous Faceplate screen:

  **RBL (block address) [ENTER]**

Figure 2.2.1 shows the RBL Continuous Faceplate screen. Table 2.2.1 describes the fields on the screen that are unique to the RBL controller.

```
<table>
<thead>
<tr>
<th>Tag =&gt;</th>
<th>Desc =&gt;</th>
<th>OUT =&gt; A =&gt; B =&gt; C =&gt; D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>Function =&gt; RBL</td>
<td><strong>RBL Rosemount Basic Lang Controller</strong></td>
</tr>
<tr>
<td></td>
<td>INPUTS A B C D</td>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td></td>
<td>Value &gt; &gt; &gt; &gt;</td>
<td><strong>.00</strong></td>
</tr>
<tr>
<td>Units</td>
<td>Tag</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag =&gt;</th>
<th>Desc =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Volume Name =&gt; COURT 1</strong> Backup Volume Name =&gt; -NONE-</td>
</tr>
<tr>
<td></td>
<td><strong>File Name =&gt; TCI-1</strong> GOTO RBLC Monitor</td>
</tr>
<tr>
<td></td>
<td><strong>Script Name =&gt; TCI-1</strong> GOTO RBLC Script</td>
</tr>
<tr>
<td></td>
<td><strong>Load Program From Disk</strong> Kill Program Execution</td>
</tr>
<tr>
<td></td>
<td><strong>Restart Program From Bubble</strong> Auto Lock =&gt; No</td>
</tr>
<tr>
<td></td>
<td><strong>Program Status RBLC Running: try+91</strong> Plant Unit &gt;0</td>
</tr>
<tr>
<td></td>
<td><strong>Block Mode&gt; AUTO</strong></td>
</tr>
</tbody>
</table>
```

**Figure 2.2.1. RBL ControlBlock Continuous Faceplate**
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume Name</td>
<td>Name of the drive volume that contains the RBL program. Status must be in the “idle” mode.</td>
</tr>
<tr>
<td>File Name</td>
<td>Name of the file that contains the RBL script. Press [SELECT] to call up the RBL Monitor screen to display the files.</td>
</tr>
<tr>
<td>Script Name</td>
<td>Name of the script that contains the RBL program. Press [SELECT] to call up the RBL Monitor screen to display the script names.</td>
</tr>
<tr>
<td>Load Program</td>
<td>Press [ENTER] on this field to load the program into this RBL controller and begin execution of the program.</td>
</tr>
<tr>
<td>Restart Program</td>
<td>Press [ENTER] on this field to begin executing the RBL program currently in memory. If no program is in memory, the program specified on this screen is loaded from the disk and begins executing.</td>
</tr>
</tbody>
</table>
| Program Status      | Displays information about the current status of program. This field displays four status conditions:  
  — RBLC Download: Downloading from disk.  
  — Idle: Not executing.  
  — Retry Download: Download failed, trying again.  
  — RBLC Running: (Program line) |
| Backup Volume Name  | Displays the name of a backup drive volume that contains the RBL file and script. |
| GOTO RBLC Monitor   | Press [ENTER] or [SELECT] on this field to call up the RBLC MONITOR screen for this RBL block. |
| GOTO RBLC Script    | Press [ENTER] or [SELECT] on this field to call up the RBLC SCRIPT screen for the program specified on this screen. |
| Kill Program Execution | Press [ENTER] on this field to stop executing a program. |
| Auto Lock           | Specifies whether or not the block mode is locked in Auto mode. “Auto Lock=yes” specifies that the block mode is locked in AUTO mode and cannot be changed. “Auto Lock=no” specifies that the block mode is not locked in Auto mode and can be changed. |
| Plant Unit          | Displays the plant unit number used for console ownership. |
| Block Mode          | Is not used (unlike other ControlBlocks). Like other blocks, the RBL ControlBlock must be in “manual” mode to delete the function block. |
Writing Scripts

RBL scripts are used to write an RBL program. Scripts are the programs that communicate with ControlBlock software, which in turn communicates with the field devices. The RBL program is stored on a disk at the console and loaded into the RBL controller when the program is to be executed. The RBL program is also backed up in nonvolatile memory. The script type must be RBLC to use special RBL features, such as writing to the block outputs and performing serial communication. (Script types are specified in the “type” field on the RBL Script screen.)

For more information about configuring scripts, see BA: 1-2. For more information about RBLC instructions, see BA: 1-14.

RBL Monitor Screen

The RBL Monitor screen is used to monitor the execution of an RBL program. The RBL Monitor screen displays program lines of the script as they are being executed, with the currently executing line backlit. The RBL Monitor screen also allows you to change the script execution mode. The RBL program can execute in Step or Run mode.

For debugging purposes, you can change program execution to step mode. While the program is executing in the step mode, the monitor screen scrolls to the current program line being executed. This line is displayed in a different color.

To call up the RBL MONITOR screen:

- Type the following command at the command line:
  
  RBLM  [ENTER]

Figure 2.2.2 shows the RBL Monitor screen.
To change runtime modes:

- Cursor to one of the following operations and press [ENTER]:
  
  - halt — stops script execution (temporary).
  
  - cont — resumes execution after a halt or brkpt command.
  
  - step — executes one line of the script after a halt command.
  
  - fast — increases priority execution speed of script.
  
  - slow — decreases priority execution speed of script.

To execute script control commands:

- Enter one of the following commands in the “Cmd” field:
  
  - none — No command is in use.
  
  - stop — Go to an on stop trap in the script.
  
  - abort — Go to an on abort trap in the script.
  
  - brkpt — Hold execution of the script at a specified line. Press [SELECT] on the line and then enter “brkpt” in the “Cmd” field.
  
  - goto — Execution of script is directed to line number selected with the > arrow at the left of the screen.
  
  - clrbrkpt — Clear the breakpoint.
  
  - resume — Resume execution of a script suspended by an on hold instruction.

Figure 2.2.2. RBL Monitor Screen
Using Checksums

A checksum is used to verify that information is transferred correctly between the RBL Controller and the field device. A checksum is attached to request or reply messages of up to 270 characters in length. The checksum number will almost certainly change value if information in the message is changed during transmission.

The RBL software supports two different checksum methods:
- Summation checksum method
- Longitudinal redundancy checksum method

Summation Checksum Method

The summation method adds up the bytes for all characters of data and takes the result modulo (%) 256. This method gives 256 possible checksums.

The RBL instruction $bcc$ calculates checksums by the summation method.

Figure 2.2.3 demonstrates the use of the summation checksum method.

![Diagram of summation checksum method](image-url)

Figure 2.2.3. Summation Checksum Method
The longitudinal redundancy check method performs an Exclusive-OR operation (XOR) across all the bytes for all characters of data in the message. The Exclusive-OR operation works as follows: for corresponding bits of the two bytes, if either bit is a 1—but not both—the corresponding bit of the result is a 1; otherwise it is a 0. The Exclusive-OR operation is performed on the corresponding bits of the the first two bytes, and then on the resulting byte and the third byte, and so on until the last byte. The final result is the checksum.

The RBL instruction \texttt{lrc8$} calculates checksums by the longitudinal redundancy checksum method.

Figure 2.2.4 demonstrates the use of the longitudinal redundancy checksum method.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{longitudinal_redundancyChecksum.png}
\caption{Longitudinal Redundancy Checksum Method}
\end{figure}
Scaling Conversions

In RBL scripts, all conversions from decimal scaling to display scaling in engineering units of measure (or vice versa) must be performed using formulas in the script. Figure 2.2.5 shows two formulas for performing scaling conversions.

Frequently, analog values need to be converted from decimal scaling used by the ControlBlocks or devices in the field to display scaling in engineering units of measure (or vice versa). A device processor or ControlBlock typically measures process values on a scale of 0 to 1. However, if this information is to be meaningful to a user, it needs to be converted to an equivalent engineering units of measure that are readily understandable. For example, a decimal value of .5 on a display scale of 100 to 500 lbs is equivalent to 300 lbs.

