Index of Command Keywords and Symbols (by Page and Footnote)

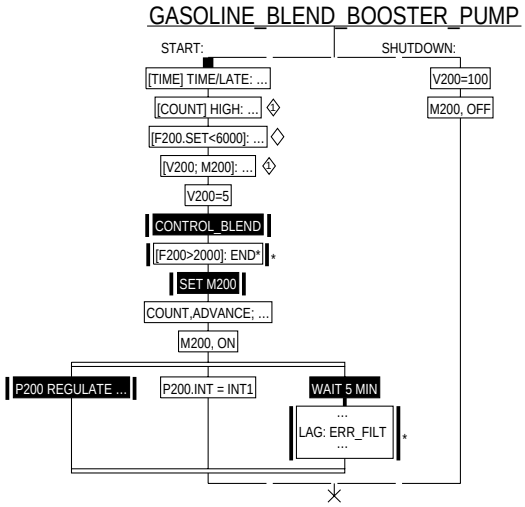
(29
) 29
** 35
.. 51, 52, 53, 54
: 14, 21, 27, 35, 36, 45, 54, 47, 52, 55, 58, 62
:: 47
; 26, 31, 47, 55, 64
< 5, 48, 59, 64
> 5, 48, 59, 64
ABORT 5, 16, 27
ACTIVE 5, 26, 64
ADVANCE 59, 62
ALL 27, 47, 65
ANY 47, 65
ANY_NOT 47, 65
ARCHIVE 44
AT 65
BACKUP 79
BLEND 67, 79
BLOCK 34, 36, 45, F27
BOOK 30, 55
CALL 22, 27
CC 80
CONTINUE 27
CONTINUED 16, 26
CONV 49, 51
COUNTING 65
COUNTS 49, F68
DATE 62, F89
DeadTime 43
DECOUPLE(X) 71, 78
DEV 50
DIN 49
DIFFERENT 42
DLBits 40
DONE 59, 63
DOUT 49
DSense 40
DSet 40
EARLY 59, 62
ELSE 47, 65
END 26, 29, 32, F8, F81
EXECUTE 31
EXPRESSION 22
FAIL 31, 55, F50
Feedback 40
FeedForward 43
FEEDFWD(X) 77
FILT 49
FINAL 42
FIRST 52, 55
FOR 65
FREEZE 31, 42, 49, 55
Function 43, 76
HI 50, 56, 64
HICONSTR 73
HIGH 63
HILIMIT 77
HISELECT 64
HOLD 5, 62, 63
HOURS 62
IN 12, 15, 33, 40, 47, 49, 65
INACTIVE 5
INITIAL 42
LAST 52, 55
LATE 59, 62, 63
LBits 40
LeadLag 38, 43
LO 50, 56, 64
LOCONSTR 77
LOLIMIT 77
LOSELECT 64
LOW 63
MAKE_RECIPE 20, 36 43, F23
MATCH 52, 55
MAX 49, 50, 56, F86
MESSAGE 22, 63
MIN 49, 50, 56, 62, 65, 66, F86
MISNEST 55
MOC 80
NAME 40, 48, 49, P58
NESTED 55
NEXT 30, 42, 52, 55
NEXTSTATE 30, 68
NEXTSTEP 39, F57
NEXTSV 42
NONE 47, 65
NSRESTORE 39 , 40, P57
NSSAVE 39 , 40, P57
ON 47, 59
OUT 40, 43, 49
OVER
OVERFLOW 56, 59, 62, 63
OVERRIDE 16, 31, 47
OVSAVE 39, 40, P57
OVSV1 39, 40, P57
OVSV2 39 , 40, P57
PID 43
PIDE 43
PIDX 43
PREV 52, 55
PROFILE 69
RAMP 66
Ratio 43
REGULATE(X,E) 76
Remainder 28, 39, 63
RESET 59, 62
RESTART 62, 63
Restore 44
Result 18, 63, 64
RETRY 64
SEC 59, 62
Sense 40
SEQUENCE 68
SET 39, 40, 49, 50, 56
Set 40, 70
SCALE 50
SOME 47, 65
SPLR 78
STATE 46, 49, 58, 59
State 64
STATES 49, 55, 58, 59
STREAM 68
TIME 49, 59, 62, 68
TIMING 64
TO 65
TRUTH_TABLE 60
UNBOOK 31, 42, 55
UNDEFINED 56, 59, 62
UNDER
UNDERFLOW 56, 59
UNFREEZE 32, 49, 55, F50
UNITS 49
UnMake 44
VALUE 39, 49, 50, 56
WAIT 65
_ 31

Index of Concepts (Key Entry *Italicized* if not in a title, Key Page underlined)

Activity 16, 17, 25, 29
Attribute 48
Block/Block Call 17, 20, 28, 34, 35, 36
Booking 31
Call 6, 13, 17, 22, 24, 26, 27, 30, 31, 33
Context 48, 50, 56, 58, 59, F63, F66, F70, F86
Footnote 6, 17, 19, 22, 25, 45, F92
Freezing 31
Idiom/Idiom Call 17, 24, 34, 37, 48, 63, 69
List 14, 27, 33, 34, 36, 38, 42, 44, 47, 48, 51
Local State Environment 17, 31, 33, 47, 52, 63, 64
Loop Statement 6, 24, 32, 37, 42, 48, 69
Named Statements 47, 66, 71
Model/Modeling 7, 13, 32, 51, F31, F32
Null Statement Task 17, 29
Operation; Call/List 5, 6, 11 ,13, 14
Page 5, 7, 11, 14, 16, 17, 20, 22, 24, 25
Prefix: State, Scoping, Naming, Override 26, 31, 33, 47
Recipe/Recipe Call 7, 11, 20, 22, 32, 34, 36, 43, 51, 81
Sequential Function Chart 32
Single Statement Tasks 17, 29
State 5, 6, 10, 11, 12, 26, 30, 33, 34, 35, 46, 47, 57, 60, 63, 65, 67
SuperVariable 48
Tasks/Calls 25, 33
Theme Statement 65
User Renaming 13, 14, 50, 52, F31

GASOLINE_BLEND_BOOSTER_PUMP Page: Parameters

```
START
START:
   [TIME] TIME/LATE :     COUNT, RESTART; TIME, RESET
◈  [COUNT] HIGH :         "Terminated; Too Frequent Starts"; END◈
◇  [F200.SET < 6000] :    "Finished; Booster Not Needed"; END◇
   [M200; P200; V200];
◈    MAN/ID_MAN :         "Terminated; Loops in Manual"; END◈
   V200 = 5 %
    CONTROL_BLEND
   * [F200 > 2000] :      END*
    SET M200                                    M200
   COUNT, ADVANCE;       TIME, RESTART
   M200, ON
     P200          V200                  P200   R   HR   V200
      REGULATE +
      HIRTLIMIT
      SET
     P200. INT = INT1
       WAIT 5 MIN
        ABSERR = ABS(P200.SET- P200)
        LAG: ERR_FILT
          IN = ABSERR
          **
          ABSERR = OUT
     * [ABSERR > 3] :    P200. INT = INT2; END*
SHUTDOWN:
   V200 = 100 %
   M200, OFF
```



The reverse video on the left shows the format, as displayed on the engineers display, with active control statements being reverse video-ed. The Sequential Function Chart on the left represents a possible counterpart for the operator display, with the following caveats:

- Statements, which go beyond the limits of the Simplified Syntax, are abbreviated, with ellipses after the first statement in a line, or after a Prefix, or after the first variable and operator in a Loop or Theme statement.
- Only four sequenced paths are allowed aside each other on a page; any further paths will be indicated in ellipsis, with means for allowing the operator to select which paths are to be elided.
- Continuous statements (e.g. Idioms or single statement Activities) or Activities are indicated with vertical solid bars beside their associated box; associated diamonds or asterisks, indicating termination are echoed to the right of the box.
- Continuous Activities are indicated by several statements (at most three) and ellipsis contained in a box, with surrounding vertical solid bars, and any right hand marking diamonds and asterisks.

* Active statements or Continuous Activities are reverse video-ed.

More generally, the Operator Interface presents a special challenge. We lack the internal expertise to do a proper human interface. Even more important, our customers lack that same expertise; it is thus incumbent on any good companion Operator Interface system to automate as much of the interface building as possible from the ICL program itself. **And the Operator Interface is the most prominent part of the system, when seen by the prospective customer.** This Interface system needs its own language design. This would include the following elements:

- A Graphics language for creating process graphics, including:
  - the ability to distinguish parts of the graphic corresponding to parts of the process (or to the corresponding subOperations),
  - the ability to position graphic, or iconic value displays or lower level subOperation displays within this graphic.
- A standardized or defaulted set of displays covering all standard data for an Operation, including both static and graphic displays, and display of the natural data in each of the basic Pages. These displays should include standard displays generated directly from the control program, as well as user configured specialty displays.
- Hierarchical access strategy for descending through the different levels of the process, and for selecting the appropriate displays, either from the graphics, or from an idealized keyboard, in either case, with mouse or touch screen selection.
- The selections strategy and displays should support multiple/redundant strategies for communicating the process information, to support different user requirements and tasks.

Appendix I. Data File Access Formats

Appendix II. Binary Core/File Formats

left, each associated with a corresponding trend plot. The valve position **V100** is indicated on the gray indicator to the right. The current measurements and setpoints are also displayed numerically. The setpoints can be manipulated by the joystick buttons. The iconic diagram displays the state of override (The gray constraint controller icon indicates an inactive override.), as well as the cascade state (the black circle, vs. a gray or white one).

Theme Statements also have associated displays. The displays below correspond to the Ramp statement with its Footnotes:

RAMP Page: Procedures

```
RAMP TEMP:*[1]      START FOR VENTTIME;*[2]
                    HEAT TO COOKTEMP + BIAS IN HEATTIME MIN;*[3]
                    COOK AT COOKTEMP*[4] FOR COOKTIME;*[5]
          RAMP P100: COOL TO 0*[6] IN COOLTIME*[7]
```

RAMP Page: Details

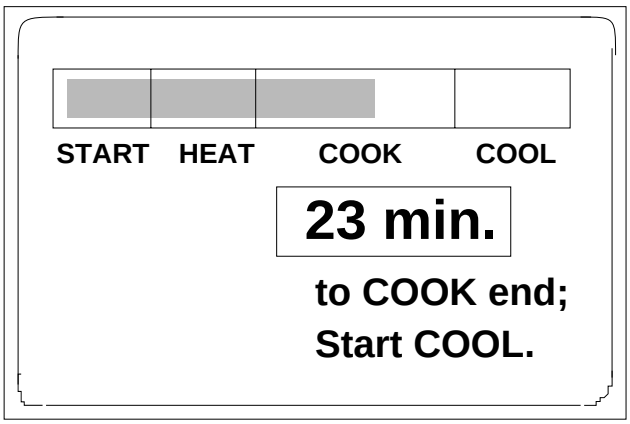
*[1] VENT, OPEN;
STEAMVALVE, OPEN;

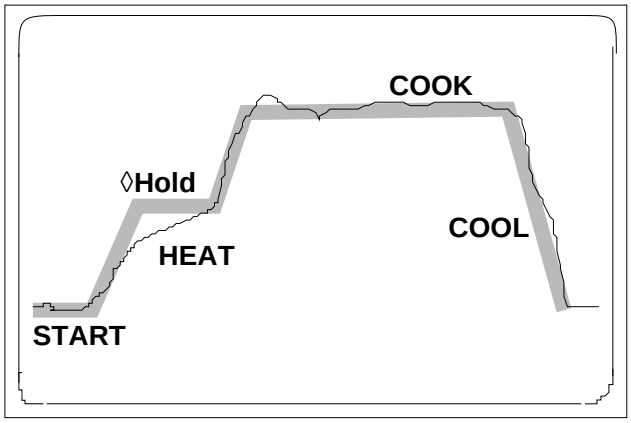
*[2] VENT, CLOSED;
STEAMVALVE, CLOSED;
"RECORD START BULB TEMPERATURE"LOG; BULBTEMP?LOG; TEMP LOG

*[3] "RECORD START BULB TEMPERATURE"LOG; BULBTEMP?LOG; TEMP LOG
BIAS = 0.0
PRODTEMP = TEMP

*[4] PRODTEMP = LOSELECT(PRODTEMP, TEMP)

*[5] TEMP LOG
STARTPRESSURE = PRESSURE.SET





The left hand display shows the state of processing on a time indicator, with a highlighted time for next state window. The right hand display shows a trend representation of the same action, with the solid gray line being the setpoint record and the thin black line being the actual measured record. The ◊**Hold** marking the setpoint record indicates a point where the control was suspended in **Hold** State. The Ramp (or any other Theme) Statement display should further permit operational altering of the State, not only allowing Holding but even allowing bypassing profile segments by changing the profile State.

Higher level Recipe and sequencing Activity Operational displays and operation might be based on the Sequential Function Charts shown earlier. These could be based on the earlier Simple sequencing version of the language, or on an assumed ability to generate restricted versions of sequential function charts for general Activities. The resulting operational display should allow the operator to skip statements or repeat them from the SFC display. For example, the Sequential Function Chart shown below[107] might be generated from the corresponding statements:

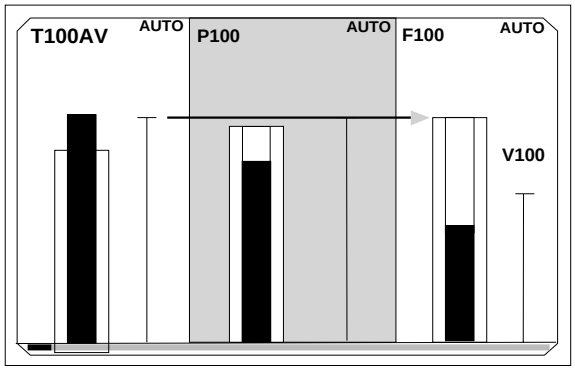
[107] The earlier discussion of Sequential Function Charts discusses the same example, rewritten in terms of the Simplified syntax, assuming appropriate SubTasks have been written.

Intended Operator Interface Function

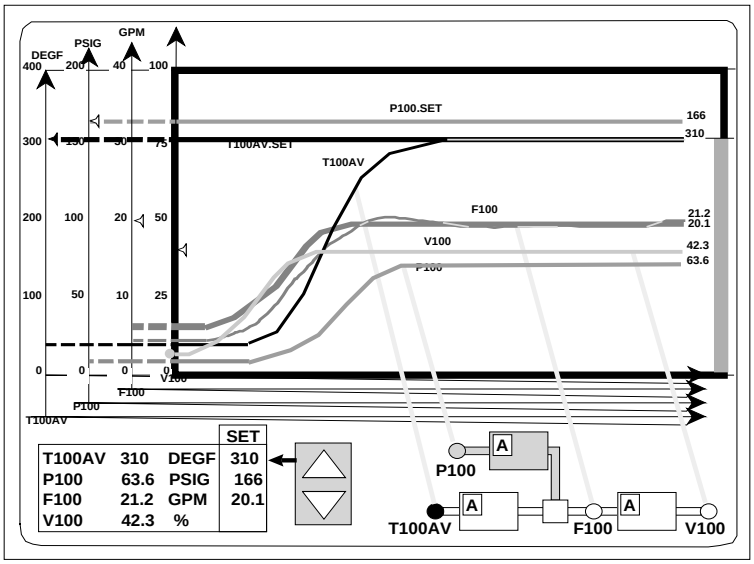
In the present day technology, the operator interface would also be defined within a GUI. The original and long run view of the ICL Operator Interface assumes that it would be based on a separate operator interface language. Where ICL defines controls, this language would define graphics and interfacing considerations. But it would be designed to fit with the ICL, using the natural hierarchy of the ICL and accessing the data of the ICL in terms of the natural ICL data components. Among other things this implies that there ought to be default displays for each level of ICL object: for the Operations, the Tasks, the Blocks, the Loops, the Ramps and other Theme statements.

At the Operation level, the operator display would include displays of key process variables as declared in the Operator Interface language, with a menu allowing the selection of the next lower level SubOperations, a second menu allowing the selection of operational displays for any of the built in controls defined within the Operation, and a menu for addressing any Recipe or sequenced Activities. The operational controls displays might include the process variables neutrally displayed in some variant of the Definitions Page, Idiom Loops displayed as below, and Theme Statements displayed as below. These would permit the operational manipulation of individual control parameters and States.

The two figures below show alternate views of a Loop Display, as might be generated automatically from the Idiom Loop Statement: **T100AV**$_{\textbf{REGULATE}}$ **P100**$_{\textbf{HICONSTR}}$ **F100**$_{\textbf{REGULATE}}$ **V100**[106]:



This first shows a multi-faceplate Loop Display, with **T100AV** in control manipulating **F100** and through it **V100**. The shaded **P100** faceplate indicates that the **P100** constraint controller is in bounds and inactive even as it is still in AUTO. The arrow extending from the output of the **T100AV** faceplate through to the **F100** controller setpoint also indicates the direct control in the cascade, bypassing the override. The black mark and gray bar at the bottom of all faceplates indicates a special cascade state (replacing the normal **AUTO** function). The position of the black ark indicates which of the three controllers is under operator setpoint control. All controllers to its left are in **MANUAL** and all controllers to its right are in **AUTO**. The individual **AUTO/MANUAL** indicators on each faceplate can be separately set to indicate local operator takeover of the role of that controller.



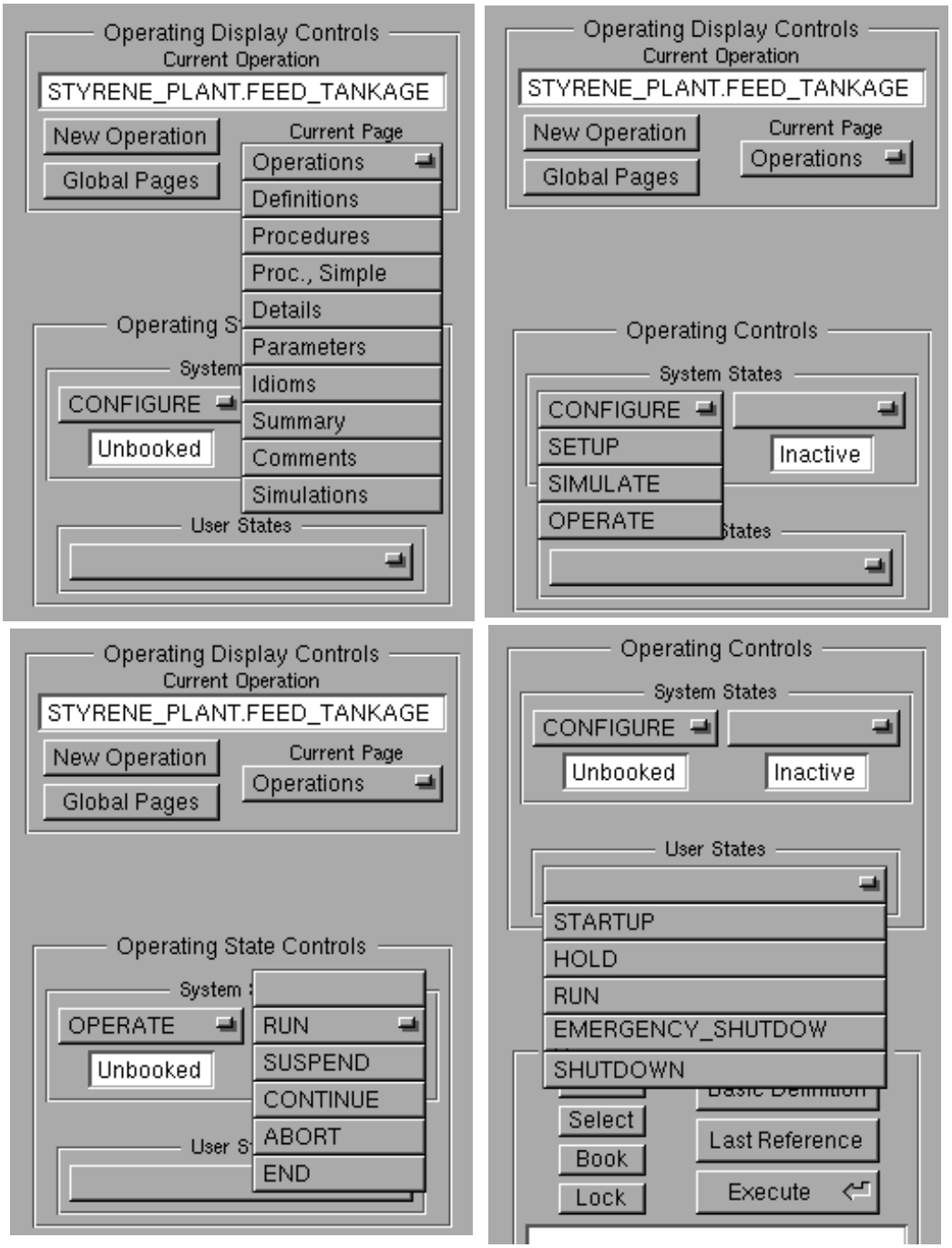
The second Loop Display emphasizes the Trend display. Each measurement has an indicated setpoint scale, to the

[106] Indicating the primary **T100AV** regulated in cascade, manipulating **F100**, itself regulated by manipulating **V100**, with a high constraint override on **F100**, when **P100** exceeds its setpoint

- Operating State Controls, for manually selecting system or user defined States, applied to the displayed Operation.
- Program and Monitoring Controls: to invoke a program Trace, to Select an Object from the listing, to Book an Object, to Lock an Object. These controls toggle on (back lit) and off. The controls also include push and return buttons to display the Basic Definition of a selected Object, to display the Last Reference (from a running program) to a selected Object, and to Execute a manually entered text command.
- Edit controls, for selecting a particular listing line, or a set of consecutive lines, to define a search pattern and to find that pattern in the listing, to Cut a line selection, or paste in its place, or insert before it, or add after it. It is assumed that the normal keyboard arrow keys allow moving to lower or higher lines on the listing, as well. It is also convenient to permit the normal delete key to delete a selection without interfering with the current paste buffer contents.

The edit controls select the line(s) to be edited, cut (delete and buffer for later pastes), pasted (replaced), inserted before, or added after. Thereafter any carriage returns define further lines to be included in the selected position (or changes in Bracket type or asterisk/diamond termination position). Changing an existing following line requires that it be selected explicitly by a Find action or by arrow key entry.

The figure below shows the action of the four pop-up menus:

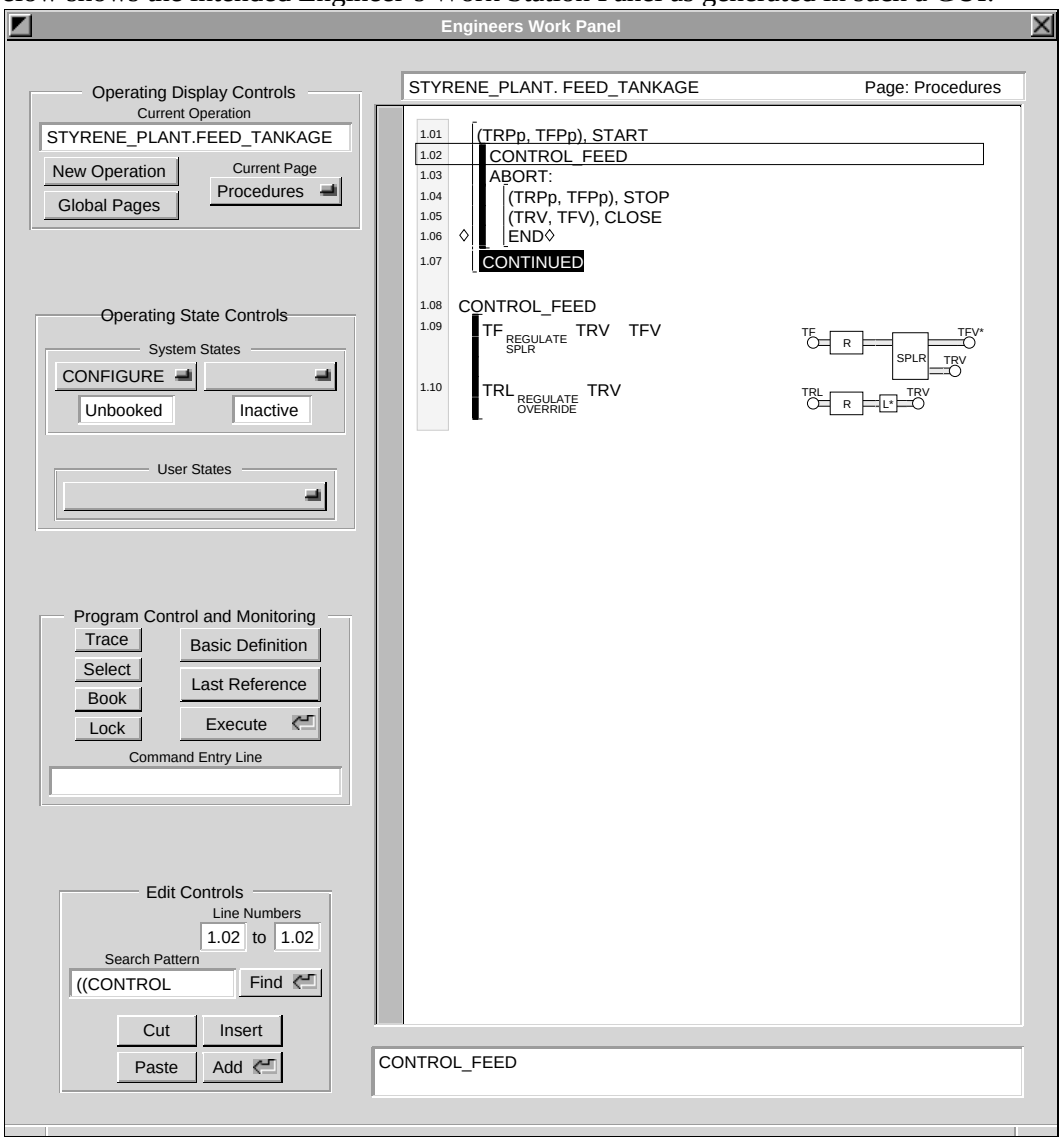


All but the lower left menu, are fixed, based on page names and system states. This last menu is derived from the user defined states.