NOTE: When performing scaling conversions, make sure that the scaling range in the formula accurately represents the range used by the ControlBlock or device.

EQUATION FOR CONVERTING DECIMAL VALUES TO VALUES IN ENGINEERING UNITS OF MEASURE

\[ EV = DV \times (\text{ABS}(\text{ENG MAX}-\text{ENG MIN})) + \text{ENG MIN} \]

where

- \( EV \) = Engineering value in Engineering Units of Measure.
- \( DV \) = Decimal value to be converted to Engineering Units of Measure.
- \( \text{ENG MAX} \) = Maximum scale value for Engineering Units of Measure. For example, on a scale of 100 to 500 lbs, 500 is the ENG MAX.
- \( \text{ENG MIN} \) = Minimum scale value for Engineering Units of Measure. For example, on a scale of 100 to 500 lbs, 100 is the ENG MIN.

EQUATION FOR CONVERTING VALUES IN ENGINEERING UNITS OF MEASURE TO DECIMAL VALUES

\[ DV = (EV - \text{ENG MIN}) / (\text{ABS}(\text{ENG MAX} - \text{ENG MIN})) \]

where

- \( DV \) = Decimal value to be converted to Engineering Units of Measure.
- \( EV \) = Engineering value in Engineering Units of Measure.
- \( \text{ENG MAX} \) = Maximum scale value for Engineering Units of Measure. For example, on a scale of 100 to 500 lbs, 500 is the ENG MAX.
- \( \text{ENG MIN} \) = Minimum scale value for Engineering Units of Measure. For example, on a scale of 100 to 500 lbs, 100 is the ENG MIN.

NOTE:
- \( \text{ABS} \) in the RBL Language is a reserved word for absolute value function.
- The formula variables above are not reserved words in the RBL Language. Local variables or constants can be used to represent formula values.

Figure 2.2.5. Scaling Conversion Formulas
Weigh Scale Sample Program

The following sample program demonstrates a hypothetical use of the RBL Controller for communicating with a weigh scale. The RBL Controller requests values for gross, tare, and net weight from the weigh scale, and assigns them to ControlBlock links for use in control operations, graphics, and reports.

Figure 2.2.6 show a flow chart representation of the steps performed by the RBLC controller.

NOTE: It is not necessary to enter the program comments in order to run the script.

**Figure 2.2.6. Weigh Scale Communications Flow Chart**
"===================== Weigh Scale ====================="
"Declare local and string variables."
shared local status; "Channel status"
shared local begtime, endtime; "Calculate time from open"
  "channel to get message"
shared local tare; "Empty with scale value from device"
shared local gross; "Load weigh scale value from device"
shared local net; "Net weigh scale value from the device"
shared local length; "Length of reply in characters"
shared local incr; "Input for simulated weigh scale values"
shared string request$; "Request message string"
shared string reply$, tempbuf$; "Reply message strings"
shared string checksum$; "Reply or request checksum"
shared string tare$; "Preconversion tare weight string value"
shared string gross$; "Preconversion weight string value"
shared string net$; "Preconversion net string value"
tempbuf$=" ", reply$=" "; "Set initial values for reply strings"
priority(300); "Set script execution speed"
begtime=sss; "Set begin time"

"Open the communication channel, where 1 is the channel number,"
TRANSMIT:; "Phase opens channel, builds message, and"
  "transmits it"
"11 is a baud rate of 9600, 0 is even parity, 3 is the code for"
  "8 data bits,"
"7 is 1 stop bit, and 3 is both transmit and receive mode."
incr=1
status=commopen(1,11,0,3,7,3)
if status; "If the communication channel fails to open,"
  "generate an alarm"
    alarm(3,1,"communication failure")
alarm(3,1,"communication channel open")
chflush(1); "clear contents of communications buffer"

"Build the request message. Three messages are sent:"
"Message 10 requests the weigh scale value, message 11 requests"
  "the gross weigh scale value"
"Message 12 requests the weigh scale value"
request$="10,11,12"

"Calculate the request message checksum"
"using summation checksum method"
"Use the 2's complement of the 7 or 8 bit character string."
checksum$=bcc$(request$, checksum$)

"Append the checksum to the request message so that the new"
"request string"
"consists of the old message with the checksum added on the"
  "end."
request$=strcat$(request$, checksum$)

"Write the request to the device. The putnbyte takes the"
"channel number"
"and the request$ character string and transmits it on the data"
"line"
status=putnbyte(1, request$)
sleep(1)

RECEIVE:; "Phase receives and parses message"
"Define EOT (end of text) character for reply message"
"58 is the ASCII representation of EOT character ‘:’ "
seteot(1,58)
"Read the reply message from the channel until the"
"end-of-message character is found."
"1 is the channel, -1 indicates read all characters until"
"end-of-message"
status=1000; "initialize status to nonstandard value"
while 1; "loop continuously"
reply$=getnbyte$(1,-1,status); "Get message in buffer"
if (status==1); "if buffer is empty, wait for message 10"
"minutes from time channel opened"
endtime=sss
if (endtime-begtime)>600; "If over 10 minutes since"
"channel opened"
alarm(3,1,"DEVICE TIMEOUT ALARM"
"If message is good (status 0), recalculate checksum and"
"continue."
"If EOT is not reached, wait .5 second and read buffer again"
"to get the rest of the message"
"If the second read is unsuccessful, flush buffer, generate"
"alarm, and wait for next message"
if (status>0); "If complete message not received"
if (status==17); "If only partial message received and no"
"EOT encountered"
sleep(.5); "Wait .5 second and read message again"
tempbuf$=getnbyte$(1,-1,status); "Get rest of message"
"in buffer"
if (status>0); if complete message still not received
gosub INERROR; "Error subroutine"
continue; "Start while loop over again (leave blank"
"line after continue)"
else; "Second read got the rest of the string; EOT"
"encountered"
reply$=strcat$(reply$,tempbuf$); "Concatenate first"
"and second read"
else; "Read error other than ‘No EOT’ encountered"
gosub INERROR; "Error subroutine"
continue; "Start while loop over again (leave blank line"
"after continue)"
"Message now in reply$"
"To terminate while loop, enter ‘break’ command "
length=strlen(reply$); "Count number of character bytes in"
"reply message"

"Recalculate the reply$ message checksum and store it in the"
"variable checksum$"
"length -1 is the number of bytes to be checksummed"
"8 is the number of data bits used when calculating the"
"checksum."
checksum$=bcc$(reply$,length-1,8); "recalculate checksum"
if strcmp(checksum$,reply$(length-1,length-1)); "Compare"
"checksums"
alarm(1,1,"checksum failed"); "if checksums not equivalent,"
"generate alarm"

"Parse the reply message and assign data to variable names."
tare$=reply$(1,2); "assign tar$ first two characters 1,2"
gross$=reply$(4,5); "assign gross$ next two character 4,5"
"(character 3 is a comma)"
net$=reply$(7,8);  "assign net$ final two characters 7,8"
    "(character 6 is a comma)"

"Convert the string bytes to RS3 floating point format."
"Floating variables do not have $ in variable name"
val(tare$,tare)
val(gross$,gross)
val(net$,net)

"Write converted values to a ControlBlock"
{Scale/A}=tare
{Scale/B}=gross
{Scale/C}=net

INERROR:
"Error occurred while reading message"
"Generate alarm and clear contents of buffer for channel 1"
alarm(3,1,"READ FAILURE OR BAD DATA")
chflush(1)
return
RBL Loop Back Sample Program

The following RBL sample program performs a loop back test to verify that the RBL Controller transmit and receive hardware is functional for any of the four communication channels on the RBLC controller and communication interface card in the FlexTerm.

**NOTE:** It is not necessary to enter the program comments in order to run the script.

1. **To perform this test:**
   1. Configure one RBLC block and fill in the correct volume, file and script information.
   2. Short pins 2 and 3 together on your RS-232 connector to create a closed loop circuit. Pin 2 is used to transmit data; pin 3 is used to receive data. Short the pins at the end of the RS-232 cable, so that the loopback includes the cable.
   3. Execute the loop back program. The RBL Monitor screen reports the number of loopback attempts in the variable `num_attempts` and the number of errors in `num_errors` variable. Zero errors are displayed if messages sent from the controller are being received correctly back at the controller, which indicates that the hardware is operational.

Figure 2.2.7 shows the positions of pins 2 and 3 on a cable connector. Figure 2.2.8 shows a flow chart representation of the loop back program.

![Shorting Pins 2 and 3 on Cable Connector](image)
Figure 2.2.8. Loop Back Test Flow Chart
Program:
"This routine performs a local loopback test on the RBL controller."

"The channel number can be changed by modifying the variable ‘channel’"
"in this program to a value in the range of 0 - 3. This value MUST"
"match the location of your communications board in the FLEXCAGE."
"The number of attempts are reported on the bottom of the RBL Monitor"
"screen in the variable num_attempts and the number of errors will"
"be reported in num_errors. The number of errors represent the"
"number of times a string was transmitted and not received correctly"

Declare local variables.

shared local status,num_attempts,num_errors, channel"
shared string outbuf$, inbuf$, alrmstrg$; "Declare string variables."

"Open channel port"
0 is channel number, 11 is a baud rate of 9600, 0 is even parity,"
3 is the code for 8 data bits, 7 is 1 stop bit,"
and 3 is both transmit and receive mode."
channel=0; "channel number"
status=commopen(channel,11,0,3,7,3); "open the channel"
if status==1; "Check open status. Generate alarm if open failed"
     alarm(3,1,"CommOpen Failed, RBL")
"set end of test message character to a ':'."
seteot(channel,58)
"Build the test message string"
outbuf$="This is a Test of the RBL Controller Communications:" 
COMMTEST: "Declare a phasename."
"Check test error count"
if num_errors>20; "If excessive errors are encountered," 
     alarm(3,1,"Excess Comm Errors"); "generate an alarm and,"
     stop; "end the test."
"Count test attempt"
num_attempts=num_attempts+1; "Increment the number of attempts by 1"
"Clear input buffer and write test message string to communications device"
inbuf$=""; "Clear the input buffer string"
status=putnbyte(channel,outbuf$); "Transmit the output character string."
while chwait (channel); "While there are still characters being transmitted," 
sleep(.1); "sleep. This is used more with slow baud rates."

"Check write status of test message string"
if (status); "If the ‘putnbyte’ fails,"
     alarm(3,1,"Putnbyte Function Failed"); "generate an alarm."
     chflush(channel);
start COMMTEST; "Retry the test again."
"Read reply message string from communication device into buffer"
"buffer. The ‘-1’ indicates to ignore the number of" bytes.
inbuf$=getnbyte$(channel,-1,status);
"check the read status of the reply message string"
if status==1; "If status value equal to ‘1’"
     "The input buffer is empty and the loopback jumper may not be in place."
     alarm(3,1,"No Loopback Jumper"); "Generate an alarm."
     hold; "Hold here until the jumper is in place, then enter ‘cont’."
"Check the serial line status"
"commstat can be used at any time in place of ‘status=function’."
if commstat(channel)>1; "If status of greater than '1'
   "There is some other error in the channel number."
   num_errors=num_errors+1; "increment the error counter by 1"
   chflush(channel1); "Reset internal communications transmit"
      "and receive queue."
   start COMMTEST; "Try the test again."
   "Compare the transmit and receive data."
   if strcmp(inbuf$,outbuf$); "If they don’t match"
      num_errors=num_errors+1; "increment the error count by one."
      chflush(channel1); "Reset internal communication receive queue."
   "Check the number of test attempts"
   if num_attempts>100; "After 100 attempts"
      commclose(channel); "Close the channel"
      if num_errors==0; "If zero errors, generate alarm"
         alarm(3,1,"Successful RBL Comm"); "to indicate successful test."
      else; "test errors encountered, generate alarms"
         alrmstrg$=print$("RBL Errors =%d",num_errors); "Build error message."
         alarm(3,1,alrmstrg$); "Send alarm message."
      stop; "This is the end of the test."
   start COMMTEST; "Start the test over again"
Appendixes

Appendix A: Exception Handling Example ............................... A-1
   About the Example .................................................. A-2
   Restarting on Power Up ......................................... A-2
   Responding to Hold, Stop, and Abort ............................ A-4
   Responding to Print Failure ...................................... A-6
   Exception Handling for an Operation ............................ A-7
   ControlBlock Output ............................................ A-15

Appendix B: Timing Backups to Nonvolatile Memory ............. B-1
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>On power_up Trap</td>
<td>A-3</td>
</tr>
<tr>
<td>A.2</td>
<td>on stop, on abort, and on hold Traps in Local Environment</td>
<td>A-5</td>
</tr>
<tr>
<td>A.3</td>
<td>ControlBlock Output, Step a</td>
<td>A-15</td>
</tr>
<tr>
<td>A.4</td>
<td>ControlBlock Output, Step b</td>
<td>A-15</td>
</tr>
</tbody>
</table>
Appendix A: Exception Handling Example

Exception handling refers to RBL or ControlBlock logic routines that prevent loss or lockup of control of the batch process. System on trap instructions are frequently used to handle exceptions to normal batch processing. Exceptions can include a wide range of operator and system errors, some of which can adversely affect critical batch processes. Batch scripts must include exception handling routines that define appropriate system responses to problems that can be foreseen.

This appendix demonstrates exception handling for several potentially disruptive types of exceptions:

- Stop, Abort, or Hold commands are issued from a BFACE process graphic during script execution.
- An operation restarts after a Coordinator Processor (CP) crashes or switches to a redundant card.
- A report file fails to open, causing the script to suspend execution at the print instruction.
About the Example

The example demonstrates exception handling in batch scripts for a single recipe operation. Since the script examples are complicated, flow charts are included to show the order in which the RBL scripts execute exception handling routines.

Restarting on Power Up

The on power_up trap restarts the operation after a CP crashes or switches to a redundant card. In the example, the user has three choices after the CP software reboots:

- Restart the operation at the last phase where the operation terminated.
- End the operation.
- Begin the operation over again. (The user makes this choice passively by not choosing either of the other two choices.)

The “Restart” routine enables the script to determine which phase was executing last before the CP crash or card switch. During normal execution, each critical phase writes a unique identification number to a designated ControlBlock. If the CP software reboots, the “Restart” routine reads that number stored in the ControlBlock to determine at which phase to restart execution. The “Restart” routine starts the phase identified by the number.

WARNING

You should not rely only on backups to CP nonvolatile memory to restart the last operation. The “Restart” routine is more reliable than backups to nonvolatile memory because the phase identification numbers are stored in the ControlBlock, not in the nonvolatile memory. The ControlBlock is always current. Nonvolatile memory, however, is normally backed up every 10 to 20 seconds. Increased system demand on the CP can increase the length of time between backups. Backup intervals of over one minute are possible.

Figure A.1 shows the order in which the “Restart” routine is executed. Device.C is the tag and register of the ControlBlock in which the phase identification numbers are stored.
Figure A.1. On power_up Trap
Responding to Hold, Stop, and Abort

On the BFACE graphic object, the Stop and Abort commands will terminate or interrupt the batch task. The on stop, on abort, and on hold traps are used to shut down or suspend critical batch activities. These on traps are included in the phase local environments if the exceptions handled are specific to the phase, or in the global script environment if the exceptions handled apply to more than one phase. For example, only in the “Run_it” phase is the timer turned off before the hold subroutine is executed.

NOTE: Use the Stop command when there is no immediate urgency to shut down the system, which is typically in the case where a prescribed sequence of activities are performed in shutting down the batch process. Use the Abort command to perform an immediate or emergency shutdown.

Figure A.2 shows the order in which stop and abort routines are executed for local on stop and on abort traps in the example. The on stop trap initiates various shutdown procedures in the “Stop” phases. The script then executes the abort code in the “Abort” phases to shut down the batch process. The on abort trap bypasses the “Stop” phases to shut down the batch process immediately.