Intended Engineer's Work Station Function

GUI Practicality Comment: Ideally the Engineer's Work Station (Panel) would allow general mouse-drag editing of the ICL program listing. However, the ICL listing goes beyond standard text formats, including several graphic elements (the diamond icons, the bracket icons, and the idiom loop icons) and multi-level subscripts. While mouse-drag editing of these elements can probably be programmed for our AP50 systems, using any chosen GUI and Display Postscript, it will require us to create, maintain, and support variant graphic objects, a technology with which we tend to have difficulty. Accordingly, the proposed engineers interface is built up out of buttons, pop-up lists, text entry lines, and a scrolled Display PostScript view, that are likely to be closely available in any reasonable GUI. The strategy uses the mouse to control buttons and to edit entry lines, and uses these to select the line to be edited in the listing. The line is is then displayed, in pure text form, on a text entry line, where it can be edited normally and then compiled to the listing display version on completion.

The figure below shows the intended Engineer's Work Station Panel as generated in such a GUI:



It incorporates a listing heading text line, a scrolled listing display area, and an edit area. The heading text line indicates the currently selected Operation and Page. The scrolled listing display area displays a listing of that Page, with internal page and line numbers added (for line selection), a box indicting the currently selected line or lines, and any reverse video-ing to indicate any currently active lines. The edit area below allows the display of the top most selected line, in pure text (Entry Format) form. This area allows mouse selection of any text for direct editing. As text is changed or entered, its corresponding Listing Format form is compiled and pretty-printed in the listing area. While terminal carriage returns are not supported with any representation in the edit area, their entry as part of the area will cycle the listing display appropriately. The top or bottom of a Bracketed Activity can be selected (as lines numbered with a following +(top) or -(bottom), and displayed in the edit area as (or) as appropriate. These can then be edited naturally.

The panel also includes four control areas:

- The Operating Display Controls, for selecting a new or old Operation to be displayed, and for selecting the Page to be displayed from that Operation (also echoed in the heading text line). The Controls assume a single set of global pages as well.

would include the calls for reading and writing dot referenced objects into standard Data Format form.

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

Multiple Output Control Idiom (MOC)

Conceptual Role and Purpose: To represent and support standard Multiple Output Control Practice: allowing a load to be shared between controlled flows of varying availability.

Detailed Role:

This Idiom ratios its output List operand elements further maintaining their ratio-ed summed value equal to an output value specified by the Idiom's control calculation, whatever happens to any one value. Its purpose is to allow a controlled application of the corresponding flow or energy so that this value is maintained constant even as contributing component flows are taken in or out of service. It is commonly applied in burner combinations.

Typical Formats:

Identical to Split Range Idiom.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

Operator Executed Commands

Conceptual Role and Purpose: To support essential engineering and operating functions, and provide the convenience of the full power of the language.

Detailed Role:

The language allows for full ICL statements to be executed from the engineer's or operator's work stations. This is the simplest vehicle for allowing operator execution of automated Recipes or other simple operating/maintenance tasks. These statements may be generated by direct engineer action or indirectly as generated by the system in response to operator interface actions. Such statements are considered to be temporary independent statements running on the Global Procedures Page. They will be visible there until completed. If they are intended to remain indefinitely they should be programmed into the same Page normally. Such a statement may have normal Details and Parameters Page consequence, which also disappear when the statement is completed. However Summary Page consequences of these temporary statements (e.g. Recipe Calls) can be preserved and filed.

A multi-statement Activity can also be entered temporarily. These are entered preceded by an **(**© (which does not by itself initiate a Sequential Activity), and assigned a user entered name so that they function as Tasks after their initial initiation. Such an Activity definition is terminated and Called by the entry of a final **)**© matching the initial **(**©. The name allows them to be terminated by user action.

The system must support system calls which permit ICL commands of the sort described above to be transmitted to it as text strings in Archival Format from external sources (including the operators console which may in fact generate the operator executed commands, possible translated from a quite different menu system). These system calls also

succeeding entries (one for each Backup Idiom) define successive alternative outputs, taken if the control to the first entry fails. The action uses the same strategy as the Split Range Idiom just defined but differs from it in that each output corresponds to a different set of controller tunings. The switching action is based on recognition of a significant difference between the controller output and its external feedback as passed to and from the variable currently being manipulated by that controller structure. The intended fan-out structure and algorithm is defined elsewhere. The Backup Idiom may include the process gain State symbol **+/–**.

Typical Formats:

Case 4 in the earlier figure.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

Blend Idiom

Conceptual Role and Purpose: To represent and support simple embedded blended controls.

Detailed Role:

This Idiom can occur as a Basic or Support Idiom. It allows a single input (with or with associated Basic Idiom) to be fanned out to ratio-ed right hand (List entry) operands, representing flow quantities to be blended. The Idiom supports pacing, that is the constraint of all outputs so that any constrained output will impose the analogous constraint on all other outputs maintaining the proper ratios, even if this requires a loss of control of the total flow.

Typical Formats:

Identical to Split Range Idiom.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

Combustion Control Idiom(CC)

Conceptual Role and Purpose: To represent and support the normal fuel/air control practice: avoiding unsafe excess fuel.

Detailed Role:

This Idiom is analogous to the Blend Idiom except that it ensures that the output Listed variables are further constrained so that any time the ratio fails (temporarily or permanently), the resulting ratios never take on unsafe values relative to each other. This is usually used in a fuel air ratio-ing control, to ensure that air always exceeds fuel in the proper ratio, never allowing excess fuel to accumulate as a hazard.

Typical Formats:

Identical to Split Range Idiom.

Keywords:

Archival/Listing Format Relationships:
The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:
Tasks and Activities are entered on the Parameters Page.

Decouple Idiom; DecoupleX

Conceptual Role and Purpose: To represent and support decoupled multivariable control.

Detailed Role:
This Idiom (with or without adaptation) represents a feedback decoupler based on our adaptive feedforward structure, to be used with the multivariable version of the Regulate Idiom.

Typical Formats:
Case 6 above.

Keywords:
Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:
Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:
The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:
Tasks and Activities are entered on the Parameters Page.

SplitRange Idiom (SPLR)

Conceptual Role and Purpose: To represent and support normal Split Range control, as best generalized.

Detailed Role:
This Idiom represents a fan-out of alternative outputs to be used if the selected output fails, as detected by a significant deviation of external feedback from intended output. The general fan-out structure and algorithm is defined elsewhere. It may be parameterized with an overlap such that several outputs are controlling, in part of their range, at the same time.

Typical Formats:
Cases 3 and 5 above.

Keywords:
Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:
Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:
The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:
Tasks and Activities are entered on the Parameters Page.

Backup Idiom

Conceptual Role and Purpose: To represent and support controls defined to back up controllers which have been blocked, for any reason, from controlling their variable.

Detailed Role:
This Idiom is an Main Idiom acting like a Support Idiom coming after a Basic Idiom except that it requires the right hand operand to be replaced by a List whose first entry defines the normal right operand of the Basic Idiom and whose

Typical Formats:

As shown in cases 1 and 10 above.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

FeedForwardX Idiom:

Tasks and Activities are entered on the Parameters Page.

Hi/Lo Limit Idiom

Conceptual Role and Purpose: To represent and support the simple in line limiting of controlled variables.

Detailed Role:

Defines an in-line High or Low Limiter on the output of the associated Main Idiom

Typical Formats:

Case 1 in the earlier figure.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

Hi/LoConstraint Idiom

Conceptual Role and Purpose: To represent and support limits on controlled variables imposed by the necessity to constrain other normally uncontrolled variables.

Detailed Role:

Defines a PID/selector based override of a normal loop, imposing a High or Low constraint relative to the left hand operand variable. The system derives the sense of the selector from the sense of the constraint and any process gain State symbols.

Typical Formats:

Cases 2 and 10 earlier.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

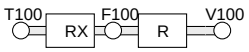
Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

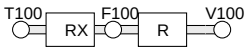
[105] A more complex notation can be defined to support general definition of arbitrary forward and backward computations. Similarly, an in-line nonlinear compensation Idiom could be defined with user specified forward and backward computations.

Case:

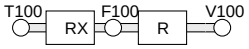
(1) TC2 $_{\text{REGULATEX}}$ RF2 $_{\text{REGULATE}}$ RV2



(2) T100 $_{\text{REGULATE X}}$ F100 $_{\text{REGULATE}}$ V100



(3) T100 $_{\text{REGULATE– X}}$ F100 $_{\text{REGULATE+}}$ V100



Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

RegulateX Idiom:

Tasks and Activities are entered on the Parameters Page.

Function Idiom; FunctionM

Conceptual Role and Purpose: To represent and support controls defined to back up controllers which have been blocked, for any reason, from controlling their variable.

Detailed Role:

This constitutes an in line breakpoint characterizing function applied to the output of its preceding Idiom, and supporting both forward compensation and backward self inverting external feedback computation. The FunctionM for is applied dually to the setpoint and measurement of the (immediately preceding) Regulate Idiom controlled variable, each also supported by back calculation.

Typical Formats:

Case 4 in the earlier figure.

Keywords:

Entered in this order:

Mandatory: **<<<OPERATION>>>**, **<<<END>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with built in character or the later declarations.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Parameters Page.

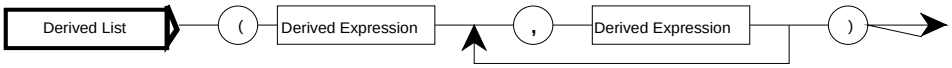
FeedForward Idiom; FeedForwardX

Conceptual Role and Purpose: To represent standard process control feedforward practice.

Detailed Role:

This corresponds to simple feedforward with or without adaptation. As shown in the above cases, the Idiom is followed by an argument variable indicating an additive feedforward variable, to which is applied a lead lag compensation with gain. If the argument is followed by a **–**, ***, */*, or **** the feedforward is instead subtractive, multiplicative, or pre- or post- divisive respectively.[105]

<Derived List>. The **<Derived List>** represents the normal multi-variable operand to a Derived Idiom operator (illustrated in Case 8 above).



Discussion; Meaning of the Loop Statement

The Idiom Loop or Idiom Loop Statement represents a progression or cascade of continuous controls, linking (one or more) primary variables to a control strategy represented by Idioms controlled by manipulating (one or more) secondary variables, in turn controlled by other Idioms manipulating tertiary variables, and so on.

As SuperVariable references or Lists are encountered immediately followed by an (Basic) Idiom (or list of Idioms), that Idiom is executed (or the listed Idioms in sequence are executed) applied to that SuperVariable (or List of SuperVariables). The result is generally stored in the next Basic SuperVariable or List of SuperVariables (or in a buffer, eventually passed to the SuperVariables). At the same time, the Idiom is pushed on a stack for later resumption in an appropriate back calculation, which will take the effective result of down stream control actions and pass them back into the controller state data, and occasionally into the original SuperVariable or List.

Basic Idioms generally process their associated SuperVariable or List of SuperVariables to arrive at a result which is passed to the (following) buffer or Basic SuperVariable. Their back calculation will generally be restricted to recalculating internal control states. But if the Basic Idiom is a fan-out Idiom they may recalculate the effective feed back value of their associated setpoint, reflecting any imposed limits.

Support Idioms generally take the value already generated by a Basic (or Override) Idiom and passed to a buffer (or manipulated Basic SuperVariable) and modify it (as in the case of a feedforward) or split it into values to be distributed among a List of SuperVariables (as in the case of a fan-out Idiom like a split range control). Their back calculation will general pass a modified buffered value back to the external feedback value or their associated Main Idiom, to modify its own back calculation.

Override Idioms generally process their associated SuperVariable or List of SuperVariables to arrive at a result which is compared to the (following) buffer or Basic SuperVariable. This comparison tests for any necessary override action. Override Idiom back calculation will generally be restricted to recalculating internal control states. But if the

In an Idiom List, acting as a Subject to an Idiom, each included SuperVariable is treated individually by the Idiom. If the List precedes the Idiom it is used in the calculation. If it follows the Idiom, it is used to store the result (or to supply data used in the back calculation). An Idiom Loop in an Idiom List provides its last SuperVariable result as the basis for following Idiom calculations. Its first SuperVariable receives data from preceding Idiom calculations.

The elements in an Override List provide data for their following Override Idioms analogously to the Basic List, and similarly receive data from preceding Override Idioms. The first such Override List after a Basic Idiom (like the first Basic List) has no Idiom driving it; its setpoint values are constant, or separately programmed.

Archival/Listing Format Relationships:

In the Listing Format consecutive Archival Format Idiom operators (indicated by preceding slashes) are regrouped vertically in a single subscript, unless there is an intervening (right) parenthesis. In that case, the Idioms following the parenthesis start a new, vertically grouped subscript.

Archival/Entry Format Relationships:

The Keyboard Entry format is the same as the Archival Format Idiom operator Keywords each being preceded by a ***/*** (slash, indicating subscript). On entry, the Listing Format immediately displays the subscripts as illustrated earlier.

Regulate Idiom; RegulateX, RegulateE

Conceptual Role and Purpose: To represent the basic regulatory use of controllers.

Detailed Role:

These are variants of the same basic regulation function. They occur as the first Idiom in a Step. and can occur in cascades. Each such Idiom may be followed (in its Idiom List) by a support Idiom, and (within the same Step) by Overrides. In the RegulateX and RegulateE Idioms two forms are allowed: the name may be used directly as a Basic Idiom: **REGULATEX**, **REGULATEE**. Or the ending (**X**, **E**) may be included on a separate line in the Idiom List as a Support Idiom, accomplishing the same thing (Case 2 below). Regulate corresponds to simple P, PI, or PID control. RegulateX corresponds to PID control with EXACT. RegulateE corresponds to PI or PID control with a breakpoint function on the error signal. Main Idioms like Regulate can be followed by a **+/–** State symbol indicating the process gain seen by the controller (Case 3 below). **+** is assumed otherwise; in either case the controller gain is set appropriately. A multivariable version takes right and left List operands and applies PID control one for one between each left and right List entry. The process gains are then assumed to be positive, or defined by a State List (e.g. **+,+,–**).

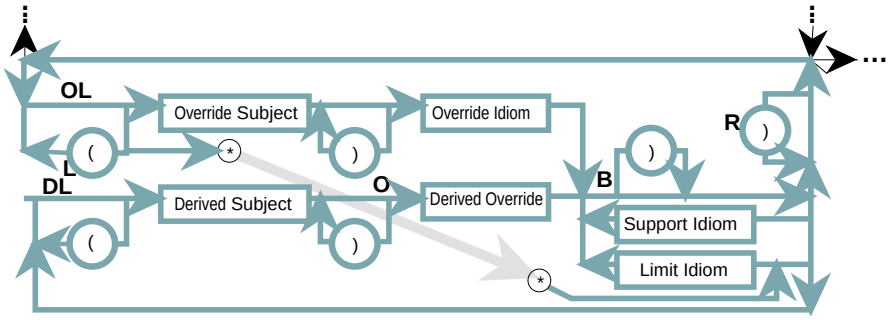
Typical Formats:

single variable defining a control condition which must cause the override of the basic Loop degrees of freedom. In the Loop statement each such override acts independently to override (unlike the Basic Subject/Idiom pairs which act in cascade). There are (apart from the multivariable constraint forms addressed with Override Lists) several more general situations (not now employed in practice, but necessary for the general form):

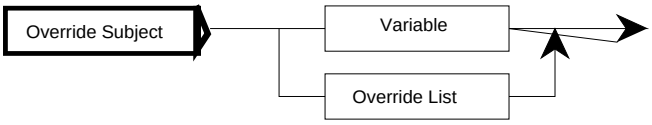
- Cascaded overrides deriving their constraints (imposed on one variable) from a more basic constrained variable. These are supported when the general Override Loop below occurs directly within an Idiom Loop.
- Overrides may also override the control constraint targets derived in an existing Override Loop. These are supported when an Override Loop is nested, parenthesized, within an outer Override Loop (supported by the following **R/L** parenthesis rule).

In the Rule Diagram below, a Override Loop is identified, analogously to an Idiom Loop as a statement or expression whenever the diagram passes initially through point **OL**, passing as many times through point **R** as point **L**, matching its right and left parentheses, and passes finally through point **O**.

An analogous expression, applicable to continue override cascades, is the Derived Loop. The expression starts passing through point **DL**, then through **B**, and, as before, matching passes through **L** with passes through **R** and all left and right parentheses, completing by passing through **O**.

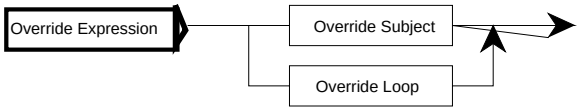


<Override Subject> ::= <Variable> | <Override List>. The **<Override Subject>** represents the normal general form of the operand to a Override Idiom operator in the Loop Statement: the first variable or list of variables in each of the above cases, including individual Variables and Lists for multi-variable override action:

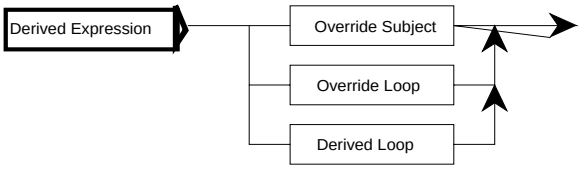


<Derived Subject> ::= <Variable> | <Derived List> (like Override Subject, replacing Override Lists with Derived Lists). The **<Derived Subject>** represents the normal general form of the operand to the Derived Override Idiom operator. Includes expressions that can continue an existing Override.

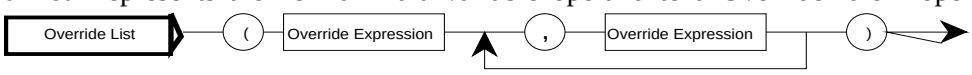
<Override Expression> ::= <Override Subject> | <Override Loop>. The **<Override Expression>** represents the general form of the entries in Override List operands. These include individual Variables (from the Override Subject), Override Lists (which themselves may include Override Expressions), and Override Loops which may express their own internally controlled combination of variables.



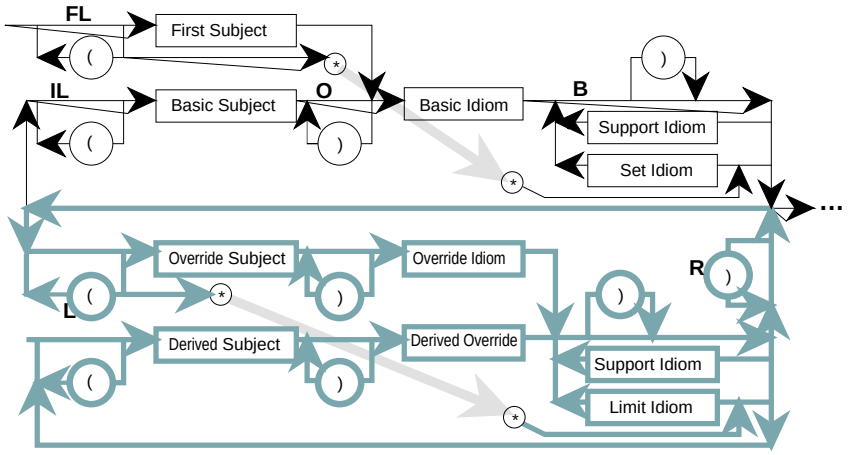
<Derived Expression> ::= <Derived Subject> | <Override Loop> | <Derived Loop>. The **<Derived Expression>** represents the general form of the entries in Derived List operands, like the Override Expression but based on Derived Subjects and able to include Derived Loops.



<Override List>. The **<Override List>** represents the normal multi-variable operand to a Override Idiom operator (illustrated in Case 9 above).

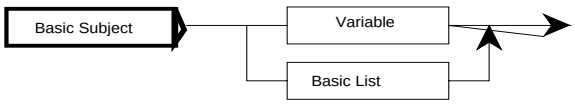


development. The expression, starting in a larger statement or expression whenever the Rule Diagram below passes point **FL** and completed when the last matching right parenthesis has been encountered and the diagram reaches point **O**, is an Idiom Loop, meets the definition of the above structure diagram. The similar expression, starting whenever the diagram passes point **IL** and and then passes point **B**, completed when the last matching right parenthesis has been encountered and the diagram reaches point **O**, is an Idiom Loop, meets the definition of the above Rule Diagram. The advantage of embedding definitions this way is that the dependence of the definition on the containing structures is made explicit.



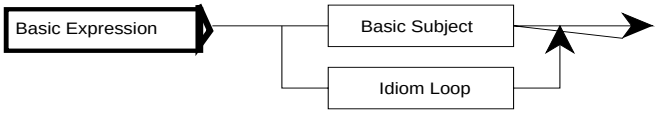
Component Definitions:

<Basic Subject> ::= <Variable> | <Basic List>. The **<Basic Subject>** represents the normal general form of the operand to a Basic Idiom operator in the Loop Statement: the first variable or list of variables (for multi-variable control) in each of the above cases. Note that the Basic and Override Lists in a Loop statement may require matching in element numbers. This requirement will be defined in the associated Idiom (system or Task) definitions.



<First Subject> ::= <Variable> | <First List>. The **<First Subject>** is like the Basic Subject, but based on First Lists (Lists whose Loops can further include Set Idioms) also.

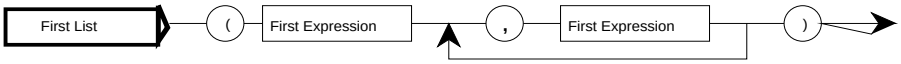
<Basic Expression> ::= <Basic Subject> | <Idiom Loop>. The **<Basic Expression>** represents the general form of the entries in Basic List operands (Cases 5–7). These include individual Variables (from the Basic Subject), nested Basic Lists (also from the Basic Subject, which themselves may include Basic Expressions), and Idiom Loops which may express their own internally controlled combination of variables.



<First Expression> ::= <First Subject> | <Idiom Loop>| <First Loop>. The **<First Expression>** is like the Basic Expression except that it is based on First Subjects, and can also contain First Loops (including Loops with Set Idioms).

<Basic List>. The **<Basic List>** represents the normal multi-variable operand to a Basic Idiom operator (illustrated in Cases 5–7 above).

<First List>. The **<First List>** is like the Basic List but based on First Expressions:



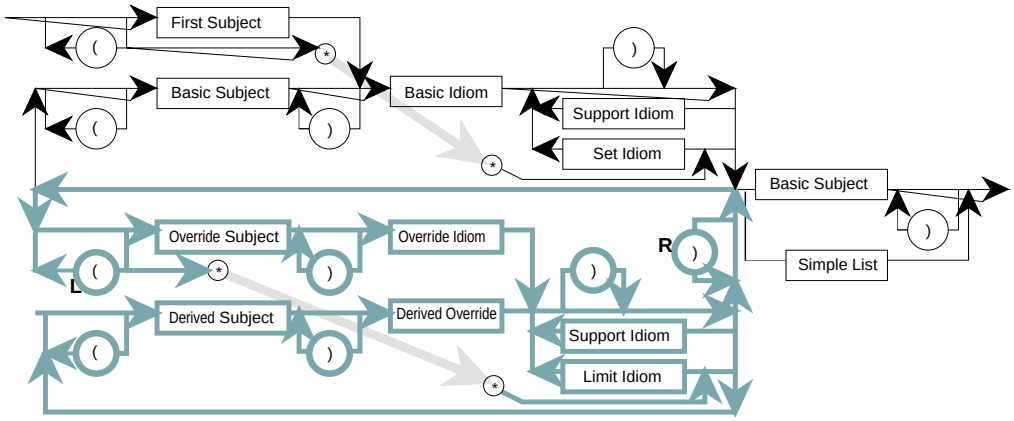
<Simple List> ::= <Basic Expression> <Basic Expression> | <Basic Expression> <Simple List>. The **<Simple List>** represents an alternative form of the List operand, applicable only to the final Subject in a Loop Statement (illustrated in Cases 3 and 4 above). Its main purpose is to permit the simple inclusion of fan out Idioms (Split Range, Blend, MOC, etc.) in single loop control statements.

<Override>, **<Override Loop>**, **<Derived Loop>**. Earlier examples show the simplest case of an Override (Case 2) a

<Derived Subject>, and **<Simple List>**. These are all variants of the sam Idiom Subject, each variant with different restrictions.[103] The Basic Subject generalizes the Variable to include multi-variable combinations of Variable and controlled cascade loops. The First Subject is a variant first Subject whose cascaded loops can include Set Idioms. The Override Subject corresponds to a First Subject for the Override Loops; the first Subject in an Override Loop. The Derived Subject corresponds later Subjects in the Override Loop. The terminal Simple List (**TRV** etc. in Case 3 and 4) simplifies the expression of terminal fan-outs. In addition to the normal Rule Diagram constraint with its Parenthesis Rule, two other parenthesis constraints are imposed:

- an expression starting with a left parenthesis on the gray path must stay on the gray paths until the matching right parenthesis has been encountered.[104]
- every left parenthesis encountered at point **L** in the figure is matched by a right parenthesis encountered at point **R**.