The “Action” subroutine is used to write the time at which the batch task is shut down to the report and log files.
Figure A.2. on stop, on abort, and on hold Traps in Local Environment
Responding to Print Failure

If a *print* instruction attempts to write to a report file that is closed, *on file_n_err* traps in both the library script and the start script will initiate exception handling procedures.

- In the start script, the *on file_n_err* trap executes the *open* instruction repeatedly in a *while* loop until the report file is successfully opened.

- In the library script for the operation, the *on file_n_err* trap directs execution of the script until the report file is opened or reports are disabled. A *filerr* instruction in a *while* loop tests whether the *on file_n_err* trap in the start script is executing. When the *on file_n_err* trap in the start script completes execution, the *filerr* instruction terminates the library script while loop, and the library script reexecutes the *print* instruction that failed.

If the report file cannot be opened, the operator can choose to disable the report. If the report is disabled, a value of 0 is assigned to the shared local variable "r", which will terminate the *while* loop in the library script *on file_n_err* trap and prevent execution of any *print* instructions controlled by the "if r" condition.
Exception Handling for an Operation

The following scripts demonstrates exception handling in the Rosemount Basic Language for an ABC Batch operation:

RBL Script WINI, Operation
Type LIBRARY

"Exception Handling for an Operation"
goto _Begin; "Skip over symbols, use label to maintain global environment"
"The scrip demonstrates the following;"
" Handling requests for HOLD, STOP, and ABORT from a BFACE graphic object."
" Use of a watch dog timer and display variables for BFACE graphic object."
" Handling a restart on power up. The restart routine instead of the last"
" nonvolatile memory backup determines where to resume the script."
" Handling failure to print to a report file."
" ******************************************************** WARNING ******************************************************** "
"Do not rely entirely on batch scripts to put the process in a safe state."
"Micro processors are not completely reliable. Batch scripts cannot be"
"guaranteed to execute in a timely and consistent manner. The most reliable"
"safety shut down system is one that is completely separate from RS3. The next"
"secure level of protection is to interlock the ControlBlocks with logic in the"
"controller."
" ******************************************************** WARNING ******************************************************** "
"The device being operated simulates a material loader. Input B is a timer"
"setpoint,input A is the timer, and @a is the toggle that turns on the timer"
"(If turned off, the timer holds the time). Step a is true when the timer"
"reaches the setpoint."
"In step b, the timer is reset to zero (A=0) whenever @b is true."
"Input C stores the phase restart number used by the restart routine"
"in the batch script. Input @c is the restart enable switch. If the original"
"node, is no longer available, enable input @c to manually restart the Working"
"Recipe in another node."
"The script continues operation of the device in that node."
shared local aborted; "If set by task, all future operations end"
shared local abort_all; "Set by startup script on abort trap"
shared local stop_all; "Set by startup script on stop trap"
shared local r; "If disk cannot be fixed, clears report enabler"
private alias Device; "Device that this operation operates"
private local setpoint; "Setpoint parameter passed to the timer"
private local holder; "Stores the value of opremain during hold"
private local watchdog; "Displays the value of phremain, stores it during hold"
private local answer; "Stores user response to input restart query"
shared string logname$; "Name of the log file"
private string opname$; "Name of the material step using the operation"
private string phase$; "Phase name used by displays and logs"
private string device$; "Tag of the device, so operator can find it"
private string action$; "Name of the action, either print or log"

_Begin:
if aborted; end; "End terminates this operation. Do not do anything"
" if an earlier operation was aborted"
"Set the default on hold, on stop, and on abort traps in the global"
"environment."
"These are used unless traps are defined in the phase environments."

on hold
  gosub Hold_it; "Use a common subroutine to handle timing, logging, and holding"
  return; "After a Hold, the script always resumes what it was doing"

on stop
  start Stop_0

on abort
  start Abort_0

on file_1_err; "print failure, start script will open report file"
  while filerr&r; "Wait until report file is opened or disabled"
  return

on power_up
  if Device.@c; start Restart; "If input @c normal restart is enabled, manually"
    "restart the recipe"
  prompt; "Should not be here. Ask for help!"
  print(*,"Restart disabled for Operation %s, device %s",opname$,device$)
  print(*,"nCondition is not normal because restart is required.")
  gosub _Option
  goto _Begin

if Device.@c; start Restart; "Restart should not normally be enabled here."
  "Ask for help!"
  prompt; "Should not be here, ask for help"
  print(*,"Restart enabled for Operation %s, device %s",opname$,device$)
  print(*,"nCondition is not normal because the operation is just beginning.")
  gosub _Option

start Initialize; "Skip over exception handling routines and start the operation"

  _Option:
  print(*,"nThe operation will restart at phase %d",Device.C)
  print(*,"nYou have the following options: n\t1 Start the Restart phase")
  print(*,"n\t2 End this operation and go to the the next operation")
  print(*,"n\nEnter any other number will ignore the restart and begin the"
    "operation over again")
  input("^Choice?",answer)
  if answer==1; start Restart
  if answer==2; end
  return

"The next routine restarts the operation after a power failure or a CP switch"
"It reads a phase restart number stored in a ControlBlock input register to"
"determine which phase to start. The number is more current than the last"
"nonvolatile memory backup. If power fails for both the CP and the Controller"
"Processor, the operation will resume where at the phase indicated by the last"
"backup of the Controller Processor, to match the Controllers backup state."

  Restart:
  action$="Restarted"; gosub Action; "Report restart time"
  if Device.C==1; start Setup
  elseif Device.C==2; start Run_it
  elseif Device.C==3; start Teardown
  elseif Device.C==4; start Report
  elseif Device.C==5; start Stop_1
  elseif Device.C==6; start Stop_2
  elseif Device.C==7; start Abort_2
  else; start Initialize
  "The use of phase names (for the routines listed above) instead of labels"
  "creates local environments in the script separate from the script global"
  "environment. If these subroutines were in the global environment, then Stop"
  "and Abort commands would direct execution to on traps in the global"
  "environment, not to traps in the phase local environment."

  Action: "This routine is used by exception routines to log or print exceptions"
if r; print(1,"\n\n%TIME %s $s in %s",action$,opname$,phase$)
else; flog(logname$,"%s $s in %s",action$,opname$,phase$)
return

Hold_it: "The common hold subroutine"
holder=opremain; "Note how much time is left in the operation"
phremain=999D; "If phremain is active, the watchdog variable saves the value of"
"phremain"
display("HOLD",opremain); "Display start time of HOLD state"
action$="Hold"; gosub Action

while holding; "Wait for Resume command"

  opremain=holder; "Freeze the value of opremain, though it may bounce"
display("RUN",opremain); "Display time change from HOLD to RUN state"
if r; print(1,"\n\n%TIME Resume %s, held %t",phase$,999D-phremain); "Log hold"

phremain=watchdog; opremain=holder; "Restore the timer values"
return; "Go back to work"

"The following Stop and Abort routines do not return to the last script line"
"They end the operation after setting a global aborted flag. Stop proceeds"
"abort because stop executes a procedure to get equipment into a state"
"that can be safely terminated by Abort."
Stop_0: "Default stop"
action$="Stop"; gosub Action
start Leave; "There is no stop procedure in the default Stop phase"

Abort_0: "Default abort"
action$="Abort"; gosub Action
start Leave; "Go to common exit"
Stop_1: "Stop from Running state"
Device.C=5
display("STOP ",opremain)
action$="Stop"; gosub Action
if Device.@a; "If the device is enabled, and not holding"
  Device.B=10; "Change the setpoint"
  until Device.a; "Wait for it to become safe to stop"

Abort_1: "Abort phase is identical to the tear down phase."
Device.C=7
start Abort_2
Stop_2: "Stop issued from the Teardown phase"
Device.C=6; "Phase restart identification number"
display("STOP",opremain)
action$="Stop",gosub Action
if Device.@b; "If the device isn’t running, continue. Operation could be in"
  "Hold state"

  while phremain; "Wait until it is safe to abort"

Abort_2: "Abort issued during an active condition"
Device.C=7; "Save the phase restart identification number"
dishold; "Disable Hold command during abort routine"
on stop; "Add a dummy trap to catch attempts to stop operation"
return; "Abort has priority over stop"
action$="Abort"; gosub Action
Device.@a=0; "Turn the device off"
if Device.A; "clear device timer, if necessary"
  Device.@b=1
  while Device.A; "Caution: Do not use many while loops in abort routine,"
  "process could hang"
Device.@b=0; "Turn off reset switch"
Leave: "Common stop and abort exit routine"
aborted=aborted+1; "Count the number of tasks aborted"
Device.@c=0; "Disable the restart routine"
end; "Terminate this task"
Initialize:
Device.C=0; "Clear the phase restart identification number"
Device.@c=1; "Enable the restart routine"
phase$="Initializing"; "Save initial phase name"
usegraphic(WINI-B,except,0); "Designate graphic for Active alarm button"
while holding; "If the hold flag has been set, wait here"
opremain=1000; "Initialize the operation countdown timer"
display("INIT", opremain); "Display INIT starting time on BFACE"
display("", phase$); "Display phase name in BFACE, no initial text (NONE on BAM)"
display("WDOG", watchdog); "Display the watchdog timer in BFACE"
priority(500); "Set the script execution speed"
Setup: "Normal operation activity begins here"
Device.C=1; "Save phase restart identification number"
phase$="Setup"; "Save the new phase name"
if abort_all; start Abort_0; "If abort trap was missed, start abort routine"
if stop_all; start Stop_0; "If stop trap was missed, start stop routine"
enhold; "Enable Hold command and trip the on hold trap"
display ("RUN", opremain); "Display time change from INIT to RUN"
if Device.@a|Device.@b; "If device is not ready, generate an alarm"
  almsg(8,1,"DEVICE NOT READY")
  hold; "Trip global on hold trap"
Device.B=Setpoint; "Set the watchdog timer setpoint"
Run_it: "This phase starts the device and waits for it to finish"
Device.C=2; "Save phase restart identification number"
phase$="Running"; "Save the new phase name"
on abort; "Set on traps"
  start Abort_1
on stop
  start Stop_1
on hold; "This phase turns off devices before using the hold subroutine"
  Device.@a=0
  gosub Hold_it
  Device.@a=1
  return
enhold; "If hold flag is already enabled, trip the hold trap"
phremain=setpoint=5;
Device.@a=1;
  if ~(watchdog=phremain);
    alarm(2,1,"TIMEOUT";
    action$="Timeout";
    until Device.a;
    clear(2,1,"TIMEOUT");
  watchdog=0; "Clear the watchdog display"
Teardown: "Following Setup phase, put plant into safe state for transition to"
  "another operation"
Device.C=3
phase$="Teardown"
on abort
  start Abort_2
if abort_all
  start Stop_2
if stop_all
  start Stop_2
on hold
  Device.@b=0
  gosub Hold_it
  Device.@b=1
  return
enhold; "If the Hold command flag is set, trip the on hold trap"
Device.@a=0; "Disable the device, which has stopped running"
phremain=30; "Use phremain to time this phase"
Device.@b=1; "Activate a device to do something (such as open a drain valve)"
while watchdog=phremain;  "Set the watchdog display time and wait until"
   "phremain=0)"
Device.@b=0;  "Deactivate a device"
Report:  "Following all device activity, perform report calculations"
Device.C=4
phase$="Reporting"
"There are no local on traps because device is not running. However, you can"
"still trip the global on stop and global on abort traps if the report routine"
"hangs somehow the r(eport) has to be tested for each print statement, or the"
"task could hang trying to print to a disk that isn’t available. Testing has"
"to be done on the same line as the print instruction, or the trap will"
"repeatedly return to the print instruction and trip it again in an endless"
"loop. The ‘if r’ condition tests the state of the shared local variable ‘r’"
"to determine whether to disable the print instruction"
"The ‘r’ is set in the start script"
if r; print(1,"^n%TIME the first of several lines, possibly")
if r; print(1,"^n%TIME the second of several lines, possibly")
if r; print(1,"^n%TIME Finished %s", opname$)
else; flog(logname$,"Finished %s", opname$)
Device.@c=0;  "Disable restart routine"
end;  "Terminate this operation - exit(n) can also be used"

********************
End of WINI-B, Operation
"Unit Definitions"
"Define shared aliases and shared variables"
shared alias SSDsys_abort: Except/a; "Safety shutdown and system abort"
shared alias Master_abort: Except/b; "Master plant abort"
shared alias Process_abort: Except/c; "Process unit was aborted"
shared alias Process_stop: Except/d; "Process unit was stopped"
shared alias Process_hold; Except/e; "Process unit was put on hold"
shared local aborted; "At least one of the operations has aborted"
shared local abort_all; "Signal to abort, set by this script"
shared local stop_all; "Signal to enable print statements"
shared local r; "Signal to enable print statement"
shared string logname$; "The log name used by all operations for this recipe"

End of WINI-B, Unit
RBL Script  WINI, Startup

Type  START

"This script performs exception handling for following types of exceptions:"
"(1) Stop, Abort, or Hold commands issued from a BFACE process graphic."
"(2) An attempt to print to a report file that fails"

"Declare aliases and variables"
shared alias SSDsys_abort; "Safety shutdown and system abort"
shared alias Master_abort; "Master plant abort"
shared alias Process_abort; "Process unit was aborted"
shared alias Process_stop; "Process unit was stopped"
shared alias Process_hold; "Process unit was put on hold"
shared local aborted; "At least one of the operations has aborted"
shared local abort_all; "Signal to abort, set by this script"
shared local stop_all; "Signal to enable print statements"
shared local r; "Signal to enable print statement"
private local answer; "Stores user response to input query"
private local remain; "Displays opremain value on BAM, not on BFACE"
shared string logname$; "The log name used by all operations for this recipe"
private string disk$, report$; "Report volume and file name"
disk$="WINI-B", report$="Except"

"The startup script traps are always active. They are set to trip in response"
"to recipe Stop and Abort commands. They set flags that global to the operation"
"of this recipe All operations test flags before proceeding in each active"
"phase, to prevent a trap from being missed during download or restart"
on abort
  abort_all=1
  return
on stop
  stop_all=1
  return

"The following traps are used to notify the recipe operations of conditions"
"that occur during a safety shutdown of the system or ControlBlock interlocks"

"****************************************************************************** WARNING ****************************************************************************** "
" The safety of the process must not depend only upon the operation of a trap. "
" It is possible to miss an alarm or event message. This script may not even be "
" running when the event occurs. Do not use a high temperature alarm to trigger "
" an on abort trap. Use an interlock to reduce the temperature and an event on the "
" interlock to abort the recipe execution."  
"****************************************************************************** WARNING ****************************************************************************** "
on SSDsys_abort
  recabort
  return

on Master_abort
  recabort
  return
on Process_abort
  recabort
  return
on Process_stop
  recabort
  return
on Process_hold
  recabort
  return
"The on file_1_err trap handles report problems for all recipe operations. This"  
"script defines report file 1 and handles errors. The operation scripts just"  
"print to 'print(1,..)'. They also include an on file error trap that suspends"  
"the script until this routine solves the printing problem. This trap tries to"  
"reopen the report file for one minute,and then asks for operator intervention."  
"If the report file cannot be opened, the operator can choose to disable"  
"reporting. The operation scripts can be set up to flog any available"  
"disk that can receive messages if reports are disabled. Once disabled,"  
"reports cannot be enabled later. (Reports could be enabled if shared local r"  
"was an alias)."

```plaintext
on file_1_err
  opremain=1M
  while open(1,disk$,report$,1)<0;  "Open existing report file and append print"
    "to last generation"
    sleep(10);  "sleep prevents excess log messages"
    if (remain=opremain);  "If allotted time runs out"
      prompt
      print(*,"Cannot open report %s on disk %s",report$,disk$)
      print(*,"^n You have two options:"
      print(*,"^n Disable the report for the rest of the recipe"
      print(*,"^n Or Allow report retries to continue.^n^n")
      input("^nDo you want to disable the report? (Y/N)",answer)
      if answer==1
        flog(logname$,"Aborted report %s on disk %s",report$,disk$)
        r=0;  "Show the operation that reports are disabled"
        return;  "to waiting state"
        opremain=30M;  "Do this again in 30 minutes"
      logname$="Except"
      opremain=1M
      while open(1,disk$,report$,0)<0;  "Open new report file"
        sleep(10);
        if ~(remain=opremain);  "If time ran out"
          almsg(8,1,"CANNOT OPEN REPORT"
          opremain=30M;  "Generate alarm again in 30 minutes"
        r=1
        print(1,"%TIME %DATE Report header"
        "The start script is suspended in a waiting state here. No on traps are active"
```
ControlBlock Output

Figure A.3 and Figure A.4 show the ControlBlock output logic used in the example:

**Figure A.3. ControlBlock Output, Step a**

**Figure A.4. ControlBlock Output, Step b**
Appendix B:
Timing Backups to Nonvolatile Memory

This appendix demonstrates a script for timing Coordinator Processor (CP) backups to nonvolatile memory. The script is intended to help you assess system demands on the CP that can affect CP backup times.