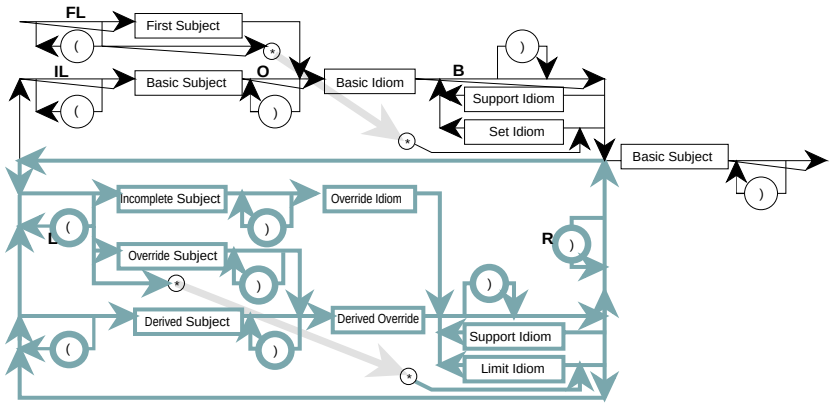
Note: The Basic and Override Subjects may not be disambiguated until the Basic or Override Idioms are encountered.



The six (Basic, Set, Override, Limit, Derived Override, and Support) forms of Idiom operator are subscripted (in the Listing format) or preceded by an included / (in the Archival format). The diagram enforces a consistency between Basic or Override Idioms and corresponding Subjects. It requires that all Support Idioms follow a single Basic or Override Idiom in the same subscript. It requires that left parentheses occur only directly preceding a (Basic or Override) Subject (or, as indicated earlier, Set and Limit Idioms). It allows right parentheses to follow a Subject or an Idiom (operator). These may be redundant, or used to group expressions to share Support Idioms as in Case 10.

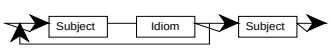
The diagram and rules require that a parenthetical expression starting with Override Subject and Idiom must continue without any Basic Subject or Idiom until the expression is completed with a matched parenthesis. The Simple List is forced to be the last element, when used. Thicker, fainter gray arrows show the paths which permit initial Set and Limit Idioms.

<First Loop>, **<Idiom Loop>**. A simplified form of the Statement, eliminating the Simple List, used recursively inside the definitions of the First and Basic Expression, List, and Subject, is shown in the Rule Diagram below below. It represents the normal cascade wherever it may occur, but it can include override structures. The structure, starting in a statement or expression whenever the diagram enters at input point **FL** and completed at the output is an First Loop. If it starts at input point **IL** and completes at the output, it is an Idiom Loop.



This form is also used, within the Loop and Loop Statement, as part of the visualization of their recursive

[103] Otherwise the whole diagram could be summarized as shown here:



[104] Only Override Subjects, or Override, Derived Override, or associated Secondary Idiom operators may be encountered.

- Different substructures within an Idiom Loop Statement have special meaning:
  - Cascaded repetitions of the basic pair of variable reference or reference List and Basic Idiom based Idiom list indicate cascaded control following the normal control degree of freedom.
  - Insertions of a pair consisting of a variable reference or reference List and Override Idiom (or a single Limit Idiom), by itself, or when continued by Derived Override Idioms (with their Lists or variables, and Idiom lists), represent an override intervention in the normal control degree of freedom. Thus Override Idioms must not occur as the first Idiom in a Loop.
  - Parentheses permit grouping of Idiom pairs and sharing of Support or other Idioms.
- Idiom Loop Statements have names (as default, the name of their first variable) and States. The default name can be overridden by a Naming Prefix. The States are **_/INITIALIZE, AUTO/States bearing names of all the statement variable names, NORMAL/OVERRIDDEN/DISABLED**.
- Individual Idioms are also similarly named.

Typical Formats:

The above cases represent typical Loop Statements. The corresponding Archival Formats are:

Case	Archival Format
(1)	**TC2 /REGULATE /FEEDFWD SF /FEEDFWD TP2 /LOLIMIT RF2 /REGULATE RV2**
(2)	**BC1 /REGULATE DP1 /HICONSTR DP2 /LOCONSTR OF1 /REGULATE OV1**
(3)	**TF1 /REGULATE /SPLR TRV TQV TFV**
(4)	**TF /REGULATE /BACKUP TRV TFV**
(5)	**TF /REGULATE /SPLR (TRV, TQV, TFV)**
(6)	**(TRF, TQF, TF) /REGULATE /DECOUPLE (TRV, TQV, TV)**
(7)	**(T100, T200, T300) /REGULATE (V100, F200 /REGULATE V200, V300)**
(8)	**(T100, T200) /REGULATE (DP1, DP2)/HICONSTR (T300, T400/DRCONSTR T500) /DRCONSTR (V100, V200)**
(9)	**(T100, T200) /REGULATE (DP1/HICONSTR F100, F200)/REGULATE (V100, V200)**
(10)	**(T100/REGULATE/FEEDFWD SF DP1/HICONSTR/FEEDFWD TP2)/FEEDFWD SF2 V100**

Keywords:

The only difference between the Listing and Archival Formats is that the Archival Format uses the slash (/) to indicate an Idiom operator which is to included within the Listing Format Idiom List subscript.

Discussion; Loop Statement Rule Diagram/Bachus Naur Definition:

The general Loop Statement Archival Format syntax is best defined in a combination of Bachus Naur forms, Structure Diagrams, and Rule Diagrams, and in terms of the primitives: **<Variable>**, **<Basic Idiom>**, **<Set Idiom>**, **<Override Idiom>**, **<Derived Override>**, **<Limit Idiom>**, and **<Support Idiom>**, whose members are defined in their appropriate sections. A **<Variable>** represents any ICL SuperVariable reference within an appropriately structured SuperVariable. The normal operation of the Zipper Reference must be able to return a value of the appropriate type (normally Real). The normal operation for the Zipper Reference on the reference expression with **.SET** added to it must return a corresponding setpoint of appropriate type. The Loop Statement defines a (single or multi-) degree of freedom control structure in which the Basic, Set, Override, and Limit (and Derived Override) Idioms generally define the degree of freedom paths and the Support Idioms define supporting calculations.

Support Idioms may include variable references (e.g. the feed forward Idioms in Cases 1 and 10 above), representing coordinating data not affecting the degree of freedom path. Basic Idioms represent the normal degree of freedom control, whereas Override Idioms represent overriding degree of freedom paths necessary if a constraint is being violated. The definitions support the use of parenthesized Lists of variables and controls to indicate multivariable controls of varying levels of complexity (Cases 5–9). But the basic loop definitions are themselves arranged to permit parentheses about Idiom operands on the left, and Idiom operators or operands on the right (Case 10). Each structure must conform to the Parenthesis Rule.

The definitions are more complex, in the general case, than the analogous Assignment (Infix) Statement forms because they must differentiate between the behavior of the five basic kinds of Idiom Operators, and in so doing, allow right parentheses to enclose both operands and operators. Idiom Loop Statements without parentheses are much simpler.

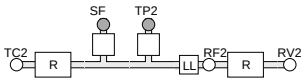
Basic (and Set) Idioms combine in cascades (Case 1) in which successive Variables pair with their associated subscripted Basic Idioms (and optional Support Idioms) to form a repeating pattern, representing the basic degree of freedom path of the cascade loop.

Override (and Limit) Idioms will normally represent interventions in that path; the similar repeated pattern represents the different behavior, the intervention. However, as a matter of generalization, Override cascades can also be defined using the Derived Overrides.

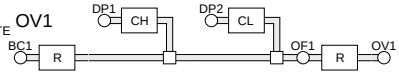
<Idiom Loop Statement>. The Idiom Loop Statement is defined in the following Rule Diagram, for which the additional basic components are defined in detail later: **<First Subject>**, **<Basic Subject>**, **<Override Subject>**,

Case:

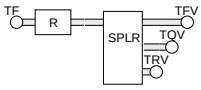
(1) TC2 $_{\substack{\text{REGULATE}\\\text{FEEDFWD SF}\\\text{FEEDFWD TP2}\\\text{LOLIMIT}}}$ RF2 $_{\text{REGULATE}}$ RV2



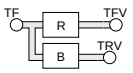
(2) BC1 $_{\text{REGULATE}}$ DP1 $_{\text{HICONSTR}}$ DP2 $_{\text{LOCONSTR}}$ OF1 $_{\text{REGULATE}}$ OV1



(3) TF $_{\substack{\text{REGULATE}\\\text{SPLR}}}$ TRV TQV TFV

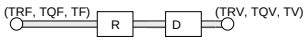


(4) TF $_{\substack{\text{REGULATE}\\\text{BACKUP}}}$ TRV TFV



(5) TF $_{\substack{\text{REGULATE}\\\text{SPLR}}}$ (TRV, TQV, TFV)

(6) (TRF, TQF, TF) $_{\substack{\text{REGULATE}\\\text{DECOUPLE}}}$ (TRV, TQV, TV)

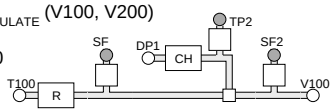


(7) (T100, T200, T300) $_{\text{REGULATE}}$ (V100, F200 $_{\text{REGULATE}}$ V200, V300)

(8) (T100, T200) $_{\text{REGULATE}}$ (DP1, DP2) $_{\text{HICONSTR}}$ (T300, T400 $_{\text{DRCONSTR}}$ T500) $_{\text{DRCONSTR}}$ (V100, V200)

(9) (T100, T200) $_{\text{REGULATE}}$ (DP1 $_{\text{HICONSTR}}$ F100, F200) $_{\text{REGULATE}}$ (V100, V200)

(10) (T100 $_{\substack{\text{REGULATE}\\\text{FEEDFWD SF}}}$ DP1 $_{\substack{\text{HICONSTR}\\\text{FEEDFWD TP2}}}$) $_{\text{FEEDFWD SF2}}$ V100



The further Cases 7–10, show:

7. The use of Lists to generalize to multivariable loops, with a single variable (it could be lower dimensional multivariable) Loop embedded in the output List;
8. The handling of constraints in multivariable loops, with cascaded multivariable, derived constraint continuations and a further single variable derived constraint override continuation embedded in the third List;
9. Multivariable loops with an embedded Listed constraint affecting only one loop;
10. Parentheses used to distinguish the application of Support (feedforward) Idioms on the main cascade, on the override, and applied to the result of both.

In these cases, each variable (or List of variables) with its immediately following Idioms represents a controlled (or constrained) variable (or List) with the sequence of control intentions to be supported in its control. The resulting control actions define control targets for a following variable (or List) which may itself require control. The language implementation supports the proper association of these variables and may involve forward and backward calculations on the expression.

Two other Idiom classes allow the direct manipulation of a first variable in a loop, without feedback control: Set Idioms in main loops (in Manual Station like roles), and Limit Idioms in constraint override loops. These also can function like Support Idioms (the Limit Idiom in Case 1).

The basic rules of syntax can be summarized as follows (before detailing below):

- An Idiom Loop Statement (legal or illegal) is recognized as an initial variable (SuperVariable) reference or List, followed by one or more subscripted (indicated in archival form below by a preceding /) Idiom operator (in an Idiom list), including optional parentheses.
- Alternatively, an Idiom Loop Statement (legal or illegal) may start with a Set Idiom with or without a continuing Idiom list of Support Idioms.
- A legal Idiom Loop Statement consists of alternating variable references or Lists and Idiom operator lists with constraints on:
  - Use of parentheses: They must be appropriately matched. Left parentheses can only occur before references or reference Lists[101], or as part of the Lists. Right parentheses occur after reference Lists, references, or Idiom operators, or as part of the Lists.
  - The Idiom lists: They must include only one Basic or Override Idiom, located at the start of the list, followed (below it in the subscript) by any number of Support, Set, of Limit Idioms. But the first Idiom list in an Idiom Loop Statement may start instead with a Set Idiom if there is no preceding variable, and the first Idiom list in an Override Loop may start with a Limit Idiom if there is no preceding variable.[102]
  - The reference Lists: They consist of appropriate variables or internal control loops.
  - The variable references: They must be able to refer directly to a value of the appropriate data type (usually Real), and must, except in the case of output variables from the Loop, have a directly associated setpoint (referenced by adding a **.SET** to the variable reference).

[101] With this exception: a left parenthesis may be followed by a Set Idiom, as the first Idiom in an Idiom Loop Statement or First Loop, or by a Limit Idiom, as the first Idiom in an Override Loop.

[102] Note that a (right) parenthesis may fall in the middle of an Idiom list, in which case, the list is split in to, starting again after the parenthesis.

Profile Statements

Conceptual Role and Purpose: To support the operational coordination of a set of higher level variables.

Detailed Role:

These statements are concerned with presenting a coordinated operation of a set of distinct and independently controlled but economically related controlled variables.

Typical Formats:

A

Keywords:

Entered in this order:

Mandatory: **<<<X>>>**.

Optional: **<<<X>>>**.

Redundant: **<<<X>>>**.

Discussion:

A

Archival/Listing Format Relationships:

A

Archival/Entry Format Relationships:

A

Idioms and Idiom Loop Statements, Including User Written Idioms

Conceptual Role and Purpose: To simply represent standard process control loop cascading practice, in its most general form.

Detailed Role:

Idioms (Idiom calls), discussed earlier, are embedded in Idiom Loop statements. In process control, a loop has come to include all control functions needed to support a basic primary control loop. Thus rather than being limited to a single controller and its measurement and valve, it can include the secondary (and tertiary etc.) controls; and the feedforwards, selector overrides, decouplers, etc. The basic Idiom Loop statement format for representing these combinations follows an alternation of variable name and Idiom operator: **<Variable Name>**$_{\text{<Idiom Operator>}}$ **<Variable Name>**$_{\text{<Idiom Operator>}}$**<Variable Name>**; for example **T100**$_{\text{REGULATE}}$ **F100**$_{\text{REGULATE}}$ **V100**. In this simple case, the statement represents a simple cascading of control, control in a single degree of freedom. This will be the common, most used case. However the specification includes the most general cases as well: the variables in the statement can be replaced by Lists, in various ways, to define multi-variable controls:

(T100, T200, T300)$_{\text{REGULATE}}$ **(F100, F200, F300)**$_{\text{REGULATE}}$ **(V100, V200, V300)**

More generally, feedback control Idioms are classified as Basic, Override, Derived Override[100], and Support Idioms. Basic Idioms define the basic control degrees of freedom (for example the Regulate Idioms in the examples below). Case 1 also illustrates Support Idioms; Idioms listed below an associated Basic or Override Idiom to qualify its effect. Further, the statement supports the representation of different kinds of Constraint Override controls (expressed as Override Idioms; the Hi and Lo constraint Idioms in Case 2) and fan-out controls (which may be expressed as Basic or Support Idioms; Cases 3 and 4). The fan-out controls can be equivalent to the List based multi-variable controls (Cases 3 and 5). Idioms may also require multi-variable treatment (like the Regulate and Decoupling Idioms in Case 6). Each Idiom Loop Statement has an iconic equivalent in which the Variables (or Lists) are indicated by named circles or nodes and the Idioms are shown by icons displayed, left to right, in the same order as the text words.

[100] These first three forms are also referred to, collectively as Main Idioms.

Entered in this order:
Mandatory: **<<<X>>>**.
Optional: **<<<X>>>**.
Redundant: **<<<X>>>**.

Discussion:
A
Archival/Listing Format Relationships:
A
Archival/Entry Format Relationships:
A

Stream Statements
Conceptual Role and Purpose: To represent coordinated conveyor production controls.
Detailed Role:
These statements are also based on a Truth Table similar to the Sequencing/Timing statements. They are concerned with coordinating conveyor activities, using a generalized shift register to coordinate production activities taking place at a number of stations.

The stream statement allows the coordination of production operations occurring in different locations in the plant. The statement is designed to allow the product in successive segments of the production line or conveyor to be assigned different quality states (**GOOD/REWORK/REJECT** below). As the product moves to further stations processing actions can be coordinated with this quality state.

The statement is coordinated by a station counting variable (C100 below), and a quality variable (C101 below with its own quality states (**NORMAL/LOW/BAD**). The production stations are indicated by named conveyor locations each recognized by a count or timing into the run (**LOAD(0)/ CHECK(150)/ REJECT(180)/ END(300)** below). The computational function of the stream statement models the production as a generalized shift register.

The statement causes the time history of changes in quality history to be stored against counts in the station counting variable. As product is estimated to arrive at the successive stations (represented by their delay in counts) the corresponding recorded quality states are acted on according to the lines in the modified truth table. In this way, the production quality at one station and time can be passed to other stations and acted on at their corresponding times.

STREAM C100
QUALITY: GOOD/ REWORK/ REJECT
STATION: LOAD(0)/ CHECK(150)/ REJECT(180)/ END(300)

STATION	QUALITY	C101	:ACTION
CHECK		LOW	: REWORK
CHECK		BAD	: REJECT
REJECT	REWORK		: C102, RECYCLE
REJECT	REJECT		: C102, REJECT

The statement form can be extended, replacing the counted state by a real value (like a flow). In this case the product flow is integrated instead of counted. The station positions are then represented by the net (integrated) product passed on the line until the station is reached.

Typical Formats:
A
Keywords:
Entered in this order:
Mandatory: **<<<X>>>**.
Optional: **<<<X>>>**.
Redundant: **<<<X>>>**.

Discussion:
A
Archival/Listing Format Relationships:
A
Archival/Entry Format Relationships:
A

then be optimized to reflect this basic coordination role.

Typical Formats:

A

Keywords:

Entered in this order:

Mandatory: **<<<X>>>.**

Optional: **<<<X>>>.**

Redundant: **<<<X>>>.**

Discussion:

A

Archival/Listing Format Relationships:

A

Archival/Entry Format Relationships:

A

Sequencing/Timing Statements

Conceptual Role and Purpose: To represent simple event conditioned and timed sequencing.

Detailed Role:

These statements are implemented as a special from of Truth Table, designed to represent drum sequencing and timer forms of control. The statement is initiated with a Sequence Header:

SEQUENCE: START/RUN/STOP/STOPPED or **SEQUENCE: START(10 SEC)/RUN/STOP(10 SEC)/STOPPED**

This consists of the keyword **SEQUENCE** followed by the consecutive user defined Phase States intended to represent the normal sequence of State processing. The timing function is used when a state is followed by a parenthesized timing value (as above). The Header is followed by a Truth Table, which may include any desired initial State column headings, but which will also include final Phase, Delay (if any Phase State has an associated parenthesized timing value), and Action. The Action column includes a continuation statement to be executed if all of the other column entries apply:

SEQUENCE: START/ RUN/ STOP/ STOPPED

POWER	SPIN	PHASE	:ACTION
		START	: "STARTING"; STARTER, START; HOLD, HOLD
POWERED	TURNING	START	: "RUNNING; STARTER, _; RUN
		STOP	: "STOPPING"; HOLD, STOP; ALARM, _
STOPPED	STOPPED	STOP	: "STOPPED"; HOLD, HOLD; STOPPED
POWERED	TURNING	RUN	: "RUNNING"
STOPPED	TURNING	STOP	: "ALARM"; ALARM, ALARM
		STOP	: "ALARM"; ALARM, ALARM

or:

SEQUENCE: START(10 SEC) / RUN/ STOP(10 SEC) / STOPPED

POWER	SPIN	PHASE	DELAY	:ACTION
		START	EARLY	: "STARTING"; STARTER, START; HOLD, HOLD
POWERED	TURNING	START	EARLY	: "RUNNING; STARTER, _; RUN
		STOP	EARLY	: "STOPPING"; HOLD, STOP; ALARM, _
STOPPED	STOPPED	STOP	EARLY	: "STOPPED"; HOLD, HOLD; STOPPED
POWERED	TURNING	RUN		: "RUNNING"
STOPPED	TURNING	STOP		: "ALARM"; ALARM, ALARM
		START	LATE	: "ALARM"; ALARM, ALARM
		STOP	LATE	: "ALARM"; ALARM, ALARM

The special command **NEXTSTATE** within the action continuation statement will cause the Phase State to be advanced one State, as defined in the Sequence Header. The Delay State Column can include any of the three standard Time value States: **EARLY/TIME/LATE.**

Typical Formats:

A

Keywords:

RAMP Page: Procedures

```
RAMP TEMP:*[1]      START FOR VENTTIME;*[2]
                    HEAT TO COOKTEMP + BIAS IN HEATTIME MIN;*[3]
                    COOK AT COOKTEMP*[4] FOR COOKTIME;*[5]
          RAMP P100: COOL TO 0*[6] IN COOLTIME*[7]
```

RAMP Page: Details

*[1] VENT, OPEN;
STEAMVALVE, OPEN;

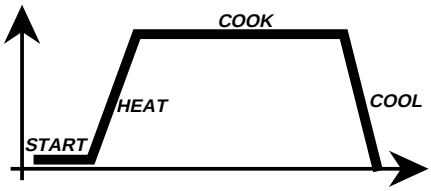
*[2] VENT, CLOSED;
STEAMVALVE, CLOSED;
"RECORD START BULB TEMPERATURE"LOG; BULBTEMP?LOG; TEMP LOG

*[3] "RECORD START BULB TEMPERATURE"LOG; BULBTEMP?LOG; TEMP LOG
BIAS = 0.0
PRODTEMP = TEMP

*[4] PRODTEMP = LOSELECT(PRODTEMP, TEMP)

*[5] TEMP LOG
STARTPRESSURE = PRESSURE.SET

The Ramp Statement allows programming of a time profile, in which, for each segment, the initial value (or a continuation from the last value), the final value (or a constant value), the ramping (or holding) time may be optionally specified. The Statement allows a change in the variable being profiled (the **RAMP P100** above). The statement allows the optional naming of each segment by an associated State name (***START***, ***HEAT***, ***COOK***, ***COOL*** above) for the purpose of supporting State computations and operator display and control.



The statement also allows the coordination of separate Footnoted activities with different positions in the profile, to occur either at particular points in time (Footnotes 1, 2, 3, 5, 7 above), marked at the beginning or end of the segment; or over the entire duration of a segment (Footnotes 4 and 6 above), marked within the segment text. The Footnotes are positioned in the statement text as asterisks (the system provides the associated bracketed numbering. The actual Footnotes are filled in on the associated Details Page.[99]

Typical Formats:

A

Keywords:

Entered in this order:

Mandatory: **<<<X>>>**.

Optional: **<<<X>>>**.

Redundant: **<<<X>>>**.

Discussion:

A

Archival/Listing Format Relationships:

A

Archival/Entry Format Relationships:

A

Blend Statements

Conceptual Role and Purpose: To represent higher level operation of blending operations.

Detailed Role:

Blending can be expressed both as a Theme statement and as an Idiom. The former usage is appropriate when the Blending operation represents the operational focus of the corresponding process unit. The corresponding display can

[99] The Local Equipment Controller does not allow for separate (Details) Pages. The associated usage is to position the Footnotes (or other associated entries) (automatically) below their associated statements.

and structuring power of these statements:

- The statements are generally constructed of repeating clauses and phrases[98], which can be combined flexibly:
  RAMP T100: *START*FOR 10 MIN; *HEAT*TO 200® IN 20 MIN; *COOK*AT 210® FOR 120 MIN; RAMP P100 *COOL*TO 0 PSIG IN 40 MIN
- The regimes of the different clauses can be assigned State names, as shown above with the superscripted italicized Start, Heat, Cook, Cool State names.
- Multiple Footnote points within the statement allow an open ended set of associated activities to be coordinated against the basic (repeating) sequencing pattern.
- Certain Theme statements address more complicated State computations. In this case, the statement behaves like a Local State Environment Declaration statement and can be followed by State Prefixed statements defining activities coordinated by state.

Typical Formats:

A

Keywords:

Entered in this order:

Mandatory: **<<<X>>>**.

Optional: **<<<X>>>**.

Redundant: **<<<X>>>**.

Discussion:

A

Archival/Listing Format Relationships:

A

Archival/Entry Format Relationships:

A

Wait Statements

Conceptual Role and Purpose: To represent timed and event conditioned waiting.

Detailed Role:

There is always a need for simple timing. This statement carries that out as in: **WAIT 2.3 MIN**. The Wait can also be conditioned on some state or test as in **WAIT HOT** or (with a Local State Environment Declaration List) **[T100, 100] WAIT HI**. States requiring Local State Environment Declaration keep that Declaration active for the duration of that WAIT statement. The command can use all of the appropriate State Prefix keywords (**ANY**, **SOME**, **ANY_NOT**, **ALL**, **NONE**) as in **[Reactor, Charge] WAIT ALL UNBOOK**.

Typical Formats:

A

Keywords:

Entered in this order:

Mandatory: **<<<X>>>**.

Optional: **<<<X>>>**.

Redundant: **<<<X>>>**.

Discussion:

A

Archival/Listing Format Relationships:

A

Archival/Entry Format Relationships:

A

Ramp Statements

Conceptual Role and Purpose: To represent the programmed control of analog or real variables to follow a continuous profile.

Detailed Role:

Ramp statements are named, taking the name of their main variable as the default. This statement carries out the programmed equivalent of a cut cam follower, being able to express a sequence of ramping actions applied to any Real valued variable. At the same time, associated actions can be coordinated with the ramping through States and Footnotes. For example:

[98] The colons and semicolons are included for clarity when the statement is listed on a line. They are distinct from the Prefix colons and the statement separating semicolons, occurring within the statement. Normally the statement will be formatted on several lines eliminating their need.