CP backups are intended as a security feature to protect the integrity of the batch operation in the event of a CP reboots, crashes, or switches to a redundant card. CP nonvolatile memory is normally backed up every 10 to 20 seconds. However, increased system demand on the CP can increase the length of time between backups. Backup intervals of over one minute are possible.

The longer it takes for the CP to back up to nonvolatile memory, the more vulnerable processes configured in batch scripts are to CP reboots, crashes, or switches to redundant cards. The following script is intended to help you assess the risk.

**NOTE:** To reduce the vulnerability of your batch processes, we strongly recommend that you configure all time-critical batch processes in ControlBlock logic and include restart routines in batch scripts. For more information on restart routines, see Appendix A.

```plaintext
1: on power_up
2:   counter=0; "Prevents incorrect log of BACKUP time through powerup"
3: start RESTART
4:
5: priority(1000); "Set to 1000 to assure top priority."
```
RB: B-2

6: record_it=1; "Set to one to begin test, record results once every half hour."
7: timer=sss; "Set timer before entering 'RESTART' for first pass only."
8: least=1000; "Start off with a large value"

RESTART:
13: "Calculate the backup time and reset timer."
14: counter=counter+1
15: backup; "Wait for the next backup"
16: timer=sss-timer; "Subtract the current time from the time before backup"
17: last_time=timer; "Store last cycle time."
18: timer=sss; "Restart timing sequence for next cycle."

if counter > 2; "Throw away the first reading"
19: if last_time > greatest; "Determine if this is the longest time."
20: greatest=last_time
21: if last_time < least; "Determine if this is the shortest time."
22: least=last_time
23: total=total+last_time
24: avg_count=avg_count+1
25: running_avg=total/avg_count

if last_time>100; "If the backup time is greater than 100 seconds, alarm."
26: alarm(3,1,"%NODE BACKUP TIME Long")

if ~(time(2)%30); "TIME(2) is the number of minutes."
27: if record_it; "If not recorded this half hour"
28: flog("BACKUPS","Last half hour average is %3.2f",total/avg_count)
29: flog("BACKUPS"," Node-%NODE Longest Backup Time is %d seconds",greatest)
30: flog("BACKUPS"," Node-%NODE Shortest Backup Time is %d seconds",least)
31: record_it=0; "Set the bit to a zero so it will only report once."
32: avg_count=0; total=0; "Reset for next half hour."

else "After the next minute:"
33: record_it=1; "Reset the bit to 1 so it will report correctly in a half hour"

start RESTART

private local timer, counter, least, greatest, record_it, last_time
private local total, running_avg, avg_count

******************************************************************************
Index

A

abort, RBL instruction, 1-6-2
abort _task, RBL instruction, 1-6-3
access level
  console login, 1-14-5
  permission mapping, 1-14-6
  user identification, 1-14-2
  user permission, 1-14-3
acqfirst, RBL instruction, 1-10-7– 1-10-8
  status values, 1-10-7
  timeout, 1-10-7
acquire, RBL instruction, 1-10-7– 1-10-8
  status values, 1-10-7
  timeout, 1-10-7
acquire/release instructions, 1-10-1– 1-10-34
  concept, 1-10-4
  definition, 1-10-1
  example, 1-10-3
  instruction format, 1-10-7– 1-10-21
  overview, 1-10-2
  passive and active masters and slaves, 1-10-32
  request queue, 1-10-1, 1-10-4
  responding to loss of master or slave task, 1-10-26– 1-10-31
  status values, 1-10-5, 1-10-10
  task synchronization, 1-10-23
acqwait, RBL instruction, 1-10-9
  status values, 1-10-9
alarm, RBL instruction, 1-9-2– 1-9-3
alarm on traps, 1-8-7
alarm/log/report instructions, 1-9-1– 1-9-21
aliasdim, RBL instruction, 1-3-12
aliases
  dot operators, 1-3-9
  format, 1-3-7
  rules, 1-3-9
align_on, RBL instruction, 1-7-2
align_wait, RBL instruction, 1-7-2
almsg, RBL instruction, 1-9-2– 1-9-3
alv2str$, RBL instruction, 1-14-2
andbit, RBL instruction, 1-4-2
array functions
  array performance, 1-3-24
  array_copy, 1-3-23– 1-3-27
  array_init, 1-3-23– 1-3-27
file_copy, 1-3-30
file_delete, 1-3-31
file_rename, 1-3-32
PeerWay messages, 1-3-23
varray_file, 1-3-33
array_copy, 1-3-23– 1-3-26
concept, 1-3-25– 1-3-26
performance, 1-3-24
array_init, 1-3-23– 1-3-27
concept, 1-3-25– 1-3-26
performance, 1-3-24
arrays
  aliasdim, 1-3-12
  definition, 1-3-10– 1-3-21
  dim, 1-3-14
  stringdim, 1-3-16
asc, RBL instruction, 1-4-4
ASCII string operations
  asc, 1-4-4
  byte2flag, 1-4-5– 1-4-9
  chr$, 1-4-11
  flag2byte$, 1-4-5– 1-4-9
  str$, 1-4-14
  strcat$, 1-4-15, 1-15-13
assigning priority to batch tasks, 1-6-25