The Local State Environment Declaration usage allows the State Prefix States to be more generally related to the States normally returned in States. When this is done, the Details Page will show a Truth Table, filled in by the user, to define the intended relation between the built-in States and the State Prefix States (see earlier Details Page discussion).

Certain other system functions also generate States in **Result**, which can then be tested by State Prefixes. For example the functions **HISELECT** and **LOSELECT** can return Real values in an assignment, indicating the highest or lowest of their arguments (any number). At the same time, through **Result**, the selected argument is returned as a State name, testable in a State Prefix. The example shows a combination of an initial Local State Environment Declaration List returning a **+/–** State, followed by State Prefixed statements involving the selector functions. The chosen function then returns states **Set/DSet**, followed by their own State Prefixes.

```
[Sense];
  + :      Set = LoSelect(Set, DSet);
  ELSE :   Set = HiSelect(Set, DSet);
  Set :    LasherBits, Free;
  ELSE :   LasherBits, Blocked
```

Certain conventional functions naturally associated with hardware (e.g. assignments to I/O variables) may give rise to a failed condition, with associated **_/FAILED** States, passed in Result, and testable in state Prefixes. In conjunction with the **FAIL** State the **RETRY** function can be used to call for a retry of the failed command, if the State returned is **FAIL** (with no action if the State is _): ◊|||RETRY; RETRY; FAIL:: END◊ (in this case with a maximum of two retries). Also certain shorthand variants of the Local State Environment Declaration List allow it to be combined with a State Prefix:

```
  [TIME] TIME/LATE :   COUNT, RESTART; TIME, RESET
◊ [COUNT] HIGH :       END◊
◊ [F200.SET < 6000] :  END◊
  [V200; M200];
◊  MAN/ID_MAN :        END◊
```

The top two examples show the simplest case with the Declaration List and Prefix combined in a single statement. The third example defines a comparison and the consequence of its success. In this simplest case it can be viewed as the equivalent of the expression **[F200.SET, 6000] LO: END**. But the expression can combine and indefinite number of comparisons: **[a<b<c]: ...**. The final case shows the normal combination of Declaration and Prefix on separate lines with semicolon separating the Declaration items.

Typical Formats:

A

Keywords:

Entered in this order:

Mandatory: **<<<X>>>**.

Optional: **<<<X>>>**.

Redundant: **<<<X>>>**.

Discussion:

LOSELECT, **HISELECT**

Archival/Listing Format Relationships:

A

Archival/Entry Format Relationships:

A

Theme Statements

Conceptual Role and Purpose: To represent standardized higher level sequencing and coordination activities.

Detailed Role:

The Idioms provide the basis for a concise way of specifying certain strategic continuous control activities. Batch control can benefit from the highlighted specification of key functional modules. Certain kinds of sequencing call for a basic repeated pattern of activity that can structure the control system. Some of these have been expressed historically with their own special hardware: timers, drum sequencers, cut cam followers, machine controllers. In a language, these kinds of activities can be expressed in templates supporting the entry and display of parameters. In ICL these activities are expressed in Theme statements. The distinction also allows the formalization of operational displays based on the strategic role of the associated statement. Several mechanisms are used to extend the flexibility

The following statements bring in some new issues:

- Communicating text messages to the outside operator's world. This is done simply with the Message statement.
- Using "side effects" to coordinate related statements, and to clearly express failure or contingent responses based on prior statements. This is addressed under extended Calls.
- Using simple statements whose function extends over a long interval of time to define an overview of activity under which more detailed statements can be coordinated and structured. The Theme statement represents a collection of specialized forms based on specialized usage patterns making use of the already discussed Footnotes, designed to support this coordination and structuring.
- Coordinated control under Idioms.

Briefly, ICL defines several global variables whose Real or Discrete values can be set in different ways to pass data through Calls. These include **Result**, **Remainder**, and several others. Thus, a Call can pass several kinds of data back, for direct or indirect usage. Among other higher statement forms, the Local State Environment Declaration statement depends critically on this action.

Messages

Conceptual Role and Purpose: To represent and control the use of operator and logging messages.

Detailed Role:

Messages represent text set aside for operator display and logging. The basic form is expressed as:

"THIS IS A MESSAGE".

Certain system elements may generate a message on execution, indicating internal results or failure conditions. This message can be displayed by the command **MESSAGE**. Also, the Keyword **LOG**, after a semicolon, after a statement, causes the text of the statement to be so set aside:

V100, OPEN; LOG.

The message so set aside would be displayed and logged in some standard location. Alternatively a variant form of Scoping Prefix can be used to specify (from a set of named logical display/log locations) the point of display:

V100, OPEN; ON CONSOLE: LOG.

Extended Calls; Local State Environment Declaration Lists and Statements

Conceptual Role and Purpose: To represent the natural use of locally selected and generated States and values to affect their neighboring computations.

Detailed Role:

The system passes the current System State through the system variable **State**. This variable can be considered to be a bag of State values which is compared to the explicit State Prefix Expressions in a State Prefixes. The State Prefix allows the execution of the rest of its statement if one of the States in this bag matches the named State Prefix Expression. The system also supports a system variable **Result** which can be used to pass data between successive statements and calls. The **Result** variable can be considered an extension of the **State** variable. **Result** can be set explicitly or copied into some other variable in an assignment. **Result** is initialized to the single valued Null State and zero Real value whenever entering a fresh Sequential or Looping Activity statement unless that statement is preceded by a semicolon. Any Assignment whose right hand expression takes the form of a call will assign the final value of **Result** computed by the called Task. Certain System commands or expression pass data in **Result**. State Prefixes test the State values in **Result** if they are preceded by semicolons.

The most common usage is with Local State Environment Declaration Lists. These consist of bracketed Lists whose elements are separated by semicolons. The opening statements of the Styrene Plant example illustrate this usage:

STYRENE_PLANT Page: Procedures

```
◊ [FURNACE; REACTOR; HEAT_RECOVERY; FEED_TANKAGE; SEPARATOR];
  ACTIVE: "END ALL OPERATIONS"; END◊
```

The listed elements each contribute States to **Result**. In the example, these elements return their States. The particular State Prefix test whether any one of the elements has returned an **ACTIVE** State. But if a referenced element consists of a numerical (Real, Time, or Count) valued variable, followed a comma and an constant numerical value or parenthesized numerical valued expression, the entire element will contribute the appropriate **LO/HI/EQUAL** value to **Result**:

STYRENE_PLANT. FURNACE Page: Procedures

```
◊ [FST, 450; FSP, 100];
  LO: END◊
  CONTROL_FURNACE
  HOLD
```

reset to zero but then to increase normally and continuously. For absolute timing (in the **DATE** State), the State **RESET** undefines the value, and the State **SET** sets the value to the current date. Normal assignments can be used to set alterntaive values (for comparison, etc.).

Under relative timing, the variable also supports the independent operational States **EARLY/TIME/LATE/_** to summarize the timing relative to a SuperVariable Set CoAttribute.[96] The Early State indicates that the variable has not yet timed to the setpoint; the Time State indicates that the variable reached or passed the setpoint in the current sampling time; the Late State indicates that the variable has already reached that time. The Undefined (_) State indicates that the variable has been reset. The time value itself will have **UNDEFINED/OVERFLOW** States, indicating that it has not been set or that it has timed beyond the limits of the storage representation.

Typical Formats:

Time will be represented internally in a format natural to the system (long integer?). For the purposes of computation and display, it will be considered to include a decimal point and show at most two values after the decimal point. (The internal format and computational additions will be unaffected by the display format). A standard data format will include Day, Month, Day, Year, Hour[97], Minutes, Seconds, and hundredths of Seconds, with AM or PM indicated; for example: **MON, JAN 4, 1993, 10:42:26.37 AM**.

Keywords:

No special Archival Entry Format Keywords or symbols are involved.

Discussion:

The functioning of Time data is associated with an associated Units CoAttribute as described above.

Discrete assignments can be used with respect to Time data references to Set, Hold, or Reset them.

Logical comparisons may be made with respect to these States, with respect to **EARLY/TIME/LATE/_** and **UNDEFINED/OVERFLOW** States, and with respect to the Units name, treated as a State value from the possible set **SEC/MIN/HOURS/DATE**.

Real assignments can be applied to change the value or Time values (if they are in the **HOLD** State).

Counting Data Type

Conceptual Role and Purpose: To represent counting and counting based conditions.

Detailed Role:

Certain kinds of discrete/continuous control involve counting. Accordingly, the language incorporates counting activities into an integer valued data type called a Counting type. This type is similar to the Time data type,except that it has no Units CoAttribute. Its control States are **COUNTING/HOLD/RESTART/RESET/ADVANCE**. Their behavior is the same as with the Time Type, except for the **ADVANCE** State. This can only be set from the **COUNTING** State (Otherwise the State remains unchanged). When the Count value is set to the **ADVANCE** State, its integer value is advanced by one count, and its State is reset to **COUNTING**. The Count value can be compared to a Set Attribute returning one of these States: **LOW/ DONE/ HIGH**. These States have the analogous interpretation of Time States.

Discussion:

Discrete assignments can be used with respect to Counting data references to Set, Hold, Reset, or Advance them.

Logical comparisons may be made with respect to these States, and with respect to **UNDER/DONE/OVER/_** and **UNDEFINED/OVERFLOW** States, and with respect to the Units name.

"Real" assignments can be applied to change the value or Counting values, at any time that they are not in the Reset State.

Other SuperVariable Attributes

Role:

A

Discussion:

X.

Higher Statements and Element Forms

Role:

The statement forms provided up to this point have supported direct computations with explicitly defined variables.

[96] These States are overridden for the similar States if related to a time comparison within a Local State Environment Declaration Statement. Alternatively, multiple allowed **DATE** Units CoAttribute values might be considered: **DATE12/DATE24**.

[97] Twenty four hour time might be considered for any legal or social reason.

hyphens (**–**) to indicate an empty column entry (as distinct from an entry containing a Blank (_) named State name. The final Table shows the inclusion of subHeadings, corresponding to the parallel logic paths in the Listing Format. such a heading line is set off by the **<<<HEADINGTTS>>>** Keyword. In this case, the heading includes entries corresponding to all columns in the original heading. Hyphens can be used to indicate column positions not included in the subHeading line. The left and right parentheses correspond, in the Listing Format, to the points of connection to the main heading line (that is the sections of the main heading for which the parenthesized sections of the subTable act as an alternate).

Keywords:

Entered in this order:

Mandatory: **<<<HEADINGTT>>>**, **<<<RESULT>>>**, **<<<ENTRYTT>>>**.

Optional: **–**, **<<<HEADINGTTS>>>**.

Discussion:

A Truth Table statement is initiated by its Heading consisting of the initial **<<<HEADINGTT>>>** Keyword, followed by legitimate (input) references, separated by spaces or tabs, followed by the **<<<RESULT>>>** Keyword, followed by legitimate (output) references. The Heading is then followed by Table entries. Hyphens may be entered in place of input or output references, to allow spaces which will have actual reference entries in following subHeading lines.

A Table entry for a (following a) Heading consists of the initial **<<<ENTRYTT>>>** Keyword, followed by just the right number of hyphens or State Expressions or Simple States Expressions to correspond to the references in the preceding Heading (or subHeading). The State or States Expression entries must be consistent with the States of the corresponding Heading entry. Output expressions can include only commas.

A subHeading entry consists of the initial **<<<HEADINGTTS>>>** Keyword, followed by input references and hyphens, sufficient to correspond to the input headings of the main heading. All references must be included in a matched pair of parentheses, which must not be nested. Any final hyphens will be deleted on Archival Format output.

A Table entry for a (following a) subHeading takes the same form as one for a Heading, except that it includes no output column entries. It includes entries just sufficient to correspond to the reference entries in its subHeading.

Archival/Listing Format Relationships:

The Archival Format Keywords correspond to the corresponding Listing Format lines, with Headings and subHeadings being underlined in the Listing Format Heading, and vertical lines being positioned to correspond to the **<<<RESULT>>** Keyword and to the parentheses (with their corresponding horizontal pieces as shown).

Vertical lines from succeeding subHeadings may intersect the ledger lines of earlier subHeadings.

Archival/Entry Format Relationships:

In Entry Format, the (legal or illegal) Truth Table is indicated by an initial (unlisted) colon with or without preceding (initial) **TRUTH_TABLE** Keyword. The heading then follows as a list of spaced or tabbed (input) references, followed by a colon, followed by a list of spaced or tabbed (output) references, followed by a carriage return:

: LCH LCM LCL: LEVEL©

The corresponding line entries are entered as appropriate spaced or tabbed lists of State or Simple States Expressions, terminated by a carriage return.

SubHeadings are entered like a Heading with an initial colon (like a heading) and following references. No second colon or output references are included. SubHeadings are followed by their line entries entered as with Headings.

The Truth Table is terminated by a repeated carriage return.

Time (and Date) Data Types

Conceptual Role and Purpose: To represent timing and time based conditions.

Detailed Role:

ICL has a distinct data type for representing time in both relative and absolute dated form. Time data has a specially intimate relationship with its operating States and its Units SuperVariable CoAttribute. For Time data the Units CoAttribute can only have the alternative values: **SEC/MIN/HOURS/DATE**. The operational States include the normal states plus (in an independent set) the States: **TIMING/HOLD/RESTART/RESET** (for relative timing), or **SET/RESET** (for absolute timing). A Time valued SuperVariable Attribute is always represented internally in standard form (hundredths of seconds?). The Units CoAttribute determines its display and computational format: in seconds, minutes, and hours (with decimal point fraction), in relative time to the start of some timing period, or as an absolute date and time, relative to some standard base (1970?). For relative timing, the State **RESET** causes the indicated time to be reset to zero, interpreted as a constant **UNDEFINED**; **TIMING** causes the indicated time to continually increase naturally (becoming defined if reset); **HOLD** causes the indicated time to freeze; **RESTART** causes the time value to be

input entries define alternative and combined state possibilities. Output column entry expressions can contain only commas; they represent composite States. Only those independent States included in the output column entry will be affected in the output header reference.

heading has been filled in, they may be deleted. The Truth Table generates output states only for those combinations of input States that have been listed in the Table. Thus the Truth Table has some of the conditional responsiveness of an IF statement, which may occur in any order. In addition, if a column entry is left unfilled [blank, but not equal to the Blank (_) State] in a particular line entry, then any value may apply for that column variable or reference in that line entry.

Since these conventions (particularly with don't cares) may give rise to conflicting output values, the topmost line entry which corresponds to the input data will always be the one that provides the output values. The blank column values can be applied to all input variables on the bottom line. In this case, the corresponding output values thus define the default output values to be applied if no earlier input line matches. This notation allows multiple Table sections to be combined each section having different heading references, but the sections being combined to generate shared output entries. In the above example, this means (as shown above) that the first line input entry under the top set of headings is tried first.

If a match is achieved then the first line output entry applies, and the processing is complete. If no match is achieved, then the first line entry under the first lower set of headings is tried (and so on with any additional heading set). Its successful match again causes termination applying the first line output entries. If no first line entry of any set of headings, or combination of headings, works, then the second line entries are considered, a successful match causing the application of the second output line (and so on to other lines). If no match occurs, then any additional output line entry (with corresponding empty input lines under the input headings) is applied as the default result. This form represents the counterpart of a ladder diagram.

The example also illustrates the natural tendency of process States to develop in hierarchies, in this case, with the sensor States at the lowest level and the Tank States derived from them in a higher level. The Truth Table is used both because it provides a State computational means and because it most clearly expresses the even cause and effect intent of the application. Some people find the programming of complicated logic in Truth Tables more complicated than working with functional operators. This problem is minimized if the applicition is explicitly conceived in terms of a heierarchy of States as illustrated above. An application designer, used to traditional Boolean logic and confused by the computation of complex logic in Truth Tables will find life easier if he deliberately builds his logic centrally in deliberate hierarchies rather than in distributed logic as needed (the usual practice).

Typical Formats:

The Archival Format for the above three Truth Tables is:

```
<<<HEADINGTT>>> LCH LCM LCL <<<RESULT>>> LEVEL
<<<ENTRYTT>>> Lo Lo Lo Empty, OK
<<<ENTRYTT>>> Lo Lo Hi Low, OK
<<<ENTRYTT>>> Lo Hi Lo Low, Failed
<<<ENTRYTT>>> Lo Hi Hi Full, OK
<<<ENTRYTT>>> Hi Lo Lo Empty, Failed
<<<ENTRYTT>>> Hi Lo Hi Full, Failed
<<<ENTRYTT>>> Hi Hi Lo Filled, Failed
<<<ENTRYTT>>> Hi Hi Hi Filled, OK
<<<ENDTT>>>

<<<HEADINGTT>>> LCH LCM LCL <<<RESULT>>> LEVEL
<<<ENTRYTT>>> – Lo Lo Empty
<<<ENTRYTT>>> Lo Hi Lo Low, Failed
<<<ENTRYTT>>> Hi Lo Hi Full, Failed
<<<ENTRYTT>>> – Lo Hi Low, OK
<<<ENTRYTT>>> Lo Hi – Full
<<<ENTRYTT>>> Hi Hi – Filled
<<<ENDTT>>>

<<<HEADINGTT>>> LCH LCM <<<RESULT>>> LEVEL
<<<ENTRYTT>>> Lo Hi Failed
<<<ENTRYTT>>> – – OK
<<<HEADINGTTS>>> (LCM LCL)
<<<ENTRYTT>>> Lo Hi
<<<ENDTT>>>
```

The first Table is straight forward, listing first the Table headings and then the entries. The column entries are spaced for separation, but commas (and slashes in the input entries) can be followed by spaces without separating the initial part of the column entry from its continuation.[95] The second Table shows the use (in Archival Format) of

[95] Input column entries can include any State expression, but should generally include only a single operator type (commas or slashes); the

grouping several State names in parenthesis: **F100, (AUTO, LOCAL)**. In such a reference, the States need not correspond to the exact same object as long as they are associated with the same reference (the way a Block may share the reference of its controlled variable; the State names must be unique).

If the State value which is to be assigned to the left hand reference, is not a constant, but the collected State values of some other object, the expression to the right of the comma must represent this by incorporating the corresponding reference in parenthesis, instead of a constant State name: **F100, ((F200), LOCAL)**. If there is a conflict in the resulting States, they will be applied to the receiving referenced value, in order, from left to right. The States passed from such a reference may be restricted by defining a mask to be applied to the referenced value as a States Expression contained in parenthesis right after the reference name itself: **F100, ((F200(AUTO/MANUAL)) , LOCAL)**. These complications are unlikely to be used normally, but they complete the picture.

Several variables may be assigned to in the same assignment, by listing them together in parenthesis to the left of the assignment comma: **(F100, F200), (AUTO, LOCAL)**. A Discrete assignment can consist of just the State Name or parenthesized State Expression if its States are intended to be used to set system States or States of the current Operation.

Typical Formats:

As shown above.

Keywords:

No special Archival Keywords are involved.

Discussion:

A (legal or illegal) Discrete assignment consists of a State Name; a parenthesized List containing at least one State Name[94]; or a name or reference, followed by a comma, followed by a parenthesized List of names.

A legal Discrete assignment can consist of a system or current Operation State Name, or parenthesized List of such Names, that is a parenthesized State Expression separated only by commas.

A legal Discrete assignment can consist of a reference [to an object including State values (an Operation, Task, Block, or SuperVariable)] or parenthesized list of such references, followed by a comma, followed by a parenthesized State Expression separated only by commas.

In the above State Expressions, any of the State Names can be replaced by parenthesized references to objects which include State values. Any such reference may be followed by a parenthesized State Expression which is a subexpression (as defined above) to the declarative State Expression defining that reference's possible State values.

Archival/Listing Format Relationships:

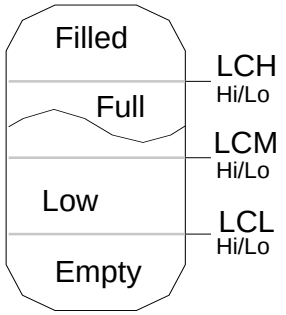
If the system permits, the assignment commas will be printed in Listing Format in larger, more distinguished font than the List or State Expression commas: **(F100, F200), (AUTO, LOCAL)**.

Discrete Data Truth Table

Conceptual Role and Purpose: To support the more complex state computations, and develop a States heierarchy.

Detailed Role:

States do not support conventional computational operators like boolean expressions. Thus more complex State computations require Truth Tables. The figure below illustrates their use to translate the states of several tank process I/O variables into states of the tank itself. This represents one kind of natural usage for States, translating from the States of lower level objects to the States of higher level objects:



LCH	LCM	LCL	Level
Lo	Lo	Lo	Empty, OK
Lo	Lo	Hi	Low, OK
Lo	Hi	Lo	Low, Failed
Lo	Hi	Hi	Full, OK
Hi	Lo	Lo	Empty, Failed
Hi	Lo	Hi	Full, Failed
Hi	Hi	Lo	Filled, Failed
Hi	Hi	Hi	Filled, OK

LCH	LCM	LCL	Level
	Lo	Lo	Empty
Lo	Hi	Lo	Low, Failed
Hi	Lo	Hi	Full, Failed
	Lo	Hi	Low, OK
Lo	Hi		Full
Hi	Hi		Filled

LCH	LCM	Level
Lo	Hi	Failed
		OK

LCM	LCL
Lo	Hi

The ICL Truth Table has been generalized to cover situations now covered by Ladder Diagrams and other computational forms. While the system will provide defaulted Table entries covering all possibilities, once the

[94] An unrecognized User Name or parenthesized List of unrecognized User Names will be assumed to be an illegal Call or List of Calls.

Unless the States Expression declaration includes a dot or is preceded by a colon with or without preceding numerical mask representation, its alternative State names and independent sets of State names are reordered canonically (alphabetically in name and set expression), before parsing to an internal Packed Boolean specification. Where multiple declarations share State names, one or the other of the declarations may be altered by the system to include extra unnamed State positions or bits.[91]

The normal States Expression (reordered or not) develops the States in each field, numbered (including 0) from the first alternative State name (those names separated by slashes) on up.

Two slashes with no intervening name (**AUTO//MANUAL**) indicates a State value without an assigned name. This contrasts with the underscore representing a blank or space named State (**AUTO/_/MANUAL**).

Two slashes with an intervening dot (**AUTO/./MANUAL**)[92] indicates skipping to the next free bit position to represent the next State named value (starting with a zero in that bit position as the value of the first such State.

A comma in the States Expression indicates the start of a new field.

Normally a new field will start with the first uncommitted bit to the right in the Packed Boolean value, extending leftward including any free bits as needed.

Alternatively, a (hexadecimal) number at the start of a new field declaration followed by a colon defines a mask specifying the bits to be included in the new field. The State values will then be filled in consecutively from right to left within that field (earlier State values being assigned numbers generated form the right most possible bits).

State Prefix Expressions used in State Prefixes or assignments, are compiled to match the format of the data or format to which they are being compared or assigned (to simplify the computation).

The parsing of expressions must be back parsed for Listing prettyprinting.

States Declared to Reflect Numerical Conditions:

States can be associated with Real, Time, and Counting values in the following ways:

A Discrete **STATE** and **STATES** Attribute pair associated after a Real **VALUE** Attribute[93] can be used to associate States with real values:

In terms of translating Discrete assignments to Real valued effects, a modified States Expression declaration of the form, **CLOSED=0/ OPEN=100**, associated with a named Real (valve) variable **V100**, allows one set the valve Closed or Open, with a Discrete assignment and have this set the Real value to 0 or 100 as appropriate.

In terms of translating from Real values to States, a modified States Expression declaration of the form, **COLD≤0<COOL≤10<WARM≤20<BALMY≤30<HOT**, allows the State value of the State Attribute to be set based on the Value Attribute.

States can be made time dependent, with or without explicit associated Time valued Attribute, with a modified States Expression declaration of the form, **START(10SEC)/ RUN(50SEC)/ STOP(10SEC)**, where the parenthesized times define a delay for each state. With this notation, a set of system states, **START/HOLD/RESET, EARLY/TIME/LATE/_, UNDEFINED/OVERFLOW**, functioning as the corresponding Time data type States described below, permit each State to be timed relative to its assigned time span.

Similarly States can be made Counts dependent with a modified States Expression declaration of the form, **START(10)/ RUN(50)/ STOP(10)**, where the parenthesized times define a delay for each state. With this notation, a set of system states, **START/HOLD/RESET/ADVANCE, UNDER/DONE/OVER/_, UNDEFINED/OVERFLOW**, functioning as the corresponding Counting data type States described below, permit each State to be associated with a counting operation relative to its assigned (pulse) count span.

Discrete Assignments

Conceptual Role and Purpose: To represent the simple changing of equipment States.

Detailed Role:

Discrete Assignments allow the simple setting of Discrete variables, as an alternative to the more complex Truth Table usages. An ICL object (variable, SuperVariable reference, Operation, Task, named Blocks) can be set to any State by a simple expression consisting of the object name or reference, followed by a comma, followed by a State name: **F100, OPEN**. Only the independent State field will be affected in the corresponding Packed Boolean expression. The corresponding State may be an independent **STATE** SuperVariable Attribute Principal Context, or a Secondary Context or object State associated with the initial reference. More than one field for the same reference may be set by

[91] This is done, where possible, if it will minimize the number of distinct declaration expression to simplify the remapping of States between variables.

[92] The dot was chosen to allow any other ICL operator (except the slash or comma) or expression to be a State name. The intended usage, permits choices to be expressed based, not only on normal names, but on other expressions that may come up. For example the State expression **+/-** is used in the implementation of a controller to indicate the expected process gain sign.

[93] This could also be done by including a Real value as a Secondary Context. This would lose the scaling support unless the Real **MIN/MAX** scaling Attributes could be made applicable to both Principal and Secondary Contexts. This strategy is even more applicable to the following Time and Count related proposals, where the scaling is not needed.

Pretty-Print Conventions:

The Pretty-Printing is the general form already described for operators and words, except that prefix words are unseparated from their following **(**.

State Named (Discrete) Data Type

Conceptual Role and Purpose: To represent the local States of process variables.

Detailed Role:

The ICL variable must also allow the representation of both system (Secondary Context) and user defined (Primary Context) States. The latter are represented by **STATE** Attributes. This depends on the already introduced need for State declaration, definition, and computational means. It also introduces its own novel problems. User defined values must be declared (as above for all user defined State data). The **STATES** Attribute provides this function. It performs a role similar to scaling of an analog variable: it defines the range of legitimate values and supports their conversion to and from internal and field forms.

As indicated earlier, Discrete or State data allows the reference to analog and discrete data of a single variable under the same name. Discrete variable Attributes will additionally have the normal operating States.

Typical Formats:

As declared on a Definitions Page in a **STATES** Attribute, the defined alternative State value names are expressed in the above described States Expression: **+/–, AUTO/MAN/TRACK, LOCAL/REMOTE**. The definitions also allow for setting aside a number of unNamed States, by including two or more slashes with no intervening symbol or space; or a blank State, by including a pair of slashes with intevening space or underscore. All such Expressions will normally be pretty-printed (in canonical form, as shown) during back compilation so that the alternative value names and sets are alphabetically ordered. But when used in connection with field devices, the **STATES** declarative expression can be more complex, including either numbered colons, or dots (periods), to force the mapping between Expression and internal representation: **5:START/STOP//HOLD,30:LOCAL/REMOTE/./CASCADE**. In this case, the order of States in the Expression is held fixed during back compilation for all affected subExpressions. A colon, by itself, before a given subExpression also inhibits its reordered prettyprinting, without imposing a mask; these special notations introduce further specifications into the mapping of the States to Packed Boolean values, as described below.

Keywords and Formats:

The Archival, Listing, and Entry Formats are identical, including the comma, slash, –, +, :, and *.

Discussion:

While all State related Expressions are subject to the same rules, the Discrete data Type emphasizes certain special conventions:

A Discrete **STATE** Attribute (or the Discrete Context of any SuperVariable Attribute, or the State value of any named Task, Block, or Operation) can consist of any number of independent States, each with any number of named alternative State values.

A Discrete Attribute or Context is represented internally as Packed Boolean (with a perhaps limited number of bits, whose limit we will ignore in this specification), each of whose fields may be of arbitrary size, but containing enough bits to represent the assigned alternative values.

The Discrete data is represented in terms of State Names and expressions.

The **STATES** Attribute States Expression includes the use of special symbols (e.g. **+** or **–**).

Any of a States Expression's sets of alternative State names corresponds to a subState, and one of the Packed Boolean fields.

A Discrete data value or States Expression (type) declaration amounts to the declaration of all possible combined values of the State and of its subStates.[90]

In a particular Discrete State Expression value, the named States correspond to the values of the individual fields in the representing Packed Boolean value.

A particular Discrete value corresponds to a single State with a single State Name (to a single Packed Boolean integer value); or it can be composite, represented by a State Expression in which there are no slashes. The named States in this State Expression correspond to the values of the individual fields in the Packed Boolean value.

A Discrete value, having no explicit or implicit States Expression declaration, includes only one subState, including all of its possible values. It takes, as its default declared set of alternative values, the set of integers.

A Discrete value can be matched against a States Expression (checking to see if each of its named subStates corresponds to a named State in a set of alternative State Names of that expression). This matching is used to interpret State Prefixes and Truth Tables.

[90] The State expression parsing will assign bits and fields, from the right most bit leftward, with the first State names assigned the lowest values in the field, but the syntax allowing unassigned extra State values.

- Otherwise (when integers are used as **COUNT** Attributes) functions cannot be used with integers, either as arguments or as returned values.
- Division in integer assignments, is fixed point division, returning a remainder through a Remainder system variable.

Whatever their internal representation, Time computations are assumed to allow for at least two decimal values to the right of the decimal point. Addition and Subtraction with one Time quantity results in a Time quantity. Division of two time quantities will result in a Real valued quantity. Division or multiplication by a Real results in a Time quantity. Other combinations are assumed undefined (meaningless and an error).

Assignments are normally (e.g. in C or PASCAL) defined in Bachus-Naur form, on in Structure Diagrams. These can be seen in the standard language definition documents. In an earlier ICL document[83] we identified a different approach. We separately distinguished the recognition of assignments, from their error detection, from their detailed syntax analysis:

A Real assignment statement (legal or illegal) is recognized as a parenthesized List or SuperVariable reference followed by an equal (=) sign. The remainder of the statement (after the = sign) will be referred to as the expression.

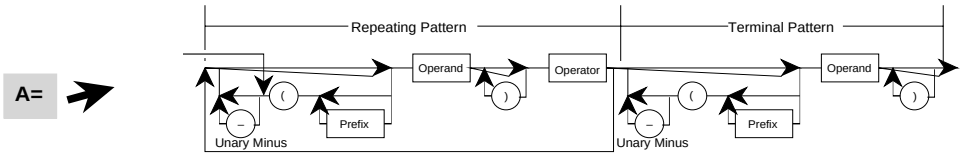
In a legal Real assignment, the List must be of SuperVariable references, and all SuperVariable references in the List (or its single reference alternative) or in the remaining expression must be treatable as references to Real values.[84]

In the execution of an assignment, the computed value for the expression is assigned to the referenced value, or simultaneously and identically to each of the Listed references.

In a legal Real assignment, the expression must conform to the following Structure Diagram[85] and to the Parenthesis Rule[86]. All possible errors are detected by deviations from these two rules![87] It will be convenient to refer to such Structure Diagrams, for which the Parenthesis, Bracket, Brace, or Generalized Parenthesis Rules apply as an additional constraint, as Rule Diagrams.

The legal Real Assignment includes, as Prefixes, the above standard function names; as Operators, the above standard operators; and, as Operands, any legitimate (Real valued) SuperVariable reference, or any Real valued constant.

The legal integer assignment does not include Prefixes, and follows the operator usage as described above.



In evaluating the meaning of the expression the operands and operators between matching parentheses will be grouped according to the conventional operator precedence.[88] Expressions are evaluated from left to right. In this framework, unary minus only applies as the first operator in an overall or parenthesized expression[89]; it then has highest precedence, acting on the constant, variable, or parenthesized expression to their immediate right. Binary **^** has next most precedence, operating on the already evaluated expression to its left and the constant, variable, or parenthesized or higher precedence expression to its immediate right. Binary ***** and **/** have next most precedence, operating on the already evaluated expression to its left and the constants, variables, or parenthesized or higher precedence expressions to their immediate left and right. Binary **+** and **–** have the least precedence, operating on the already evaluated expression to its left and the constants, variables, or parenthesized expressions to their immediate left and right.

ICL assignments require that the assigned data be transformed to reflect the intended process value. In the case of Real values and assignments, when both sides of the assignment represent individual scaled SuperVariable references, the data will be transformed so that the left hand value represents the same percentage of its scaled range as the right hand value. This supports natural scaling in Block and control computations. The assumption behind this is that the two values correspond to different representations of the same value (e.g. the process measurement and the corresponding PID Block **MEAS** parameter). It can be defeated simply by multiplying by **1** (or adding **0**).

---

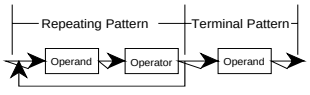
[83] Internal note: "Infix Compiler Interpreter for ICL, an example of a Software Standard Part"

[84] For an integer (Counting) assignment statement, all of the SuperVariable references would be treatable as references to Counting values.

[85] The term Operand refers to a Real valued (or where appropriate integer valued) SuperVariable reference or Real Constant (number); the term Operator refers to a **+**, **–**, *****, **/**, or **^**; the term Prefix refers to a keyword: **sin**, **cos**, **tan**, **ln**, or **exp**.

[86] As a reminder: The **Parenthesis Rule** requires that, for a particular expression: The total number of **(**'s exactly equals the total number of **)**'s; and, between the beginning of the expression and any other point, the number of **)**'s will never exceed the number of **(**'s. Recall its related rules.

[87] This structure diagram generalizes the simpler diagram which applies without parentheses:



[88] The expected implementation envisions a translation to reverse Polish representation. The Rule Diagram lends itself to direct translation to a flow chart, in which the included stack operations manage the operator precedence and the testing of the Parenthesis Rule.

[89] This restriction on unary minus (and unary plus) avoids consecutive operators and other appearance complications. It avoids unnecessarily convoluted statement appearance. At an implementation level, the unary minus is treated as a variant form of function call.

associated computations.

Detailed Role:

The language envisions three kinds of basic data, expressed as three kinds of Context: Numerical data, Discrete (State Valued) data, and Text (or Name) data. It also envisions that Numerical data may take either a Real value (for representing the value, setpoints, and alarm or control limits of analog process data) of integer data (for representing Count and Time valued data, and its associated Attributes). In this normal role, Number data can be subjected to the normal numerical computations. Integer values are also used as the default values (when there is no **STATES** Attribute) for Discrete data. In this role numerical computations play no role.

All data can be undefined, indicated as a blank (_) State, which is as an alternative for any defined value. When included as Principal Value Attribute Principal Context data, it will also have the associated Secondary Context States: **FREEZE/UNFREEZE**, **BOOKED/ UNBOOKED**. Because of the nature of the SuperVariable, any variable may have user defined States, which (by the Context Rules) can be accessed by the same Zipper Reference as the normal data value of the variable even if it is stored in a separate State Attribute.

Discussion:

Each data type is associated with different SuperVariable Attributes as their Principal or Secondary Contexts. Depending on the type of Attribute, each Attribute may have simultaneously: Text, Numerical, and Discrete

Real Data Type

Conceptual Role and Purpose: To represent the analog or real valued process variables.

Detailed Role:

Real data represents the analog process variables. It takes the form of a floating point number, when explicitly declared as a result of a definition or computation. If undefined it has an undefined State, represented as a blank or underscore. This State applies only if the floating point value is undefined; it has no other alternative State. In the language there is no special floating point representation. Instead, these values can be indicated by a computational expression using the asterisk multiply or slash divide, and caret power operators: **1.23*10^5** or **1.23/10^5**. On output, the pretty-printing will generate a similar expression, explicitly interpretable as a computational expression. Any data value can be entered in as arithmetic expression (including **+**, **–**, *****, **/**, **^**). Real data has associated operating States, as defined with the related SuperVariable Attributes. The Real value itself will have **UNDEFINED/UNDERFLOW/OVERFLOW** States, indicating that it has not been set initially in the program, or a computation underflow or overflow.

Discussion:

The Real data type applies as the Principal Context for: **VALUE**, **MIN**, **MAX**, **HI**, **LO** Attributes; and, when applied as CoAttributes to a **VALUE** Attribute, for: **SET** Attributes.

Real and Integer (Number) Assignments; Real and Integer Expressions

Conceptual Role and Purpose: To represent conventional numerical computations.

Detailed Role:

The Real assignment statement [applicable both to Real data, to Time data, and to integer (Counting) data], serves as the basic means for carrying out simple numerical computations. The syntax is similar to most standard computer languages, with an assignment operator (=), the basic set of computational operators [**+** (addition), **–** (for subtraction and unary minus), ***** (multiplication), **/** (division), **^** (power)] and functions (**sin**, **cos**, **tan**, **ln**, **exp**). The syntax is the same for Listing and Archival formats.

The Real expression can be used (applied to real, time, and integer data) whenever a single real (or integer) value (or constant) could be applied in a statement, subject to the same integer restrictions as the assignment.[82]

Typical Formats:

The Listing or Archival format are identical: the typical assignment with = operator:

```
A = 1
A = A+1
A = B * X + C * (X + Y) / Z
```

Discussion:

The discussion will refer to Real assignments and expressions. The same assignment or expression form is applicable to other number based (integer) values, except:

- Integers used as default States cannot be used in assignments or numerical computations or expressions.

[82] For now, it is intended, as making most sense, that declarative values (on the Definitions and Parameters Pages) be restricted to constants.

Discussion:

Within an Operation, List names, like all subOperation, Task, and State Names, must be mutually unique.

Lists are declared on the Definitions Page with an initial Archival Format **<<<LIST ENTRY>>>** Keyword; followed by a List Name; optionally followed, in parenthesis, by a Model element (variable, Block, Operation, Task) name; followed optionally by a dot and parenthesized list of (unique) instance names; followed by a colon; followed by the List (in parenthesis) to be declared, or other List equivalent as described above. The Model element may be itself in an additional pair of parentheses, if only its value is to be modeled, in which case all List elements are in parenthesis also.

Explicitly declared Lists can be compared against their usage in any programmed activity. Every execution of a reference to them must conform to the context of their application, either with respect to their Model element.

Lists used as objects (right hand operands) in assignments must be checked as having only Discrete value, to be used in Discrete assignments.

Implicitly declared Lists, used as subjects (left hand operands) of assignments must include elements matching the object (right hand) operand, and include any necessary allowed State values needed to correspond to the object operand.

Task definition argument List elements must individually be consistent with their programmed usage (as declared language objects); Call argument List elements must match (by type or Model) their definition counterparts.

FIRST(Argument List**)** acts on its Argument List to position the internal pointer to the first element. If this same command is used before a colon, it acts as a State Prefix, testing that the internal pointer of the List is in fact positioned on the first element.

NEXT(Argument List**)** acts on its Argument List to increment the internal pointer to the next element, if there is one. If, instead, the List has no more elements, and the statement is followed (terminated) by a semicolon, the next statement can have a **END** State Prefix[81] which will define what to do in the case of termination. If the statement is not followed by a semicolon, the **NEXT** statement acts like an **END** statement if it fails to find any more List elements. In this case it can take all of the **END** statement behaviors (moving asterisks, diamonds, etc.). The **NEXT** statement can be followed with State Prefixed statements only if that statement is terminated with a semicolon. In that case the **END** statement action does not apply. The Prefixes are based on the returned States: **END** (overriding the last List item), **NEW** (encountering a nested List), **OLD** (overriding the end of a nested List item).

LAST(Argument List**)** acts on its Argument List to position the internal pointer to the last element. If this same command is used before a colon, it acts as a State Prefix, testing that the internal pointer of the List is in fact positioned on the last element.

PREV(Argument List**)** acts on its Argument List to decrement the internal pointer to the next earlier element, if there is one. If, instead, the List has no more elements, and the statement is followed (terminated) by a semicolon, the next statement can have a **END** State Prefix which will define what to do in the case of termination. If the statement is not followed by a semicolon, the **NEXT** statement acts like an **END** statement if it fails to find any more List elements. In this case it can take all of the **END** statement behaviors (moving asterisks, diamonds, etc.).

MATCH(Argument List of Lists**)** Sets the internal pointers of all element Lists to the same relative position as the internal pointer of the first element List, the corresponding level of nested List or end element. States associated with **MATCH** are: **FAIL** (No match possible because of too few appropriate elements), **MISNEST** (the nesting of the Lists doesn't match, but the value has been taken based on the number of Lists of the intended Listing level, **NESTED** (the nesting and choices match), **MATCH** (a single level unnested match has been made).

Archival/Listing Format Relationships:

The **<<<LIST ENTRY>>>** Archival Format keyword identifies the List declarations on the Definitions Page. All such entries are grouped together in the Listing Format after other definitions.

Archival/Entry Format Relationships:

List Entry Format identifies the start of a new List declaration after normal variable Definitions Page entries by the double carriage return terminating that Definition Page Group, followed by a unique name, not a legitimate Attribute name, followed by the above normal List definition format. If the resulting syntax is in error, the line can be re-entered as a corrected List declaration or the proper Header declaration for a new Group.

List Entry Format identifies the start of a new List declaration after an old one by the intervening carriage return.

A double carriage return terminates the List declarations, allowing a new Header declaration (or a further List declaration to be recognized).

Data Types and Basic Computational Statements

Conceptual Role and Purpose: To represent the basic data for representing process variables and States, and time and counting, and the

[81] For example:
```
NEXT(LIST);
END: FIRST(LIST)
```

```
BATCH_PLANT. BATCH_TRAIN: [2*]                                  Page: Operation
        (ReactorA*(Reactor): (Reactor1/Reactor2), Load1, Load2, Water_Load, StoreA(Store): (Store1/Store2))
```

```
BATCH_PLANT. BATCH_TRAIN: [2]                                   Page: Procedures

CHECK_BOOKING
  [ReactorA; StoreA] Reactor1, Store1:
.   [BOOK(ReactorA, Charge1, Charge2, OV11); FREEZE(OV12, CLOSE; OV21, CLOSE)]
    WAIT NONE FAIL; EXECUTE·
  [ReactorA; StoreA] Reactor1, Store2:
.   [BOOK(ReactorA, Charge1, Charge2, OV12); FREEZE(OV11, CLOSE; OV22, CLOSE)]
    WAIT NONE FAIL; EXECUTE·
  [ReactorA; StoreA] Reactor2, Store1:
.   [BOOK(ReactorA, Charge1, Charge2, OV21); FREEZE(OV11, CLOSE; OV22, CLOSE)]
    WAIT NONE FAIL; EXECUTE·
  [ReactorA; StoreA] Reactor2, Store2:
.   [BOOK(ReactorA, Charge1, Charge2, OV22); FREEZE(OV12, CLOSE; OV21, CLOSE)]
    WAIT NONE FAIL; EXECUTE·
  (Charge1, Charge2), START

TRANSFER_RECEIVE
  [ReactorA; StoreA];
    Reactor1, Store1:  OV11, OPEN; WAIT 5 MIN; OV11, CLOSE;
    Reactor1, Store2:  OV12, OPEN; WAIT 5 MIN; OV12, CLOSE;
    Reactor2, Store1:  OV21, OPEN; WAIT 5 MIN; OV21, CLOSE;
    Reactor2, Store2:  OV22, OPEN; WAIT 5 MIN; OV22, CLOSE;
```

Typical Formats:

The earlier discussion and footnote describes several equivalent List forms used and generated (pretty-printed) in different contexts. Lists can be declared explicitly named on the Definitions Page, or implicitly as part of a statement or as an argument List in a Task definition or Call. Argument Lists can be declared with names in the Task definition by preceding the List by the name followed by a colon (e.g. the Call: **TASK ARG_LIST:(X, Y, Z)**). This permits each or any level of argument in a nested argument List to be named. In a Task definition, a numbered or open ended List of unnamed arguments (of identical type) may be expressed as a List declaration Name followed by a parenthesized Model variable Name, followed by a colon followed by the number or two or more dots in brackets (e.g. **TRFX(TRF):[5]** or **TRFX(TRF):[..]**). The List Name, assisted by numbered dot reference or iteration control commands, supports the reference to the List elements. When several definition arguments are declared as Lists in this way, all such arguments whose final dots are of the same number, must have the same number of actual List elements in any particular Call. Earlier Operations discussion also defined the *Operation List*, declared by following the Operation Name by a colon and List definition.

On the Definitions Page[79] List declarations might look as follows:

STYRENE_PLANT Page: Definitions

NAME	IN	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
FST	1	_	0	800	DGC	_	780	300	20
TRF	2	_	0	100	GPM	_	_	_	20
TFF	3	_	0	100	GPM	_	_	_	20
TF	_*	_	0	100	GPM	_	_	_	20
LDist	4	_	0	300	FT	_	30	10	20
LH2O	5	_	0	300	FT	_	30	10	20

LISTS

TANKAGE_FLOWS: TF2(TRF): (TRF, TFF, TF)

LEVELS((LDist)): ((LDist), 0, _, (LH2O))

The first List is modeled after **TRF**; each entry represents a full SuperVariable. The second List is modeled after **LDist**, but only the Real value is included for each entry in the list[80]. Any List references to other Attributes for any of the elements will always come from **LDist** SuperVariable. The corresponding Archival Format might appear as:

<<<DEFINITIONS>>> STYRENE_PLANT
<<<HEADING>>> NAME IN VALUE MIN MAX UNITS SET HI LO DEV
…
<<<ENTRY>>> LH2O 5 _ 0 300 FT - 30 10 20
<<<LIST ENTRY>>> TANKAGE_FLOWS: TF2(TRF): (TRF, TFF, TF)
<<<LIST ENTRY>>> LEVELS((LDist)): (0, _, (LH2O))
<<<ENDD>>>.

Keywords:

<<<LIST ENTRY>>> (Definitions Page only).

[79] In every other context, their syntax requires no special formatting or keywords.

[80] Indicated by the extra parentheses around the model variable name.

Within an argument List, a variable is indicated as a SuperVariable reference.

When any SuperVariable argument reference is in parenthesis it means that a specific data value is being passed, not a general SuperVariable reference.[76]

- When in a definition, this means that only the value (with any variable name) can be passed in the Call argument; remaining Attributes, referenced in the definition body, are forced from the definition argument; the Call value must be consistent with these Attributes (as in the case of a State value limited by the States Attribute).
- When in a Call, the definition must also be parenthesized and the appropriate value is truncated from the reference, consistent with the definition.
- When a constant is passed in the call it must correspond to a parenthesized definition argument and be consistent with the definition; if a real value, the definition must support a real value; if a State, the definition must include that State in its States Attribute. Any reference to an associated name will return the constant value as a text name; any other Attribute reference, will be drawn from the definition call SuperVariable.

When an argument is an unparenthesized SuperVariable reference, the Call argument reference must be modeled against the definition reference. When the definition reference is not to a single data value, but to the SuperVariable as a whole, this will require a matching Call argument such that any references in the body of the definition created by extending a dot reference onto the definition argument will unambiguously match the corresponding extension to the Call argument.[77]

A SuperVariable Call or definition argument may be a dot reference to a SuperVariable, extended to include normal Zipper Referencing.[78] A single valued (parenthesized) argument, in this case, stands for the data type of the referenced Attribute. A general (un-parenthesized) reference, requires a Call argument SuperVariable, whose continuing Format matches the rest of Format of the definition argument SuperVariable. (The earlier parts of either SuperVariable are irrelevant.)

A Task definition argument List, must type or model each of its arguments in one or other of the above ways: by using the name of an already defined object, by following the name of the argument by a parenthesized name of an already defined object, or by declaring a list of alternative Call arguments.

In a definition argument List, all List arguments indicated as given number of bracketed dots must have the same number of elements in the corresponding call; **[..]** matches **[..]**, and **[...]** matches **[...]**.

Lists and their Derived States:

Lists and State declaration expressions have an analogous notation through their use of commas and slashes. The expression with slashes can indicate either a List with possible alternative elements, or composite State value with permitted alternative individual State values. The expression without slashes can indicate the particular List with chosen elements (or Call argument values), or the particular composite State. The analogy can be used to indicate the selections made in particular argument or similarly specified Lists.

This parallel can be used deliberately to define a State associated with any named List, whenever any element is restricted to be from some slash separated list. The possible associated State names will be the corresponding List element names. The resulting States can then be tested in State Prefixes, for instance to condition computations on the users choice of Task arguments, without special effort or ad hoc means of passing the choice. The example below shows restricted arguments in an Operation Call being used to define State Prefix tests in its Tasks:

---

[76] This rule applies more generally to indicate Lists of specific values. As described above, the usage requires that each such reference indicate the Principal Context of the specified Attribute. The earlier List footnote addresses the handling of Context rule single value ambiguities.

[77] In general, definition arguments must refer unambiguously to a fixed object (value, List, etc., or Operation, Task, Call) capable of being matched to their Call counterpart.