B

backup, RBL instruction, 1-6-4
badunit, RBL instruction, 1-7-4
Batch Acquire Queues screen, 1-10-24
batch management instructions, 1-6-1– 1-6-41
Batch Script screen, 1-2-3
bcc$, RBL instruction, 1-15-2
  checksum summation method, 2-2-6
begin _task, RBL instruction, 1-6-5
begin_recipe, 1-7-5
branching and conditional instructions, 1-5-1– 1-5-25
break, RBL instruction, 1-6-6
bumpid, RBL instruction, 1-6-7
byte operations
  byte2flag, 1-4-5– 1-4-9
  flag2byte$, 1-4-5– 1-4-9
  getbyte$, 1-15-12
  getnbyte$, 1-15-16
putbyte, 1-15-14
putnbyte, 1-15-15
swab$, 1-4-18
byte2flag, RBL instruction, 1-4-5–1-4-9
status values, 1-4-7, 1-4-10
using, 1-4-7–1-4-8
calling up, scripts, 1-2-7, 1-2-22
chain, RBL instruction, 1-5-2–1-5-6
script not found, 1-5-2
string variables, 1-3-39
using private aliases, 1-5-6
using shared aliases, 1-5-4–1-5-5
changing variables, 1-3-42
character to decimal conversion, 1-4-4
checksum
bcc$, 1-15-2
CRC (cyclic redundancy check), 1-15-2
longitudinal redundancy check method, 1-15-13, 2-2-7
lrc8$, 1-15-13
summation method, 1-15-2, 2-2-6
chflush, RBL instruction, 1-15-4
child script, 1-5-2–1-5-3
chr$, RBL instruction, 1-4-11
chwait, RBL instruction, 1-15-5
clear, RBL instruction, 1-9-2–1-9-3
clear_alarm_queue, RBL instruction, 1-8-12
close, RBL instruction, 1-9-7–1-9-8
instruction format, 1-9-7
status values, 1-9-9
commclose, RBL instruction, 1-15-6
comments, 1-1-11
commopen, RBL instruction, 1-15-7
communication parameters, 1-15-7
status variables, 1-15-7
commstat, RBL instruction, 1-15-10
Communication FlexTerm, 2-1-2
communications
buffer
chflush, 1-15-4
commopen, 1-15-7
overview, 1-15-6, 2-1-3–2-1-4
channels, 1-15-6–1-15-11
parameters, 2-1-2
communications parameters, 2-1-3
comparing floating point values, 1-3-43
conserving volatile memory, 1-3-43
constants, 1-3-34
cont_task, RBL instruction, 1-6-10
continue, RBL instruction, 1-6-8
Continuous Faceplate screen, 2-2-2–2-2-3
ControlBlock
RBL Controller
configuring, 2-2-2
description, 2-1-3, 2-2-1
RBL Interface
addresses, 1-1-12
modes, 1-1-13
values, 1-1-12
Coordinator Processors, setting batch priorities, 1-6-24
copying scripts, 1-2-9
copying scripts from script screen, 1-2-12
CRC (cyclic redundancy check), 1-15-2
creating an initial script, 1-2-2
creating an RBL Files folder, 1-2-2
creating RBL files, 1-2-2
DEC to IEEE floating point format, 1-15-11
decimal scaling to display scaling, 2-2-8
decimal to character conversion, 1-4-11
dectoieee, RBL instruction, 1-15-11
deleting scripts, 1-2-8
dequeue, RBL instruction, 1-10-10–1-10-11
status values, 1-10-10
synchronization, 1-10-23–1-10-24
dim, RBL instruction, 1-3-14
dishold, RBL instruction, 1-6-11
disownunit, RBL instruction, 1-7-25–1-7-26
display, RBL instruction, 1-7-8
dot operators, 1-3-9
edit script, 1-1-3
editing scripts, 1-2-10
colors, 1-2-21
copying a script, 1-2-12
delete buffer, 1-2-13
deleting text, 1-2-13
entering text, 1-2-11
inserting text, 1-2-13
moving, 1-2-16–1-2-17
parsing, 1-2-15
reading, 1-2-16–1-2-17
searching and replacing text, 1-2-18
writing to disk, 1-2-15
else, RBL instruction, 1-5-16
elseif, RBL instruction, 1-5-18
end, RBL instruction, 1-6-12
endchain, RBL instruction, 1-5-2–1-5-6
enhold, RBL instruction, 1-6-11
entering text in scripts, 1-2-11
event, RBL instruction, 1-9-4
exception handling, A-1–A-16
Exclusive-OR operation, 2-2-7
exit, RBL instruction, 1-7-10
fetch, RBL instruction, 1-7-12
fetch$, (RBL instruction), 1-7-12
file_copy, 1-3-30
file_delete, 1-3-31
file_rename, 1-7-15– 1-7-16
finish, RBL instruction, 1-6-13
flag2byte$, RBL instruction, 1-4-5– 1-4-9
status values, 1-4-7, 1-4-10
using, 1-4-7– 1-4-8
FlexTerm Port I/O cards, 2-1-2
flog, RBL instruction, 1-9-5– 1-9-6
for, RBL instruction, 1-5-8
formula script, 1-1-3
getbit, RBL instruction, 1-4-12
getbyte$, RBL instruction, 1-15-12
getmaterial, RBL instruction, 1-7-17
getmatlim, RBL instruction, 1-7-18
getnbyte$, RBL instruction, 1-15-16– 1-15-17
getssm, RBL instruction, 1-13-2
getsss, RBL instruction, 1-13-3
global variables, 1-3-36
for, RBL instruction, 1-5-10– 1-5-11
masterof, RBL instruction, 1-10-12
status values, 1-10-12
materialdesc, RBL instruction, 1-7-21
materialunits, RBL instruction, 1-7-22
mathematical operators, 1-1-18– 1-1-19
max, RBL language, 1-6-19
min, RBL language, 1-6-19
moving scripts, 1-2-16
maintaining script files, 1-2-6– 1-2-9
mapunit, RBL instruction, 1-7-19
master task
concepts, 1-10-4
removing master request from slave task acquire queue, 1-10-32
responding to loss of master task, 1-10-27– 1-10-32
masterof, RBL instruction, 1-10-12
status values, 1-10-12
materialdesc, RBL instruction, 1-7-21
materialunits, RBL instruction, 1-7-22
mathematical operators, 1-1-18– 1-1-19
max, RBL language, 1-6-19
min, RBL language, 1-6-19
moving scripts, 1-2-16
nesting, RBL
limits, 1-1-10
stack, 1-1-10
next, RBL instruction, 1-6-20
nil, RBL instruction, 1-13-8
nonvolatile memory
arrays, 1-3-10
limits, 1-1-16
string variables, 1-3-39
symbol backup, 1-3-1
notify, RBL instruction, 1-11-2
nvfree, RBL instruction, 1-6-21
alarm trap priority designation, 1-8-7
alarm traps without priority designated, 1-8-8
definition, 1-8-1
on lost master, 1-8-10
on trap instructions, 1-8-1– 1-8-13
alarm trap priority designation, 1-8-7
alarm traps, 1-8-7
alarm traps without priority designated, 1-8-8
definition, 1-8-1
execution order, 1-8-5
execution priority, 1-8-3
more than one trap per ControlBlock address, 1-8-3– 1-8-4
on abort, 1-8-9
on acquire_err, 1-8-10, 1-10-13
on chain_err, 1-8-9
on file_n_err, 1-8-10
on hold, 1-8-9
on input_err, 1-8-10
on link_err, 1-8-9
on lost_master, 1-10-14– 1-10-15
on lost_slave, 1-8-11, 1-10-16– 1-10-17, 1-10-27
on nest_err, 1-8-10
on no_backup, 1-8-11
on ownunit, 1-8-9
on power_up, 1-8-10
on stop, 1-8-9
on unacquire, 1-10-19
on unowned_unit, 1-7-24
overview, 1-8-2
system traps, 1-8-9– 1-8-10
on unowned_unit, RBL instruction, 1-7-24
open, RBL instruction, 1-9-7– 1-9-8
instruction format, 1-9-7
status values, 1-9-9
opremain, RBL instruction, 1-6-22
orbit, RBL instruction, 1-4-2
ownunit, RBL instruction, 1-7-25– 1-7-26