[78] Other definition or Call arguments may also be dot referenced, but the situation is then straightforward. The use of dot references, in this way, allows arguments which are outside of the immediate scope of the Task definition and Call.

may consist of arbitrary elements, or be modeled after a standard element (Operation, Task, Block or Recipe Call, or SuperVariable). The List includes an internal pointer, indicating a currently selected element. The list serves several roles:

- It can act as the argument list in Task and Operation definitions and Calls (see earlier relevant sections). In this case, the Task or Operation definition List can include any sequence of referenced elements, and the corresponding Call argument Lists must match their definition List in number, order, and element type (argument List Tasks and Operations must be modeled after their definition List elements; named argument List Blocks must be to identical Block types; argument List SuperVariable references must be equivalent to the definition references).
- In declarative situations (such as the argument list in a Task definition), a List element can be replaced by a lower level list (with or without containing parentheses) of elements separated themselves by slashes, e.g.: **(FEED_TANK, REACTOR1/REACTOR2, STORAGE_TANK)**. In such a declaration, the lower level list represents allowed alternative list elements, when the declared structure is filled in (as by Call argument elements).
- In declarative situations (such as the argument list in a Task definition), any named elements which corresponds to an already defined object, must represent elements modeled after the defined object.
- In declarative situations (such as the argument list in a Task definition), a unique name followed by a parenthesized name which corresponds to an already defined object, must represent an element modeled after the defined object.
- Independent Lists are useful only if they can be referenced. Thus Lists can be declared and named by Naming Prefixes either as independently defined Lists, or as argument Lists, or when nested in larger Lists.[73] As indicated earlier, the Naming Prefix consists of a name followed by a colon followed by the parenthesized List. The Prefix name can be used with or without following parenthesized name, as above, to define a common model for the List elements.
- A parenthesized State definition expression (with commas and slashes) can be viewed as a list of definitions of the alternative values for the individual State valued elements. This serves as the basis for the special form of State value argument List, not based on variables.
- In specialized form with brackets it represents the Local State Environment Declaration expression and statements.
- A List acts as a counterpart to vectors or matrices (vectors of vectors). In this case, the List is declared and defined on the Definitions Page, usually Modeled after some specified program element, so that all elements can be checked to be of equivalent type. Vector references are made extending the normal dot expression to include entries which contain integers or integer (Real or Counts) valued[74] variables (in parentheses): **TANKAGE_FLOWS. 3** or **TANKAGE_FLOWS. (INDEX)**. Such a reference can include any level of preceding dot expression reference: **STYRENE_PLANT. FURNACE. VECTOR. 5**. An indexed reference is exactly equivalent to the corresponding name reference in the same List; to a direct Call or reference to the named object.[75] While a List can be declared like a vector or matrix, as an array of appropriate values, its normal vector/ matrix operation assumes that its purpose is to declare the arrangement of a set of already declared variables for the purposes of a vector/matrix oriented computation.
- A declarative List of unnamed modeled (as in the above items) elements (modeling a vector or matrix) can be declared as a bracketed number (indicating the number of elements), or as two or more dots bracketed (indicating an unspecified number of elements): **LIST_NAME: [5]** or **LIST_NAME: [..]**.
- It acts as the iteration controlling data source in Looping Activities, and is subject then to the **FIRST**, **NEXT**, **LAST**, **PREV** List functions. These functions set or change the internal pointer. The pointer indicated value can be accessed by a dot expression referencing the List, terminated by a final dot (e.g.: **TANKAGE_FLOWS.**).
- As part of the iteration control, a List can be declared with alternative Names. These Names each represent different internal pointer selections within the basic List, independently operated on by the **FIRST**, **NEXT**, **LAST**, **PREV**, **MATCH** List functions, and separately referenced through these

  (e.g.: **TANKAGE_FLOWS.**, **TANKAGE_FLOWS1.**, **TANKAGE_FLOWS2.**, the names initially declared in the form:

  TANKAGE_FLOWS: TANKAGE_FLOWS1: TANKAGE_FLOWS2: TF2(TRF): (TRF, TFF, TF)).

  This facilitates multi-looping without using indices.
- The internal pointer serves as a controlled alternative to a pointer variable. This pointer is set by the iteration control functions or any dot expression reference involving the List and one of its elements, whether by name or by index. When the List thus serves explicitly as a pointer, the List elements serve as the list of permitted targets of the internal pointer.
- As an (implicitly declared) left hand List of equivalent references in a Real or Discrete assignment, setting several referenced values.
- As an (implicitly declared) List of Task or Operation Calls to be carried out in parallel, invoked in the Listed order.
- Effectively, a List of named States defines an intended composite State, as a right hand side in a Discrete assignment.

Special Argument List SuperVariable Conventions:

[72] Because of the normal Context based ambiguity of the SuperVariable Zipper Reference rules, these rules are disabled in this context. SuperVariable references inside List element parentheses must refer directly to the intended Attribute, and to its Principal Context. When this is inadequate, the special Attribute qualifiers apply: **NC** (Numerical Context), **DC** (Discrete Context), **TC** (Text Context); as in **(F100.VALUE.NC)** vs. **(F100.VALUE.DC)**.

[73] E.g.: **LIST_NAME: (A, B, C)**.

[74] Integers are used as the value type for **COUNTS** Attribute (Counting values) and as the default Discrete values (the **STATE** Attribute) when no accompanying **STATES** Attribute defines explicit State names. Counting values support arithmetic operations and functions but State values do not.

[75] In the Definitions Page below, **TANKAGE_FLOW. 2** is equivalent to **TANKAGE_FLOW. TFF** is equivalent to **TFF**.

- Scaling can be forced by a Scaling function: **SCALE(VALUE, MIN, MAX)** (which scales the data from the neutral 0.0–1.0 range to the process units range); and **UNSCALE(VALUE, MIN, MAX)** (which scales the data from the process units range to the neutral 0.0–1.0 range). Scaling does not necessary involve saturation, and so either neutral or porcess ranges can be overranged.
- Two special functions are provided, **SET(REFERENCE)** and **VALUE(REFERENCE)**, which represent the best approximating Attribute to a setpoint or measurement value for the associated Zipper Reference. These functions are provided to accommodate the best Idiom cascading usages, taking into account the different forms of Process Inputs or Outputs.

There are a couple of other useful SuperVariable usages:

STYRENE_PLANT Page: Definitions

NAME	IN	CONV SQRT	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
TRF	2		_	0	100	GPM	_	_	_	20
TFF	3		_	0	100	GPM	_	_	_	20

In the Heading shown, the **CONV** Attribute has been entered with a **SQRT** value choice directly under its underlined Type Name, included as part of the Heading. The meaning of this is that the Square Root Conversion is to be applied to all SuperVariables in the Group (not just one). The Heading Type entry is entered as **CONV/SQRT** and the Heading as a whole is entered as: **NAME IN CONV/SQRT VALUE MIN MAX UNITS SET HI LO DEV**. The Archival Format for this Heading entry on the Definitions Page is:

<<<HEADING>>> NAME IN CONV/SQRT VALUE MIN MAX UNITS SET HI LO DEV

Another extended notation allows SuperVariables to enforce a strict hierarchical data structure Zipper reference. This uses matched brackets within the SuperVariable Format Heading:

NAME [POSITION [IN VALUE UNITS HI] FLOW [IN VALUE UNITS HI]]

The additional notation requires several additional rules:

- The brackets conform to the Parenthesis Rule, being paired in the usual way taking into account nesting.
- The Attribute name just preceding a left bracket may be user defined, without value or function. (In that case it cannot meaningfully be the final Attribute in a reference expression.)
- If a Zipper Reference match occurs on an Attribute just before a left bracket in the Format, then the next match must be to an Attribute between the corresponding bracket pair.
- No match may occur involving an Attribute between two paired brackets, unless the immediate previous match involved the Attribute immediately to the left of the left bracket in the pair.[70]

Two other extensions have been proposed for data base access: If the dot in the Zipper Reference is replaced by a comma, this could mean a List including the corresponding items from the Zipper Reference scan. As stands this can give rise to syntax conflicts with the normal List definition. Also if a dot in the Zipper Reference is replaced by an elision symbol (two dots: **..**) this could mean a List including all elements from the item referenced up to the symbol, to the item reference up to the next elision, or comma, or end of the reference. The two forms can thus be combined.

Archival/Listing Format Relationships:

SuperVariable references will be compiled internally to pointer equivalents. On listing, these compiled references will be back compiled according to the given rules, to give the simplest expression consistent with all elements of the reference.

Lists

Conceptual Role and Purpose: To represent (parenthesized) lists, in all their natural uses: as argument lists or vector matrix lists of process or parametric elements.

Detailed Role:

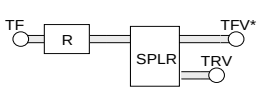
The ICL List normally consists (in both Listing and Archival Formats) of a parenthesized list of names or references to already defined elements, separated by commas.[71] Normally, these references represent a whole named ICL object or element (Operation, Task, Block or Recipe Call, or SuperVariable). However, if a (SuperVariable) reference is itself listed surrounded by parentheses, it represents the single value normally returned by that reference.[72] The List

[70] This rule ensures that only one match can occur between any paired bracket at any level of nesting.

[71] Several alternative forms are also defined, recognized in special contexts:

- Nested Lists:**((A, B, C), (D, E), (F, G, H))**, **(A, B, C; D, E; F, G, H)**, and **(A, B, C) (D, E) (F, G, H)**, where the last form applies principally to Task definitions and calls; the commas can be ignored because the listed items are recognizably Lists themselves.
- State Environment Declaration Lists use outer Brackets instead of parentheses (**[T100; T200; T300]**) and semicolon separators in the simplest case. They can be thought of as equivalent to the parenthesized cases but will always be pretty-printed with brackets and semicolons, however entered.
- Idiom Loop continuation lists, as at right. These lists represent multiple outputs to an Idiom. They can be expressed without a parentheses or commas, or in parentheses with commas. They are interpreted (and pretty-printed) in this form only in the context of an Idiom Loop Statement.

$TF_{\substack{REGULATE\\SPLR}}$ TRV TFV or $TF_{\substack{REGULATE\\SPLR}}$ (TRV, TFV)



Derived Attributes:

ACCUM.

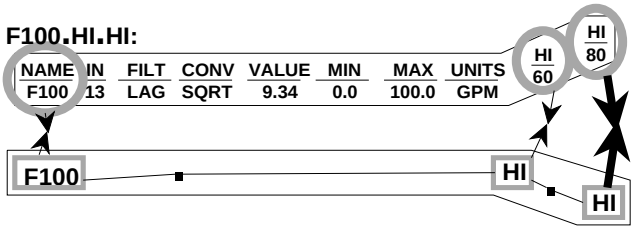
ACCUM. The **ACCUM** Attribute represents the integrated value of its associated Real Principal Value Attribute, or an accumulation of the values of an associated Counts Principal Value Attribute. In integration, it takes its Units from the units of the Principal Value Attribute multiplied by the appropriate Time scale. It has the States **ACCUMULATING/HOLD/RESTART/RESET**, to start or continue integration, hold the current value, rezero and restart the integration, or rezero and disable the integration. When applied to Counts data it accumulates only from a value which has just been changed.

Typical Formats:

The associated Formats and Keywords are defined in the Definitions Page discussion with some caveats under Lists as they relate to listed SuperVariable references.

Zipper Reference:

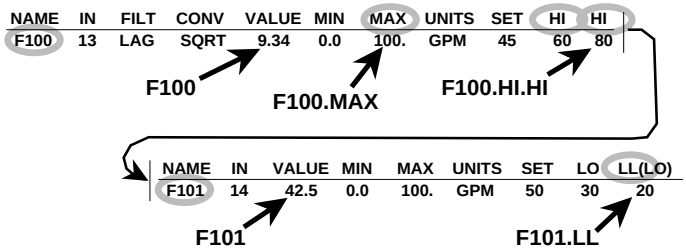
Zipper Reference represents the SuperVariable support extension of the hierarchical language dot reference and dot reference notation. The name Zipper Reference reflects a SuperVariable matching process used to localize a referenced Attribute in its reference. The process is illustrated in the figure:



Zipper Reference is intended to provide a natural strategy for referencing variables and their naturally associated operational States directly by name, and accessing parameters in their natural hierarchical relation to the associated variables, while still permitting the open ended user controlled SuperVariable structure. In the figure, **F100.HI.HI** is a typical reference to the HiHi alarm and limit of **F100**. The rules for the reference depend on scanning the reference expression and SuperVariable definition together, and, for each reference expression name, scanning further down the definition, looking first for:

- A **NAME** Attribute, whose Name value matches the reference expression name, or then for
- An Attribute Type, whose Type Name matches the reference expression name, or then, **after the final reference expression name has been matched,** for
- Either a Primary or Secondary Context matching of the current Attribute which matches the current application Context (Real in a Real assignment, Discrete in a Discrete assignment, etc.), or a further Attribute with appropriate Principal Context.[69]

The above reference first matches the **F100 NAME** Attribute and then the two successive **HI** Attributes by Type Name. Further examples of Zipper Reference are shown below:



In this case the application Context is assumed to call for all Real valued data. The references **F100** and **F101** both illustrate the application of the initial **NAME** rule followed by the final Context rule. The other references represent cases of the initial **NAME** rule followed by applications of the Type Name rule. The **F101.LL** example illustrates the application of another SuperVariable usage: If a SuperVariable Attribute Type Name is expressed as a user defined Name followed, in parenthesis by a legitimate Attribute Type name, the user defined Name represents a user defined Attribute Type Name. This Type behaves in every way (except for its Name and corresponding Zipper Reference) like the Attribute with Type Name in parentheses.

There are several special Zipper Reference related usages worth mentioning at this point:

- Assignments between two Zipper References with supporting scaling Attributes cause a scaling conversion so that the right and left hand values each correspond to the same percentage of their scaling ranges. This supports the uniform handling of process measurements and their control/controller display usages.

[69] The additional data accessible beyond the actually named Attribute, of different Context type, will be referred to as the extended Contexts of that data.

certain relationships must be defined. A given Attribute will be said to have certain CoAttributes needed to support its function.

In general, the Attributes are also grouped in Categories of similar Attributes with similar CoAttribute requirements. In order the formalize the CoAttribute relationships and simplify the SuperVariable scan, CoAttributes will be positioned predictably relative to the associated Attributes, within the SuperVariable. This position is also chosen so that the corresponding Definitions Page appearance will be natural and standardized. The positioning is also designed to make the Zipper Reference described later most natural. The are five main Categories of Attribute:

- Name Attributes: There is just the single Name Attribute, which defines as its value a single ICL User Name. It serves as a name for the corresponding SuperVariable. Specifically, under the Zipper Reference rules, the Name Attribute acts as a name for the Principal Value following it and for associated Attributes. However, it can be positioned anywhere; it does not depend on any CoAttributes.
- Principal Value Attributes: It represents the value of the process or processing variable, in one of the four ICL process value data types: Real, Discrete, Timed, or Counts. It is not itself processed during the I/O scan. But it needs, as CoAttributes, Usage Conversion Attributes (or their default) for operator display or input, and for Block interfacing.
- I/O Related Attributes: These define I/O related processing of the following Principal Value of appropriate type. These include the actual I/O operations as well as filtering and conversion or characterizing. The affected Principal Value Attribute acts as the CoAttribute of all preceding, associated I/O Related Attributes. The corresponding I/O function is carried out as part of the SuperVariable scan.
- Usage Conversion (including Scaling) Attributes: These define the appropriate conversions between internal or I/O representation, and the display format. For this purpose, the Usage Conversion Attributes are CoAttributes of their immediately preceding Principal Value Attribute and their immediately following Target Attributes.
- Target Attributes: These defined control and alarm setpoints associated with the process variable. The SuperVariable scan carries out any associated alarm tests. A special variant is included to support integration:
  - Derived Attribute: This is a calculated value based on the Principal Value Attribute with Units derived from it. It is included to support accumulation and integration where the units are the Units of the Principal Value Attribute times Time. One of its affects is to set off a whole new set of Attributes (Setpoints, alarms, etc.) based on its derived value. It thus must occur after all of the other Attributes in a SuperVariable variable definition.

In addition, two additional categories of Attribute will be useful in Field Device small language variants as alternatives to some of the large language capabilities.

- Domain Parameter Attributes: These represent general operating modes set for all following Attributes up to some associated terminating Attribute.
- Local Parameter Attributes: These modify the action of an immediately preceding or following Attribute in some function specific way.

Basic SuperVariable Attributes

This section lists the intended basic SuperVariable Attributes under their categories. The development of field devices and smaller system versions of the language is likely to add to this set since, the corresponding systems make more sophisticated use of the SuperVariables to take the place of higher level language functions.

Name Attributes:

NAME. This is the sole Name Attribute.

NAME. The **NAME** Attribute has a single text string (It can also be viewed as a State value.) representing a legal ICL User Name

Principal Value Attributes:

VALUE, **STATE**, **TIME**, **COUNTS**.

VALUE. The **VALUE** Attribute represents any Real value for a variable. It includes its own **FREEZE/UNFREEZE** State.

I/O Related Attributes:

I/O Attributes: **IN**, **OUT**, **DIN**, **DOUT**. I/O Support Attributes: **FILT**, **CONV**,

IN. The **IN** Attribute represents the software or rack address of any Real I/O Input, dependent on the arrangement of the I/O hardware. This Attribute will have associated States: **ONSCAN/OFFSCAN, GOOD/BAD**.

Usage Conversion Attributes:

MIN, **MAX**, **UNITS**, **STATES**.

MIN. The **MIN** Attribute is paired with the following **MAX** Attribute to define the minimum and maximum Real operational value for the associated variable, under I/O or control function. These define the scaling of I/O and display operations. Special scaling considerations are also called into play in connecting variables to control Blocks.

Target Attributes:

SET, **HI**, **LO**, **DEV**.

SET. The **SET** Attribute represents the Real or Discrete control and alarm setpoint for the associated Principal Value Attribute.

NAME F100_START: F100$_{REGULATE}$ V100.

Nameable statements will have default names which are overridden by the Naming Prefix. In the above Idiom Loop Statement, the default name is the same as the name of the initial Idiom and its associated Block: **F100**; the overriding name is **F100_START.**

Override Prefixes allow the override of a Booking or Freezing (or other operational) constraint in the following statement, if the affected elements are named in the Override argument list, as in:

OVERRIDE(FURNACE, REACTOR): FNHF, CLOSE

Typical Formats:

Examples occur throughout the text.

Discussion:

A State Prefix consists of a colon preceded by a State Prefix Expression, restricted to contain a single State name; or either only commas or only slashes as separators; or **ANY**, **SOME**, **ANY NOT**, **ALL**, **NONE** Keyword, followed by a single State name; or **ELSE** Keyword followed by a (similarly) restricted State Prefix Expression.

A Scoping Prefix consists of an **IN** Keyword followed by an Operation reference, or an **ON** Keyword followed by an environment Keyword.

A special form of State Prefix replaces the single colon by a double colon, indicating the initiation of a test which is to be continued until the containing Activity terminates.

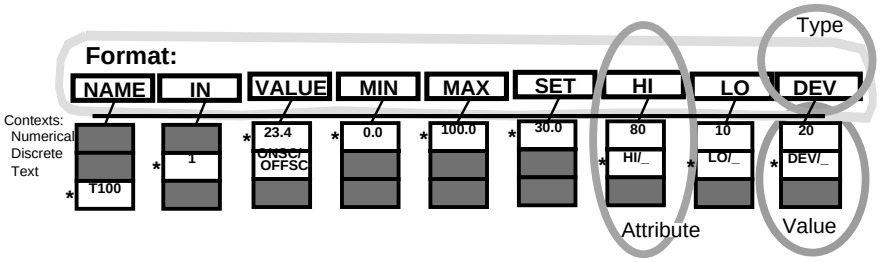
A variant from of the State Prefix allows the Local State Environment Declaration List (described in a later section) to be combined with the State Prefix directly (**[A100] HOT: V100, OPEN**), or in an altered form that allows the testing of conventional comparisons: **[a<b<c]: ….**

SuperVariables

Conceptual Role and Purpose: To represent the declaration and definition computational and control attributes of process and control variables, and the associated process I/O.

Detailed Role:

The SuperVariable provides a simple structure for defining the ICL variables with all of its associated operational States, setpoints, alarms, scaling parameters, etc. It represents the variable structure defined as an entry earlier on the Definitions Page. Conceptually, a SuperVariable is an open ended, ordered list of user chosen Attributes. Each Attribute is characterized by its Type and Value. The corresponding ordered list of *Attribute* Types is the SuperVariable's Format. The Format defines the Heading on the Definitions Page under which the SuperVariable will fall; all SuperVariables sharing identical Formats will be listed together on the Definitions Page as a single Definitions Group.



The individual Attributes normally correspond to a principal data value, which can be of one of three data types: Numerical [Real, Timed, or Counting (integer valued)], Discrete (State valued), or Text (corresponding to a name). The details of Real (and integer), and Discrete data types will be addressed shortly. However the language allows association of operational States with this principal value in secondary data values. The language supports several strategies for accessing the principal and secondary data according to context. Accordingly, we will distinguish between the Principal *Context* (asterisked in the above figure) and the Secondary *Contexts.* The Principal *Context* represents that data configured on the Definitions Page. The Secondary *Contexts* are controlled by the system even as they may in some cases be displayed by the system on the Definitions Page. By data type we may also refer to each Attribute as having three typed Contexts: Real, Discrete, Text. None of this makes any implications about actual storage and implementation.

In addition to defining the values and parameters, the SuperVariable defines associated I/O and display processing of its data. Certain Attributes (**IN**, **OUT**, **DIN**, **DOUT**, **CONV**, etc.) define process I/O conversion operations as well as including the I/O addresses (Letter Bugs) and parameters. Conceptually each SuperVariable is scanned, each sample time, more or less in the order of its Attributes, to carry out any needed process I/O. As part of this data it is converted under the scaling Attributes. These same Attributes come into play whenever the corresponding process value is connected to a control Block, assigned, or displayed. Thus, while the Attributes are nominally independent, in fact,

scope. These restrictions are expressed by Prefixes, expressions preceding a statement, separated from it by a colon. There are four current forms of Prefix: State, Scoping, Naming, and Override Prefixes. A State Prefix consists of a named State, State Expression, or Simple States Expression followed by the colon. In State Prefixes, these expressions occur in a restricted form (To avoid unnecessary complexity in the expression), including any of the appropriate State names, and either only slashes or only commas: **AUTO/REMOTE: V100, CLOSE** or **AUTO, LOCAL: V100, CLOSE**.

The States in the Prefix are either system States, or States of the current Operation, or States of an including Task definition, however Called. They may also be drawn from an associated Local State Environment Declaration statement in that case overriding (where there is a conflict) the other sources of named States. The statement following the State Prefix is executed only if the current system, Operation, Task, and Declaration statement States include subStates consistent with the above expression: If the separator is a comma, each State name in the Prefix must be exactly matched by one of the subState values of one of these elements; if a slash, the Prefix must include a name that exactly matches at least one such subState value. State Prefixes can also consist of a single State name preceded by one of the following Keywords:

- **ANY**, meaning that one or more of the current system States possible matches the Prefix State.
- **SOME**, meaning that one or more but not all of the current system States possible matches the Prefix State.
- **ANY NOT**, meaning that one or more of the current system States which could match the Prefix State does not.
- **ALL**, meaning that all possible current system States match the Prefix State.
- **NONE** or **NOT**, meaning that none of the current system States possible matches the Prefix State.
- **ELSE**, (this can be followed by a normal State Prefix State Expression) meaning that all prior associated State Prefixes have failed.

The expressions included in a State Prefox will be referred to as State Prefix Expressions. The colon in a State Prefix may be replaced by a double colon indicating a continuous test active until the including Task or Activity terminates: ◈|||FAIL:: END◈.

State Prefixes are associated with each other or with a Local State Environment Declaration statement as described above if they are in consecutive lines (the Declaration being on the top line) each line terminated by a semicolon. The relationship between the Local State Environment Declaration and the State Prefix States can be further elaborated with a Details Page Truth Table associated with the Declaration as described under the Details Page discussion. The lines are executed sequentially as described earlier, whether or not any State Prefix succeeds. If a State Prefix occurs in a line containing several statements separated by semicolons, they are executed sequentially among themselves, but all such statements occurring after the Prefix will execute only if the Prefix is satisfied.[67]

The combination of Local State Environment Declarations with Prefixes allows Case statement like forms, reflecting the view that complex combinations of conditional computation are common in process control, more clearly expressed with Case forms than extended If-Then-Elses. The thinking also reflects the notion that process State logic is best developed in terms of hierarchies of process States, as developed further in the discussion of Truth Tables.

A normal Scoping Prefix consists of an initial **IN** Keyword followed by an Operation reference followed by the colon. The following statement will execute as if the current Operation were the one referenced in the Scoping Prefix. A variant form of Scoping Prefix uses the Keyword **ON**, followed by a system Keyword representing some system or display environment. It causes the following statement to execute in the context of that environment (usually to print a message there).

Naming Prefixes and Postfixes declare User Names for Lists, Blocks, and certain Statement types. The simplest Naming Prefix consists of a Name followed by a colon followed by the named List or Block. These work because their elements do not occur in a context ambiguous with State Prefixes. In some cases, the Name will be identical to the name of an already defined application object. For Lists, the listed elements must have the same structure or type as the earlier object. Alternatively, a distinct name may be used but followed by a parenthesized name, itself corresponding to an already defined object. In this case, the listed elements must be of type or structure as the named object even as the list name is distinct.

The normal Block Call depends on a Naming Postfix: a colon following after the argument Lists and Blocks which names all of the Blocks under a single Name. Certain *statements* may have an assumed name (Idiom statements take as a default name the name of their first operand variable: **F100** in **F100$_{\text{REGULATE}}$ V100**), which can be overridden by a Naming Prefix: **F100_START: F100$_{\text{REGULATE}}$ V100**.[68] If the Naming Prefix would be ambiguous with a State Prefix it becomes unambiguous if initiated with the keyword **NAME** as in:

[67] Similarly if some statement within a line suspends, the statements following it will not execute, but lines following it, separated from it and each other by semicolons, will execute.

[68] Individual Idioms (or Idiom Lists) are also named by their preceding operand variable. Or they may have an explicit Naming Prefix like a List.

closing valves (as in V100, CLOSE), starting or stopping motors, etc., assigning a State Value to an ICL object.
- The conditional execution of statements is supported by special simplified combinatorial forms used in the State Prefix, defined below.
- The State Driven Activity carries conditional execution further: ongoing choice of actions, based on immediate State.
- When this usage is extended to representing the State of one ICL object to match the State of a second object (as in **V100, (V200)**), a complication arises: The arrangement of the corresponding State internal Packed Boolean may not correspond to each other, requiring a Truth Table to carry out the intended mapping. This is further complicated when several variables are set to the States of several variables in the same statement.[66]

The State data, as formulated supports two basic goals in clarifying programs: it allows and enforces the description of discrete events and States in terms of the intended process nature. and it allows the reference to analog and discrete attributes of a single variable under the same name.

Typical Formats:

Discrete data is entered anywhere in terms of its name as described above. As declared in an Operation, Task, or Variable declaration, the defined alternative State value names are expressed as a States Expression including the alternative values (separated by slashes) of each independent State and the set of such sets (separated by commas): **+/–, AUTO/MAN/TRACK, LOCAL/REMOTE.** Such an expression will normally be pretty-printed (in canonical form, as shown) so that the alternative value names and sets are alphabetically ordered. In State assignments, the expressions can occur in State Expressions including only commas, in parenthesis, **F100, (AUTO, LOCAL)**, meaning that the referenced object is to be set in the corresponding combined State.

Keywords and Formats:

The Archival, Listing, and Entry Formats are identical.

Discussion:

A Discrete **STATE** value or declaration can consist of any number of independent States, each with any number of named alternative State values.

The Discrete data is represented in terms of State Names and expressions.

A normal States Expression is a list of unique State Names separated by commas and slashes (with at least one slash separator occurring between any two commas or between a comma and the beginning or end of the Expression.

Such a States Expression containing both commas and slashes is a Composite States Expression.

A States Expression containing only slashes is a Simple States Expression. Thus every States Expression is either a Simple States Expression of a Composite States Expression consisting of consecutive Simple States Expressions separated by commas. A Simple States Expression defines the set of alternative State names of the smallest possible subState.

A State Expression is a list of States separated by commas only.

When used as a declaration of an independent value a States Expression shall be restricted to have at least two State Names, including at most one Blank (_) name, separated by slashes, between any two separating commas.

It will be convenient to define a subExpression of a States Expression to consist of an ordered list of the State Names, commas, and slashes, taken from the States Expression, which conforms to the definition of a normal States Expression and deletes a commas only by deleting all symbols either before that comma or after it up to the next comma. Such a subExpression is a general declaration of a subset of the possible State values represented by the expression.

A Discrete data value or State (type) declaration is represented as a States Expression, with the sets of alternative State names representing the declarations of the subStates. The value or State declaration amounts to the declaration of all possible combined values of the State and of its subStates.

A particular Discrete value corresponds to a single State with a single State Name; or it can be composite, represented by a State Expression.

A Discrete value can be matched against a States Expression (checking to see if each of its named subStates corresponds to a named State in a set of alternative State Names of that expression). This matching is used to interpret State Prefixes and Truth Tables.

Prefixes; State Prefixes, Scoping Prefixes, Naming Prefixes and Postfixes

Conceptual Role and Purpose: To represent all conditional computation, in Case form; conditional on State or Scope; and to permit naming of substructures.

Detailed Role:

In general, statements may be carried out not only unconstrained, but under certain restrictions of condition or

[66] To increase efficiency, the system will, by default, allow for a canonical representation of such values (visible in the prettyprinting): naming States alphabetically in their above declaration sets and arranging those sets also alphabetically. It will also try to match declarations where some States are missing in the declarations for some State declarations. When this is insufficient (for example, where the field device States arrangements must be preserved to reflect hardware States), a distinction will be made in the declaration.

parentheses **()** or **BLOCK** Keywords, one of either for each corresponding Task definition argument List; followed by a colon, followed by the Formula Block name; followed by an optional dot and instance name. Note that Parameters Page Blocks must have unique names for any Operation.

The resulting Block Parameters Page takes the same Archival Format as any Parameters Page listing, except that it starts with a **<<<MAKE>>>** Keyword.

On Archival Format input, the entries follow the usual Parameters Page form. They may occur in any order, and may include any parameters in the Task definition argument List. Missing parameters (in the input, relative to the Task definition) will be filled into the data base, blank, with their proper name and units. On output, all parameters will be listed in Task definition argument List order.

Footnotes; Footnote References

Conceptual Role and Purpose: To provide a general mechanism for inserting idiosyncratic computations within larger computation formats, without obscuring the result.

Detailed Role:

An asterisk, marking Footnoted element may be placed after any Definitions Page Heading or Entry Attribute position and after any statement. Theme Statements may have internal Keywords, and, where meaningful, they may contain marking asterisks after any user name which is then immediately followed by a Keyword. Each such marking asterisk corresponds to a Details Page entry as illustrated under the Details Page discussion. This entry then includes the computational statement which is represented by the marking asterisk in the original statement. In general, the Footnote is to be carried out computationally whenever the preceding (marked) statement elements are processed in the execution of that statement. For Footnotes marked at the end of statements or Definition Page Entries, this is the case. For Theme Statement Footnotes, the discussion of the Theme Statement covers the function of included Footnotes. For Footnotes included in a Definition Page heading, the result is intended to be equivalent to what would occur if the same Footnote had been marked after every column entry under the heading Attribute.

A special usage applies when several statements are listed on a line with separating semicolons. Each such statement can have a normal Footnoting asterisk, at its end. But a double asterisk at the end of the whole line can also be used to indicate a Footnote which is to be applied after each included statement.

Typical Formats:

See Details Page Footnote discussion.

Keywords:

<<<REF @.#>>> where @ represents the number of the Details Page containing the Footnote, as described earlier, and # represents the Footnote number within that Page and statement.

Discussion:

Within the Footnoted statement or Definition Page entry, the Entry and Listing Format asterisk is replaced by the Keyword **<<<REF @.#>>>** as described. The Details Page and Footnote numbers are filled in consecutively numbered by the system.

The Details Page entry occurs in order of the Footnote and starts with the Keyword **<<<FOOTNOTE #>>>** followed by the desired computational statement.

Archival/Listing Format Relationships:

See Details Page.

Archival/Entry Format Relationships:

See Details Page.

States and State Computation

Conceptual Role and Purpose: To represent process and equipment States, and more clearly express logical and conditional computation.

Detailed Role:

In ICL all traditional logic is intended to be implemented in terms of user named States: values corresponding to Packed Boolean internally, but addressed in the language in every respect as having user named values. These values will be attributes of all levels of ICL object, from Operation to Variable. This introduces several novel problems:

- The language needs a simple format for declaring the different value names for the States. This format takes the place of scaling Attributes for real data and is used in the SuperVariable States Attribute, to support the conversion of Discrete data from internal and I/O format to display format. This format lists sets of mutually exclusive State names separated by slashes, and independent sets of these (representing different Packed Boolean fields) separated by commas, as in: **+/–, AUTO/MAN/TRACK, LOCAL/REMOTE.** Special additional conventions allow specifying skipped States, bits, or fields, and otherwise controlling the order of translated States.
- The language needs an appropriate general computational vehicle for Discrete data: the Truth Table.
- But the Truth Table is too cumbersome for the simplest application. A simple State assignment is provided for opening or

```
STYRENE_PLANT                                   Page: Parameters

TASK() BLOCK BLOCK: PURPLE_LOTS
  LOT_NAME                  _
  PRODUCT_QUANTITY          _      GAL
  PRODUCT_COLOR             _
  UNIT_SIZE                 _      GAL
```

Note that the named Formulae can include several argument Lists together, and need not all divide the argument Lists the same way. This permits the expression of restrictions to the independent use of different combinations of Formulae. Above, the **PURPLE_LOTS** Formula requires not only its Lots parameters but can only be run with a particular unit. The corresponding Parameters Page incorporates both argument Lists in its Formulae, under the single name. Once these tables have been allocated, they can be filled in by the user on the engineer's work station panel, or through appropriate operator tools.

UNMAKE

Because a recipe Formula Block is not programmed explicitly as part of the application program; and its corresponding **MAKE_RECIPE** command not permanent the system cannot delete it, based on any standard program edit. As a consequence, the system needs an explicit off-line **UNMAKE** command (as in **UNMAKE PURPLE_LOTS**) which can be applied to any Formula to delete it.

ARCHIVE

There is a need to be able to file away a copy of the current state or values of an existing Block, or any other named language object, either to be able to switch back and forth between several different version of parameters, or to back up the object. The **ARCHIVE** command (on-line or off-line) would do this, storing in a compiled Checkfile form for fast reload. The form of the command would be **ARCHIVE PURPLE_LOTS** or **ARCHIVE PURPLE_LOTS AS PURPLE_LOTS.A**. The first version simply files the object (**PURPLE_LOTS**) under its own name. The second one files the name under an alternative name (in this case with an instance tag).

RESTORE

With the **ARCHIVE** command the language needs an (on-line or off-line) command to recover a file once archived. The **RESTORE** command does this: **RESTORE PURPLE_LOTS.A**. It would restore the named filed object to the on-line version from which it originally came.

Typical Formats:

The **MAKE_RECIPE** command itself does not have any permanent form; it disappears as soon as it has done its thing. However it may be included as entered above in an Archival Format Listing; it will not regenerate itself as part of a file dump. In its place the **<<<MAKE>>>** Keyword is inserted in the normal Parameters Page entry in place of the **<<<PAGE @>>>** Keyword. This informs the system that the entry is to be created even though it does not correspond to a Block Call. The above Parameters Pages would be entered in Archival Format as:

```
<<<PARAMETERS>>> STYRENE_PLANT <<<MAKE>>>
<<<HEADINGM>>> TASK () (): PRODUCT_A
<<<ENTRYM>>> PERCENT_ACTIVE _  %
<<<ENTRYM>>> PERCENT_FILLER _  %
<<<ENDM>>>.
```

and:

```
<<<MAKE>>> <<<PARAMETERS>>> STYRENE_PLANT
<<<HEADINGM>>> TASK () BLOCK BLOCK: PURPLE_LOTS
<<<ENTRYM>>> LOT_NAME _
<<<ENTRYM>>> PRODUCT_QUANTITY _ GAL
<<<ENTRYM>>> PRODUCT_COLOR
<<<ENTRYM>>> UNIT_SIZE _ GAL
<<<ENDM>>>.
```

Note the absence of the **<<<PAGE @>>>** Keyword.

Keywords:

<<<MAKE>>>.

Discussion:

A **MAKE_RECIPE** command is entered as an independent unnamed statement on the Procedures Page. It will not be duplicated back there in any file dump of the running program. The command is entered; followed by either paired

algorithm serving as an analog shift register. This latter version also relates to the Stream Theme statement function. These capabilities can be combined in the following way: The Block could have the following Parameters:

- STATES — The set of Operating and Option States: ONSCAN/OFFSCAN
- NBKTS — Number of Buckets.
- DEADTIME — The currently assigned value of the DeadTime.
- INPUT — Name of the Input Variable.
- OUTPUT — Current Output value for the DeadTime Block.
- BKT1 … BKT*N* — The set of Buckets themselves.

Break Point Function/Characterizer:

The traditional breakpoint function characterized (10 breakpoints?). This can be supported with more natural configuration to allow the breakpoints to be defaulted from some standard shapes (S-Shaped, L-Shaped, U-Shaped, Γ-Shaped, Stair-Shaped, etc.). The goal would be to relate this function more directly to Fuzzy Logic and Neural Net functions without reducing its technical integrity. The Blocks Parameters would include an **IN** and **OUT** parameter and 22 break point parameters including **OUT0** and **OUT11** (the output values corresponding to the minimum and maximum input scale values) and the ten break points: **IN1**, **OUT1**, **IN2**, **OUT2**, … , **IN10**, **OUT10**. The Block can compute its own input depending on a State value: **NORMAL**/ **INVERTED** (computing **IN** from **OUT** by the inverse) /**INVERSE** (computing **OUT** from **IN** by the inverse).

Object/Task Call Support Commands

Make_Recipe **Unmake** **Archive** **Restore**

Roles:

MAKE_RECIPE

The language has two ways of setting aside operator accessible blocks of parameters: the Block Call, and the **MAKE_RECIPE** command/statement described here (See also Recipe Call). The Block Call occurring in a defined control Task sets aside the necessary (named) Block. On the other hand, Recipes also require operationally selected parameter (their Formulae) and are not likely to correspond to a control Task Block Call. The Recipe Call is likely to be invoked from the operator's work station on demand. The **MAKE_RECIPE** command allows a Block to be set aside, before the corresponding Task Call.

For example, a Task defined at the **STYRENE_PLANT** level might have the following name and argument lists:

TASK(PERCENT_ACTIVE, PERCENT_FILLER)(LOT_NAME, PRODUCT_QUANTITY, PRODUCT_COLOR)(UNIT_SIZE).

Several particular sets of recipe formula may now be defined, for each definition argument List:

MAKE_RECIPE TASK BLOCK() (): PRODUCT_A.
MAKE_RECIPE TASK BLOCK() (): PRODUCT_B.
MAKE_RECIPE TASK () BLOCK (): BLUE_LOTS
MAKE_RECIPE TASK() BLOCK (): RED_LOTS
MAKE_RECIPE TASK() () BLOCK: BIG_UNIT or **MAKE RECIPE TASK() (): BIG_UNIT**
MAKE_RECIPE TASK() () BLOCK: LITTLE_UNIT or **MAKE RECIPE TASK() (): LITTLE_UNIT**
MAKE_RECIPE TASK() BLOCK BLOCK: PURPLE_LOTS or **MAKE RECIPE TASK(): PURPLE_LOTS**

The statement format is analogous to the Block Call format, after the initial command, the argument lists are given either as a matched pair of parentheses (to indicate an argument List which is to be skipped), or as the **BLOCK** Keyword (to indicate an argument List which is to be included in the Parameters Page Table):

Each named recipe formula would create a corresponding editable Parameters Page, although several different named argument Lists may be indicated in a single statement, each creating its own Parameters Page.

STYRENE_PLANT Page: Parameters

TASK() (): PRODUCT_A
PERCENT_ACTIVE _ %
PERCENT_FILLER _ %

or:

unFrozen, unless a Discrete assignment has in the interim also set the State of the variable to the **FREEZE** State. The system will manage the state of all **BOOK** and **FREEZE** statement and Path Tasks to coordinate all of this.

Discussion:

No Path Task can be Called recursively when already active (on the same Modeled Operation).

Built-in Block Calls

PID PIDX PIDE Ratio FeedForward FeedFwdX LeadLag DeadTime Function

Purpose: To support traditional built-in Control Blocks.

One of the ICL goals relative to control Blocks and Idioms is to simplify their form compared to traditional digital block algorithms. The SuperVariables, Idioms, and general language flexibility support those functions that, in traditional block environments, seem to require the massively complex inclusion of extra states and parameters. Accordingly, the built-in blocks are designed and supported with the view of minimizing all unnecessary parameters, States, and blocks themselves. Those Blocks that are provided are intended to include only those functions that are absolutely necessary, because:

- They are essential and so complex to program that they would complicate the application program.
- They correspond to standard operational displays, requiring the set aside of the Block data structure.
- They support basic dynamic control compensation functions, requiring accessible tuning parameters.
- They simplify the associated Idiom computations.

The following Block forms represent a minimum set. Certain combinations will also be common. For example, the Decouple Idiom is expected to set up a collection of FeedForward or FeedForwardX Blocks. If this turns out to be cumbersome in practice, a corresponding Block may be appropriate. But it can be programmed explicitly; it need not be built in!

PID:

The normal PID computation. It is assumed that this naturally includes the P, PI, PD combinations. The controller is assumed to have the supporting States: **AUTO / MANUAL, + / –, _ / INITIALIZE**, representing the normal auto/manual state, the positive or negative gain of the process (from valve to measurement), and a null or bumpless controller Initialization State. The minimum parameterization is assumed, for example: **SET**, **MEAS**, **OUT**, **XFB** (External Feedback), **PB**, **INT**, **DER**, **INTLG** (Integrator Lag state), **INTRM** (Integrator Lag state Remainder), **DERLG** (Derivative Lag state), **DERRM** (Derivative Lag state Remainder)

PIDX:

The adaptive PID controller.

PIDE:

The PID controller with nonlinear error characterization. It is assumed that the PID, PIDX, and PIDE forms are actually implemented together in a single algorithm, such that their functions can be combined (PIDEX).

Ratio:

Conceptually the Ratio is a multiplicative feed forward. This view is assumed to be compatible with its operational role.

FeedForward:

The normal additive feed forward.[65] Feed forward involves some complications and design choices. It could be left out as a Block, being programmed directly as needed. At the simplest level, the only essential complication of the feedfoward is its combination with the main control loop. This part, additive, subtractive, multiplicative, or divisive (or some home brew), needs to be inverted in the back calculation of the loop. The actual feed forward can be a simple linear or nonlinear computation, with whatever needed dynamic elements and no back calculation, whose results are fed to this combining computation. Even the combining computation could be explicitly programmed. The benefit of a feed forward Block is that it provides a standard tuning and operator interface, and a point of entry for the adaptive form. It could also be the primitive out of which decouplers are combined.

FeedForwardX:

The adaptive feed forward Block; the basis for both adaptive feed forward control and adaptive multivariable control.

LeadLag:

Our standard dynamic compensator, and a basic element for traditional advanced control.

DeadTime:

A refined Dead Time algorithm for advanced continuous control. This would be based on our special version of the bucket brigade algorithm. There is some discussion about the need to separate this from a simpler Dead Time

[65] We have considered, but not yet resolved, a concept of feed forward with explicitly programmed forward and backward calculations.

GLOBAL Page: Procedures

DRCONSTR(DV, MV) STATE:(+/–)
NEXTSTEP(DV, MV)
PID: NAME(DV) (STATE) AUTO
SET = Set
MEAS = CV.VALUE

Sense	STATE	Sense
+	–	–
–	+	–
·	·	+

**
Set = OUT
NSSAVE
XFB = Feedback

Cascaded Loop Generalization:

Compare the simple parenthesis free cascaded Loop **$T100_{REGULATE}$ $Q100_{REGULATE}$ $F100_{REGULATE}$ V100** to the equivalent parenthesized Loop in which the middle parenthesized Loop expression could be seen as replacing a single variable **$T100_{REGULATE}$ $(Q100_{REGULATE}$ F100$)_{REGULATE}$ V100**. One of the properties of our Loop algebra is that the variable in the normal case can represent a variable output from one Idiom and the same variable input to the other, but the two variables need not be the same; an intervening control expression can support their difference! Our algebra allows this two sided effect not permitted by a normal arithmetic algebra. We can use this property to do a number of things: to share final Idiom actions between direct and override paths, to express overrides within overrides, and to define multivariable controls. The other cases will be returned to later, but the multivariable case can be addressed now.

Multivariable Generalization; Multivariable Stack:

Multivariable Idiom Loop Statements replace the single controlled and manipulated variables in the above example statements by Lists of variables or nested Loop expressions. The following discussion is predicated on an execution strategy for which the resulting multivariable statement is executed as a set of independent single variable Steps. In the initial List (and later in succeeding Lists) these Steps include the consecutive Steps in each consecutive Loop expression. Between the Lists the Steps are made up of the right most operand in each element of the initial List, the intervening Idiom operators, and the left most operand in each element of the following List. Intervening Overrides are handled analogously.

In order to support these actions the List command **NEXT**, described later will be used, and also these commands for selecting the initial and final variables in a Loop expression:

INITIAL. This command takes a single named List argument. The elements of that List will consist of variables or Loop expressions. The command acts on that variable or expression pointed out by the List's internal pointer. It returns the first variable in that expression.

FINAL. This command takes a single named List argument. The elements of that List will consist of variables or Loop expressions. The command acts on that variable or expression pointed out by the List's internal pointer. It returns the last variable in that expression. **NEXTSV DIFFERENT**

Formats, Keywords, and Other Relationships:

Idiom Task definitions are normal Task definitions with no other special listing or entry conventions.

Path Calls

Conceptual Role and Purpose: To represent the elsewhere scheduled commitment of equipment combinations.

Detailed Role:

Any Call to a Path Task is a Path Call. These Calls are intended to execute the basic process re-configuration and control configuration for a multi-unit production operation in which there may be significant overlap with other potential operations. Through the special **BOOK** and **FREEZE** statement function, the Path Call is able to commit the control strategy of its related elements and the State of its related variables, so that any incompatible attempts by other processing tasks are blocked while any constant process States which may be required by the shared operation of all tasks are enforced.

While a particular Path Task is operating, it may uniquely Book some set of process elements. These will be committed until it terminates. Any override of that Booking will be communicated to the Task. At the same time some other process variables will be Frozen. These will stay Frozen (subject to override and override notification) as long as the Task is active. At the same time other Path Tasks may require some of those same variable to stay Frozen. Each of those variables will stay Frozen until all of the Path Tasks requiring their Freeze terminate; then they will become

OVSV2 (Override Save 2). This is the second Override backward computation Override suspension routine and it can also pass variables to be restored on recovery (**Feedback** and **DLasherBits** below). It returns control to the program when the system detects the point where the backward computation for the Override must end, returning control to the direct path. In the case below, the one of the stored values are returned in an (second argument List) alternative variable **LasherBits**.

The computation also introduces four more system buffer variables.

Sense. This buffer is used to pass loop gain signs from one level of overriding to a nested level or overriding.

DSense. This buffer is a deferred value counterpart for **Sense** used to store direct path loop gain signs.

LasherBits. This buffer is used to store a State Free/Blocked, indicating whether a particular Idiom path remains in control.

DLasherBits. This buffer is a deferred **LasherBits** buffer, used to store the state of that path not covered by the **LasherBits** path.

The override actions immediately suspend the computation, storing the **Sense** and **Set** buffers from the direct computation, returning in the final override computation to execute the final selector constraint computation. The choice of selector computation is based on the value of **Sense** as originally set to **STATE** (below) and later modified by intervening forward Idiom computations. The selector selects between the value of **Set** passed in the Override and the deferred value returned (as **DSet**) from the direct computations. The Hi and Lo Selector functions also pass their selection in the form of a State value which can condition further tests. In this case, a previous selection of the **Set** State (selection of the Override output, indicating actual overriding by the Override) will cause the Override related **LasherBits** to be set as **Free**; the Override is **Free**, otherwise it is **Blocked**. The **Sense** buffer can now be set to the deferred **DSense** (the state during the direct path).

At the end of the Override computation the calculation is again suspended (**OVSV1**) passing the **LasherBits** as calculated, to return as **DLasherBits**. On return (initiating the backward computation of the Override), the deferred **DLasherBits** (from the Override) are combined with the **LasherBits** returning along the main calculation. These are re-combined **Block**ing both buffers if the main path is **Block**ed, otherwise **Block**ing the **DLasherBits** deferred for the main path continuation, and **Free**ing the **LasherBits** for the override computation; otherwise the reverse occurs. At this point the Feedback buffer provides the external feedback for both paths. The process is suspended (**OVSV2**) passing **Feedback** and **DLasherBits** back to the resumed backward computation of the direct paths. On recovery this Sequential Activity terminates.

The main algorithm output (the second Sequential Activity carried out in parallel), which is initiated after the Override action, sets the **Set** buffer to the Block **OUT** parameter, and the Sense buffer to **STATE.** It suspends, and is restored when all later Idioms have completed their computations.

GLOBAL Page: Procedures

```
HICONSTR(OV, DV) STATE:(+/–)
  PID: NAME(OV) (STATE) AUTO
    SET = OV.SET
    MEAS = OV.VALUE
    **
      OVSAVE(Sense, Set) (DSense, DSet)
      [Sense];
        + :      Set = LoSelect(Set, DSet);
        ELSE :   Set = HiSelect(Set, DSet);
        Set :    LasherBits, Free;
        ELSE :   LasherBits, Blocked
      Sense, (DSense)
      OVSV1(LasherBits) (DLasherBits)
```

LasherBits	DLasherBits	DLasherBits	LasherBits
Block	·	Block	Block
·	Free	Block	Free
·	·	Free	Block

```
      OVSV2(Feedback, DLasherBits) (, LasherBits)
      Set = OUT
      Sense, (STATE)
      NSSAVE
      XFB = Feedback
```

The Loop statement has been generalized in a number of ways, some generalizations not necessary for current practice. One of these is to support cascaded Overrides. A derived Constraint is one whose sense and direction are wholly determined by a preceding Constraint Idiom. This Idiom acts like secondary Regulate Idioms in the direct path, and is similar in computation. Its main difference is the recalculation of the Sense buffer to accommodate any loop gain changes in the cascaded loop.

```
GLOBAL                                                    Page: Procedures

REGULATE(CV, MV) STATE:(+/–)
    NEXTSTEP(CV, MV)
    PID: NAME(CV) (STATE) AUTO
        SET = CV.SET
        MEAS = CV.VALUE
        **
        Set = OUT
        NSSAVE
        XFB = Feedback

FEEDFWD(CV, MV) (FV)
    LEADLAG: NAME(CV).NAME(FV)
        IN = FV
        **
        Set = Set+OUT
        NSSAVE
        Feedback = Feedback–OUT
```

The Regulate Idiom also has a named State defining attribute List (**STATE**) with the named States + and –, which define the loop gain expected for the corresponding process (corresponding to the usual Increase/Decrease switches. These programmed definitions add two further commands to the language, and two system Buffer variables:

NAME. Like the **SET**, but it replaced when called by the text name given as its argument.[64]

NSSAVE (Next Step Save). It causes the suspension of the processing at the point in the program of the command, any argument List variables onto the top of a backward computation stack. These variables will be restored to their value on recovery from suspension, unless a second argument List is included. The command returns control after the forward computations in the Step are completed and all later Idioms in the Step have executed their backward computation. In that case the restored variable values are entered into the corresponding variables of the second List.

Set. This buffer is used to pass setpoint data forward in the forward computation. In general it will pass the output of one Idiom to the input or setpoint of the next, until the end of the Step.

Feedback. The reverse of the **Set** buffer, this buffer passes data backward from Idiom to Idiom in the backward computation, generally developing the external feedback data.