P
parameter limits, RBL, 1-1-16
parent script, 1-5-2– 1-5-3
parsing scripts, 1-2-15
password security instructions, 1-14-1– 1-14-8
phases
- format, 1-1-4
- gosub, 1-5-10– 1-5-11
- labels, 1-1-5
phremain, RBL instruction, 1-6-22
plant units
- bad units, 1-7-4
- ownership, 1-7-25
print, RBL instruction, 1-9-10– 1-9-18
constant conversion options, 1-9-15
decimal conversion options, 1-9-16
format control options, 1-9-14
print failure, 1-7-15
print options, 1-9-13
system string options, 1-9-18
variable conversion options, 1-9-17
print$, RBL instruction, 1-9-12
printing a script, 1-2-6
priority
- assigning to process functions, 1-6-24
- RBL instruction, 1-6-23– 1-6-25
- CP resources, 1-6-24
- setting execution rate, 1-6-25
private symbols
- batch recipe, 1-3-2
- concept, 1-3-3, 1-3-5
procedure script, 1-1-3
prompt, RBL instruction, 1-11-4
putbyte, RBL instruction, 1-15-14
putmaterial, RBL instruction, 1-7-17
putbyte, RBL instruction, 1-15-15
puttime, RBL instruction, 1-13-9
quietlog, RBL instruction, 1-9-19
R
RBL, timestamp functions, 1-13-1
RBL ControlBlock, 2-1-3, 2-2-1
RBL Controller
- commopen parameters, 1-15-7
- commopen status variables, 1-15-8
- Communication FlexTerm, 2-1-2– 2-1-3
- communications channel, 1-15-6– 1-15-12
- configuring RBL ControlBlock, 2-2-2– 2-2-3
- Continuous Faceplate ControlBlock, 2-2-2– 2-2-3
- hardware, 2-1-2– 2-1-3
- link limitation, 2-1-1
- longitudinal redundancy check, 1-15-13, 2-2-7
- loop back test, 2-2-13– 2-2-16
- program image, 2-1-4
- RBL ControlBlock, 2-1-3, 2-2-1– 2-2-3
- RBL Monitor screen, 2-2-4– 2-2-5
- RBL scripts, 2-1-3, 2-2-4
- receive queue, 1-15-4
- scaling equations, 2-2-8
- software, 2-1-3– 2-1-4
- summation check, 1-15-2, 2-2-6
- transmit queue, 1-15-5
- weigh scale sample program, 2-2-9– 2-2-12
RBL Controller instructions, 1-15-1– 1-15-17
RBL File Contents screen
- example, 1-2-4
- field definitions, 1-2-5
RBL Files folder, 1-2-2, 2-1-5
RBL Language
- acquire/release instructions, 1-10-1– 1-10-34
- alarm/log/report instructions, 1-9-1– 1-9-21
- branching and conditional instructions, 1-5-1– 1-5-25
- concepts, 1-1-1
- editing scripts, 1-2-10– 1-2-16
- instructions, 1-1-8
- interactive instructions, 1-11-1– 1-11-8
- maintaining scripts, 1-2-6– 1-2-9
- numeric and string symbols, 1-3-1– 1-3-47
on trap instructions, 1-8-1 – 1-8-13
Password Security Instructions, 1-14-1 – 1-14-8
RBL Controller instructions, 1-15-1
recipe management instructions, 1-7-1 – 1-7-34
scaling instructions, 1-12-1 – 1-12-7
script management, 1-2-1 – 1-2-23
string functions, 1-4-1 – 1-4-21
RBL language, batch management instructions, 1-6-1 – 1-6-41
RBL Monitor screen, 2-2-4
RBL script, 2-1-3, 2-2-4
reading scripts, 1-2-16
realtime, RBL instruction, 1-13-10
recabort, RBL instruction, 1-6-26
rechold, RBL instruction, 1-6-26
recipe management instructions, 1-7-1 – 1-7-34
recipe_local, RBL instruction, 1-7-27
recresume, RBL instruction, 1-6-26
release, RBL instruction, 1-10-20
relwait, RBL instruction, 1-10-21
renotify, RBL instruction, 1-11-2
reprompt, RBL instruction, 1-11-6
restart routine, A-2 – A-16
resume, RBL instruction, 1-6-27
resume_task, RBL instruction, 1-6-27
retry, RBL instruction, 1-6-29
retry instructions, 1-1-14
return, RBL instruction, 1-5-20
run_recipe, RBL instruction, 1-7-28 – 1-7-29
rvdim, 1-3-20 – 1-3-21
rvstringdim, 1-3-20 – 1-3-21
colors, 1-2-21
copying, 1-2-9
creating, 1-2-2
deleting, 1-2-8
displaying, 1-2-4, 1-2-23
edit, 1-1-3
editing, 1-2-10 – 1-2-23
maintaining, 1-2-6 – 1-2-9
management, 1-2-1 – 1-2-23
parent, 1-5-2 – 1-5-3
printing, 1-2-6
RBL File Contents screen, 1-2-4
setbit$, RBL instruction, 1-6-26
seteot, RBL instruction, 1-15-16 – 1-15-17
setting task priority, 1-6-25
shared symbols
batch recipe, 1-3-2
concept, 1-3-2, 1-3-5
defining, 1-3-6
skipping symbol lines, 1-3-43
slave task
concept, 1-10-4
on lost_slave, 1-10-16
responding to loss of slave task, 1-10-27 – 1-10-32
sleep, RBL instruction, 1-6-30 – 1-6-31
ssm, RBL instruction, 1-6-32
sss, RBL instruction, 1-6-32
start, RBL instruction, 1-5-21
started_by, RBL instruction, 1-7-31
status values
acquire/release instructions, 1-10-5, 1-10-10
commonopen, 1-15-8
definition, 1-1-9
RBL
byte2flag, 1-4-7, 1-4-10
flag2byte$, 1-4-7, 1-4-10
open, 1-9-9
status_task, RBL instruction, 1-6-33 – 1-6-34
stop, RBL instruction, 1-6-36
stop_task, RBL instruction, 1-6-37
str$, RBL instruction, 1-4-14
strcat$, RBL instruction, 1-4-15
strcmp, RBL instruction, 1-4-16
string functions
acquire/release, 1-10-10
andbit, 1-4-2 – 1-4-3
asc, 1-4-4
byte2flag, 1-4-5 – 1-4-9
chr$, 1-4-11
definition, 1-4-1
flag2byte$, 1-4-5 – 1-4-9
getbit, 1-4-12
gettrend, 1-13-5
orbit, 1-4-2 – 1-4-3
setbit$, 1-4-13
str$, 1-4-14
strcat$, 1-4-15
strcmp, 1-4-16
S
scaling instructions, 1-12-1 – 1-12-7
scdp, RBL instruction, 1-12-3
schi, RBL instruction, 1-12-4
callback, RBL instruction, 1-12-5 – 1-12-9
colo, RBL instruction, 1-12-4
scripts
 calling up, 1-2-7
 child, 1-5-2 – 1-5-3
RS3: Rosemount Basic Language Manual
strlen, 1-4-17
swab$, 1-4-18
val, 1-4-19
string variables, 1-3-39
  characteristics, 1-3-40
  definition, 1-3-39
  indexing, 1-3-2, 1-3-40
  instruction format, 1-3-39
stringdim, RBL instruction, 1-3-16
strlen, RBL instruction, 1-4-17
structured programming, 1-1-1
summation method, 2-2-6
swab$, RBL instruction, 1-4-18
symbol table, 1-3-1
symbols
  binary values, 1-3-6
  conserving memory, 1-3-43
  declaration, 1-3-1
  definition, 1-1-7, 1-3-1
  floating point values, 1-3-43
  hex values, 1-3-6
  hints for using, 1-3-42
  private, 1-3-2
  shared, 1-3-2
  shared and private
    batch recipe, 1-3-2
    Batch Run screen, 1-3-4
sync, RBL instruction, 1-6-38
system on traps, 1-8-9

T

tags
  byte2flag, 1-4-7
  flag2byte$, 1-4-7
  scliffe, 1-12-5
  unsc, 1-12-5
time, RBL instruction, 1-6-39–1-6-40
timeout status, acquire/release, 1-10-7
timestamp functions, 1-13-2–1-13-10

U

unacquire, RBL instruction, 1-10-19
undisplay, RBL instruction, 1-7-8
unit script, 1-1-3
unsc, RBL instruction, 1-12-5–1-12-6
until, RBL instruction, 1-5-22
usegraphic, RBL instruction, 1-7-33

V

val, RBL instruction, 1-4-19
variables
  changing, 1-3-42
  display values, 1-7-8
  floating point values, 1-3-43
  global, 1-3-36
  local, 1-3-38
  string, 1-3-39
varray_file, 1-3-33
vdim
  instruction format, 1-3-18
  RBL instruction, 1-3-18
vfree, RBL instruction, 1-6-21
virtual arrays
  computing disk space, 1-3-22
  definition, 1-3-17
  instruction format, 1-3-18
  rvdim, 1-3-20–1-3-21
  rvstringdim, 1-3-20–1-3-21
  vdim, 1-3-18
  vstringdim, 1-3-19
volatile memory, limits, 1-1-16
vstringdim, RBL instruction, 1-3-19

W

wait_bid, RBL instruction, 1-7-35–1-7-36
weigh scale sample program, 2-2-10
while, RBL instruction, 1-5-24
  special cases, 1-5-26
while acquire, RBL instruction, 1-5-26
while nobackup, RBL instruction, 1-5-26
whois, 1-14-5