In essence, the Regulate Idiom controls its controlled variable (**CV**), manipulating the variable **MV** to do so, and doing all the necessary backward computation to compute the External Feedback of preceding Idioms (the **NEXTSTEP** command). In particular, it makes a PID Block Call, carries any **+/– STATE** State values from the Idiom Call into the Block Call, sets the Block **SET** and **MEAS** parameters for the new controller call, and calls the PID Task. It sets the **Set** buffer to the resulting **OUT** parameter, and suspends the backward calculations (**NSSAVE**). When these are resumed, the external feedback for the Block (**XFB**) will be set from **Feedback.**

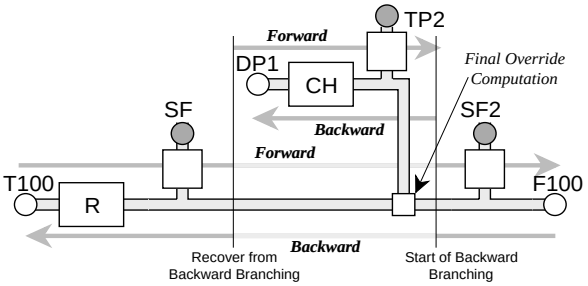
The (additive) FeedForward Idiom makes a Lag Block Call and sets the **IN** parameter of that Block from the feedforward argument **FV**. It runs the Lag calculation and adds the resulting feedforward correction (**OUT**) to the Set buffer value (passed from previous Idioms). It then suspends itself (**NSSAVE**), resuming control to subtract out the feedfoward correction **OUT** value from the **Feedback** buffer value, adjusting it for the following Idiom backward calculations.

A HiConstraint Idiom might be written similarly, but with a more complex purpose. The **NEXTSTEP** command is not used, because the override suspensions will need the current **Set** buffer value to pass data in the suspensions. As before, a PID Block Call starts by loading the Block SET and MEAS parameters for the left operand, and then the algorithm is executed. At this point two separate continuations proceed in parallel: the override actions, and the main algorithm output, analogous to the previous Regulate Idiom. The override actions depend on a coordinated set of commands, always called in the indicated order:

OVSAVE (Override Save). This is the main Override suspension routine and, like **OVSAVE**, it can pass variables to be restored on recovery (**Sense** and **Set** below). It returns control to the program when the system detects the end of the current Loop Statement Override (defined above). In the case below, the stored values are returned in the (second argument List) deferred variables **DSense** and **DSet**.

OVSV1 (Override Save 1). This is the first Override backward computation suspension routine and it can also pass variables to be restored on recovery (**LasherBits** below). It returns control to the program when the system detects the point where the backward computation for the Override must begin. In the case below, the stored values are returned in the (second argument List) variable **DLasherBits**.

[64] A special convention applies to Idiom Tasks when called: Any argument variables which are referenced from Block Calls, and any associated **SET**, **VALUE**, and **NAME** commands are to be compiled onto the Details Page. (Later, in case of List arguments for multivariable application, this will mean that a separate Details Page entry will be created for every Block Call and every involved List entry.) The **SET** and **VALUE** functions are replaced in this compilation by appropriate SuperVariable Zipper Reference expressions. The **NAME** function is replaced by the argument names.



This forward and backward computation is supported by stack processing; generalized to include the most elaborate overrides, the interpretation requires four stacks.[63] However, from a user point of view the definition of an Idiom is to be defined in a single listing in which the forward computation is carried out, followed by the backward computation, with a temporary suspension in between. Thus the problem is to provide a simple set of commands which permit all of this simply. For our discussion we can imagine PID and LeadLag algorithms as listed below, used later in the Idiom definitions:

```
GLOBAL                                                    Page: Procedures

PID((SET), (MEAS), (OUT), (XFB), (PB), (INT), (DER), (INTLG), (INTRM), (DERLG), (DERRM))
                                                                (+/–, MANUAL/AUTO)
  ERROR = SET - MEAS
  ...
  INTLG = (XFB*INT+DelT*INTLG+INTRM)/(INT+DelT)
  INTRM = Remainder
  ...
            – :  ERROR = –ERROR
  INITIALIZE :  INTLG = XFB–ERROR/PB
       AUTO :  OUT = (ERROR+INTLG)/PB

LEADLAG((IN), (OUT), (LEAD), (LAG), (K), (LGINT), (LGRM))(+/–, MANUAL/AUTO)
  LGINT = (IN*LAG+DelT*LGINT+LGRM)/(LAG+DelT)
  ...
  OUT = (LEAD*IN+(LAG-LEAD)*LGINT)/LAG)

NEXTSTEP(CV, MV)
* FirstIdiom/Override: Set = SET(CV); END*
  SET(CV) = Set
  Feedback = VALUE(CV)
  NSRESTORE
```

The Initialize State is both a system State and a State of a particular Idiom Loop Statement containing the Call to the Idiom. A Block will act on an Initialize State Prefix if called from a Loop Statement whose State is set or if the system State is set. The PID and LeadLag computations are expected to be called in Block Calls, which provide the argument data in the Block. The above listing also includes ICL pseudo-code for a system function called **NEXTSTEP**, which uses several other system calls:

NEXTSTEP. This is a system call executed at the beginning of any new Step (called from Main Idiom definitions which will start such Steps. It initiates any new Loop statement or Override, but otherwise initiates the backward calculation of the previous step.

SET. This function takes an argument List and refers (either as a value returned, or as a location stored into from an assignment) to the appropriate command value for the referenced variable under Idiom control. This is the **SET** Attribute for a measurement under control, and the **VALUE** Attribute for an actuator variable.

VALUE. Like **SET**, but it refers to the appropriate actual value of the variable (usually the **VALUE** Attribute).

NSRESTORE. This recovers the information for the backward calculation of the Step, carrying out each Idiom's backward computation until the Step backward calculation has been completed (treating each entry appropriately until the Main Stack is empty).

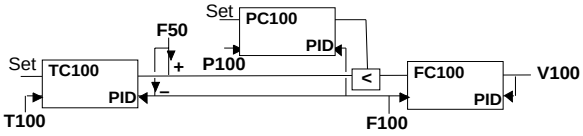
The arguments in the Loop statements (as interpreted into Calls) will match the corresponding definition arguments as in the other forms of Call. But the final argument List, with its two arguments, is treated specially because its argument references will include operate on both associated Set and Value Attributes. A simplified Regulate and FeedForward Idiom might be written as:

[63] The deferral for each Idiom's backward computation at the end of its forward computation, as part of each Step computation, is supported by a Main Stack and the **NEXTSTEP**, **NSSAVE**, and **NSRESTORE** system commands. Override data and execution deferral from the Override Idiom to the final Override computation, the corresponding end of override and beginning of override computations in the backward computations are supported by Override, First Data, and Second Data Stacks, and by the **OVSAVE**. **OVSV1**, and **OVSV2** system commands. Initial and final marks for each Override are also placed on the Main stack by the system, as **OVSAVE** is first executed and as the final override computation is carried out.

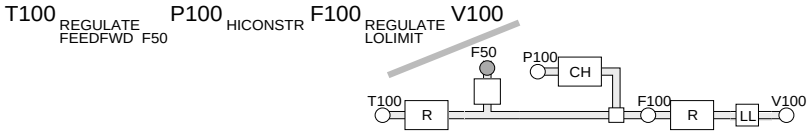
Conceptual Role and Purpose: To represent user defined Idioms: supporting user standardized control and control connection strategies.

Detailed Role:

The actual Idiom Call will be discussed under Idiom Loop Statements. But these statements are intended to allow for user defined Idioms. As with conventional algebraic computational statements, Loop Statements represent complex interacting combinations of calls to the underlying computations. For the purposes of our discussion we can take the conventional loop diagram:



which translates directly to the corresponding Idiom Loop statement below [note correspondence between diagram and the Idiom Loop statement and iconic forms:



Subject to certain re-orderings of the computations and special conventions of the passing of data and intermediary result, the statement can be thought of as carrying out the following Simple Calls with argument Lists (and one State setting):

REGULATE(T100,F100); FEEDFWD(F50)(T100, F100); HICONSTR(P100, F100); REGULATE(F100, V100) –

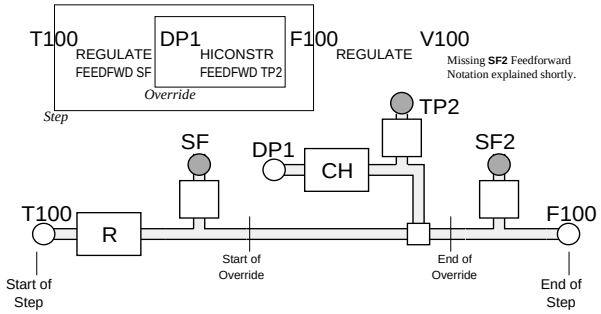
The later section on Idiom Loop statements addresses how the statement is translated to generate the succession of calls. This section addresses how these Idioms can be thought of as (user written) Tasks, permitting the user to write his own less standard forms. User defined Idioms are defined as simple Tasks, formulated with a final process connection argument List of two standard arguments representing, as above, the basic input (primary) and output (secondary) controlled variables, plus optional earlier argument Lists usable for including feedforward or parametric data, plus operating state definition argument Lists. The two arguments in the process connection argument List are treated specially according to internal rules of the Idiom Loop Statement. The remaining arguments will be treated directly, written in after the Idiom Keyword in the Loop statement (as in the expression **FEEDFWD TP2, TP3**).

In discussing the override Idioms and their actions, it will be convenient to refer to the override path or calculation (the fan-in path including the HiConstraint Idiom) and the direct path or calculation (the main path on which the override acts) whether or not the override overrides.

The discussion will require the following definitions for Step and Override:

Step. The interval in the Idiom Loop statement between left and right operands of a Main Idiom (Regulate and HiConstraint above). In the above figure, T100 to F100, P100 to F100, and F100 to V100 are all steps. The importance of Steps is that each can be interpreted independently of any following (or preceding) Step, developing a setpoint value for their right hand (output) operand and taking its measurement value to compute the external feedbacks of all of the Idiom calls within the Step.

Override. The interval from the start of an Override Idiom expression, including its left operand to the point where it has returned to the direct loop path.



The basic Idiom computation consists of two parts: a forward part defining the main control action, and a backward part providing the external feedback to each associated Block. Overrides involve a greater complication because their computation requires a temporary storage of forward calculation data developed in the direct calculation, a deferral of its own final forward action and data until any of its supporting Idioms (e.g. Feedfwd TP2 above) are completed, and similar suspensions in the backward calculation. The figure below illustrates the associated forward and backward computations and the passing of stored or deferred data.

```
STYRENE_PLANT                                    Page: Parameters

TASK() (): PRODUCT_A
  PERCENT_ACTIVE      21.0    %
  PERCENT_FILLER      70.0    %
```

or

```
STYRENE_PLANT                                    Page: Parameters

TASK() BLOCK BLOCK: PURPLE_LOTS
  LOT_NAME              PURPLE_LOT#?
  PRODUCT_QUANTITY      500.0    GAL
  PRODUCT_COLOR          RED
  UNIT_SIZE             1000.0   GAL
```

Note that the named Formulae can include several argument Lists together, and need not all divide the argument Lists the same way. This permits the expression of restrictions to the independent use of different combinations of Formulae. Above, the **PURPLE_LOTS** Formula requires not only its Lots parameters but can only be run with a particular unit (of the indicated unit size). The corresponding Parameters Page incorporates both argument Lists in its named Formulae. The **LOT_NAME** entry in the table defines a default name, but presumes that the operator may change it, as indicated by the following question mark, which will be superscripted in the listing.

A Recipe Call consists of the name of the Task[62] being used (, followed by any explicit argument Lists), followed by a colon and any intended Formulae names in any order, followed by additional argument Lists in appropriate order:

TASK: PRODUCT_A RED_LOTS BIG UNIT.

The Call would have a corresponding Parameters Page echoing the Call and then the Parameters Page contents of each of the Formulae, in order listed (simply echoing data stored on the individual Formula Parameters Page).

Discussion:

Under the **MAKE_RECIPE** command, each named Formula would create a corresponding editable Parameters Page, visible as long as the command is active. The Formulae are then listed on the Parameters Page of the associated Task, continuing to exist until explicitly deleted.

Each Call generates a Parameters Page including all its Formulae as defined above.

A Recipe Call consists of the name of the Task being used, followed by remaining Call argument Lists (argument Lists not corresponding to included named Formulae) in their definition order, followed by any intended Formulae names in any order, each (for generality) with optional dot and instance name, followed by any State values to be set.

Each Call generates a Parameters Page including all its Formulae as defined above, which is available as long as the Call itself exists (in the case of an operator entered command, as long as the Call is executing).

Included named Formulae must correspond to distinct definition argument Lists.

Additional Call argument Lists must be distinct from those others included or related to named Formulae.

The usual conventions about missing argument Lists and **BLOCK** Keyword argument Lists apply.

The result of this is that the Block Calls and Recipe Calls and simple Calls can all be combined in a programmed statement, for whatever perverted purpose one might imagine: A Block Call which also depended on online arguments, and on a separate Formula. Recall that Recipe Calls are not normally intended to be included in the control program.

The Block names included in the Call must match corresponding definition argument Lists, with no overlapping membership, and the Call must have implicit or explicit **BLOCK** Keywords corresponding to each of those Lists, with the exception below.

Only the last Block name may be previously undefined, and then only if there are unmatched and unfilled in definition argument Lists. In that case the remaining name, and the unmatched or unfilled argument Lists define a new Formulae (as if a control Block or **MAKE_RECIPE** command had been invoked). This usage is the unlikely combination of Recipe and Block Call.

Archival/Listing Format Relationships:

Block Calls have no special statement usages.

Archival/Entry Format Relationships:

Recipe Block Calls are normally treated as operator entered commands, entered temporarily on the Global Procedures Page, like all such commands, as discussed later. The statements must include appropriate reference scoping or Scope Prefixes, to locate the associated Tasks. The associated Formulae are entered permanently on the associated Parameters Page, but their execution is always linked to the associated Task wherever defined. ???and include the scoping data in its header? ???

Task Definitions for Idiom Calls

[62] Or of an Operation, when the Recipe permits the specification of a train or unit.

special keyword **<<<CONNECTIONS>>>**, instead of **<<<SEQUENTIAL>>>.**

The simple Block Call with a single Block name is able to set aside all Block data in a single (so named) Parameters Page Block. The default (initial) values of the Block data are copied from the definition argument List element's Definition Page values.

With a single Block name, a single Block is set aside which includes, consecutively listed, all data for all of the corresponding Task definition argument Lists (there may be more than one). If there is more than one name, all but the last must be already defined as described in the Recipe discussion below.

Archival/Listing Format Relationships:

In the Listing Format, the Block Call is recognized by the colon, name, and **BLOCK** Keyword or missing argument List, and associated Connection Activity. The Connection Activity is indented to the right and grouped closely to the initial line, to distinguish it from a simple unrelated Activity following the Call line.

Archival/Entry Format Relationships:

In the keyboard Entry Format, the first line of a Block Call consists of the algorithm name, the colon, the Blocks name, any instance name (with preceding dot), any argument Lists, **BLOCK** Keyword, any State names, and a final unmatched **(**©, initiating the associated Sequential Connection Activity. Repeated ©s have no affect on the Sequential Activity. While there is no intended need for this, an empty Sequence Activity can be used, indicated with its termination included **()**©.[60]

Block Calls to Support Recipes and Formulae

Definition: A Formula is a Block for which there is no Connection Activity (and generally no existing Block Call within the control Program). A Recipe Call is a Block Call with a missing Connection Activity.

Conceptual Role and Purpose: To represent named Recipes (which represent the combination of a Recipe procedure and its Formula or specification data).

Detailed Role:

In addition to the kind of call representing a control block with programmable connections integrated into the control program, there is a need for a similar kind of call, operator callable with previously defined (or defaulted) parameter blocks accessible for operation, to represent the recipe grade variants and formulae. It would differ from the usual Block Call in that it would run on demand or on schedule, rather than continuously as part of the control system. It would represent a defined set of static supervisory targets or parameters, which specified the production, rather than adjustable control targets. These values would be changed only by the operator, or by higher level supervisory systems, whenever current events required an accommodation in the static targets for a particular batch of product.

In its most basic role, the Recipe Call represents an operator initiated invocation of a previously parameterized Call; he makes the Call but the Formulae (parameterizations) of that Call have already been set up or defaulted (by a **MAKE_RECIPE** statement). The associated application envisions an indefinite number of such calls being set up beforehand, to express different Recipe variants run on the same process, under the same control system. It envisions that further variants will be added continuously over time.

The Recipe Call is based on the previously defined and named argument Lists. There may be several included argument Lists involved in the same call, some specifying the product grade targets, some specifying lot properties, some specifying process related parameterization. For example, a Task might have the following name and argument lists:

TASK(PERCENT_ACTIVE, PERCENT_FILLER) (LOT_NAME, PRODUCT_QUANTITY, PRODUCT_COLOR) (UNIT_SIZE).

Several particular sets of recipe formula may have been defined, for each definition argument List:[61]

MAKE_RECIPE TASK BLOCK() (): PRODUCT_A.
MAKE_RECIPE TASK BLOCK() (): PRODUCT_B.
MAKE_RECIPE TASK () BLOCK (): BLUE_LOTS
MAKE_RECIPE TASK() BLOCK (): RED_LOTS
MAKE_RECIPE TASK() () BLOCK: BIG_UNIT or **MAKE RECIPE TASK() (): BIG_UNIT**
MAKE_RECIPE TASK() () BLOCK: LITTLE_UNIT or **MAKE RECIPE TASK() (): LITTLE_UNIT**
MAKE_RECIPE TASK() BLOCK BLOCK: PURPLE_LOTS or **MAKE RECIPE TASK(): PURPLE_LOTS**

Each named recipe formula would create a corresponding editable Parameters Page, although several different named argument Lists may be indicated in a single statement, each creating its own Parameters Page.

---

[60] If the Sequence Activity is left off, the result is a Recipe Call (described next), which functions similarly to a Block Call but without the Sequential Activity connection behavior.

[61] The empty matched parentheses mark the positions of the argument Lists not included in the named recipe formulae.

Call is activated. If there are then more Call argument Lists than definition argument Lists (or one is explicitly a Block List), each distinct Call will have its own distinct (unnamed) Block data. The Block can be used to store the internal state of processing between re-entrant switchings, but it will not be re-initialized before the fresh invocation of any Call. State data can also be passed from a Call to the executing state of the Task, although the use of Call States and Blocks is largely the provence of Block Calls.

Typical Formats:

The Tasks and Activities Styrene Plant example contains many examples of Task calls without argument Lists. The Task with single argument List is analogous to the conventional language (BASIC, FORTRAN, C, or PASCAL) call.

Block Calls

Conceptual Role and Purpose: To represent the traditional Control Block with its operator accessible operational data.

Detailed Role:

There is a need for a kind of subroutine call corresponding to conventionally configured control blocks: which allows the associated definition of connection statements, and which derives the Call argument List data from an independent operator accessible table.

Block Calls are Calls with assigned names and Blocks (whether implied by a missing argument List or made explicit by the **BLOCK** Keyword) named and permitting passed State values. They allow the implementation of conventional control blocks in ICL. The Block is declared as a result of the Call along with the necessary connection statements, contained in an immediately following, associated Sequential Activity. The Block Call is further supported by its Parameters Page table. The Block Call consists of all of this as shown below. [Note: the PID Task (actually a built in algorithm) is assumed to have a single Block argument List, plus its state declarations.]

```
PID: STP + AUTO
  SET = STP.SET
  MEAS = STP.VALUE
  **
  XFB = OUT
  STV = OUT
```

It would be nice if connection statements could be executed both before and after the Call to the Task or algorithm. That is the purpose of the ****** statement. When the Block Call is executed, the connection statements before the ****** statement are executed first (if there is a ****** statement and there are connection statements before it), then the Task or algorithm, and finally any further connection statements.

Typical Formats:

The example above might (depending on Parameters Page) translate into Archival Format, as shown earlier, with the normal Activity Keywords:

<<<BLOCK CALL>>> PID <<<BLOCK>>>: STP + AUTO <<<PARAM 42>>>
<<<CONNECTIONS>>>
SET = STP.SET
MEAS = STP.VALUE
******
XFB = OUT
STV = OUT
<<<ENDSQ>>>

Keywords:

Entered in this order:

Mandatory: **<<<BLOCK CALL>>>**.

Optional: **BLOCK** or **<<<BLOCK>>>**. The **<<<BLOCK>>>** Keyword is set used to represent each implicit Block corresponding to unlisted final argument Lists. It is inserted automatically on archiving from a listing.

Discussion:

The Archival Format Block Call is initiated by the **<<<BLOCK CALL>>>** Keyword; followed by the name of the control algorithm; followed by optional Call argument Lists; including, whether or not other Call argument Lists are provided, either at least one **BLOCK** Keyword, or one or more missing final Call argument Lists, relative to the definition argument Lists and a corresponding number of Archival Format **<<<BLOCK>>>** Keywords; followed by a colon; followed by the Block name; optionally followed by a dot and instance name; followed by optional State names.

In the normal Block Call, the initial Call line is followed by a Sequential Connection Activity distinguished by the

[59] While the most elegant implementation would link all Call (and Block) references to Task Definitions without ever actually changing the corresponding Operation variables (in call by reference), the proposal currently makes no such assumption. For now, all variable data is assumed copied into as part of the call and left in the state computed by a call. X????X

normally represented by SFC looping.

Keywords:

The Entry, Listing, and Archival Formats would be identical to the Procedures Page Task formats except that the Keyword **<<<RECIPES>>>** would replace the normal Procedures Page Keyword **<<<PROCEDURES>>>**:

<<<PROCEDURES>>> STYRENE_PLANT
...

becomes

<<<RECIPES>>> STYRENE_PLANT
...

Task Calls in General

Conceptual Role and Purpose: Task Calls provide the general mechanism of representing the specified execution of shared computational or control procedure definitions.

Detailed Role:

Task Calls invoke the Task definition, applied to the data of the associated Operation and argument Lists. In the simplest case, the call consists of a single line statement containing the Task name, and Call argument Lists corresponding to the definition argument Lists. These Calls represent conventional computational Calls, lacking any mechanism for operator data storage and access. Block Calls represent calls which are supported by an operational (dot reference) accessible data Block, and associated connecting statements, permitting their operational usage to monitor and adjust controls. Recipe Calls permit the generation and filing of standard (recipe) data blocks to particularize a Batch control Task to a variant product grade.

Idiom Calls represent the means for letting the user define ICL Tasks in such a way that they can be linked into the system's Idiom Loop Statement software and used to provide user defined Idioms. Path Calls represent the normal Call, assigned a special role to re-configure the process connections to a particular (temporary) production operation, in a way that they are (for that duration) made secure from inappropriate manipulation by other ICL production Tasks, while posing minimum limitations to those other Tasks. These Calls are supported by several special commands which allow the unlimited creation of new Recipes, the archiving of particular control system states and parameter combinations, and the restoration of those states and combinations. While each of these forms have standardized roles, they permit arbitrary combinations and perversions: The computation of a Block Call may depend on a set of argument parameters outside the Block, etc.

Discussion:

The general Task Call contains the name of the Task, one or more argument Lists (which must match in maximum number, in order, and in element type the Task definition argument Lists), an optional colon and Block name [a Naming Postfix, applicable for Block and (allowing more than one name, separated by commas) Recipe Calls], one or more State names (accessible through dot reference for named or unique Calls) appropriate to the declared user or system states of the Task[58]. The detailed variants will be elaborated below under the more specialized usages.

A Call argument List may be a conventional List, matching its definition List and elements, or it may be replaced by the keyword **BLOCK** (representing a data Block to be set aside to contain the data in the arguments for operational access and storage between reentrant Calls). Any final missing Call argument Lists in a Call (compared to the number of argument Lists in the definition) are treated as **BLOCK** argument Lists.

As indicated earlier in the Parameters Page discussion, Calls define unique (Parameters Page) Blocks corresponding to explicit or implicit **BLOCK** keywords whenever the corresponding Block name is unique or absent from the call; different Calls sharing the same Block name also share the same actual Block and Block data. For unnamed Blocks its unique Block supports reentrancy. For named Blocks, each Call is a distinct Call to the same corresponding control function, with the same parameters.

Calls can be used to Operations in these cases:

- When the Call is not now operating; Operations cannot be re-Called before the completion of any previous Call to the same Operations.
- The arguments in Operation Calls must be other (sub)Operations.
- Operations cannot be Called in Block Calls (or Recipe Calls).

Simple Calls

Conceptual Role and Purpose: To represent the dedicated control tasks and traditional subroutine call usage.

Detailed Role:

The simplest simple Call consists of the Task name for a Task defined to have no argument Lists. It executes using whatever variable or Object references (usually to the Operations data and Tasks) are made within the Definition, acting on that data. Where there is one (or more) explicit argument Lists in both definition and Call (with no Block names), the definition references and data elements are linked up from the Call, for the duration of the activity of that Call.[59] Where there may be two or more re-entrant Calls active in parallel, this requires switching whenever each

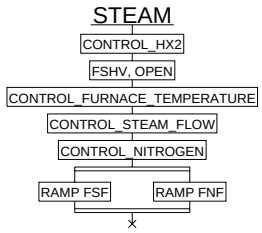
[58] The user States are assigned as a default value, the first named State in each subState declaration unless that subState receives a passed value in the Call. Either default or passed value is usable within the defined Called Task execution.

be intended principally for manual call from the Operators Work Station. In this light, it is expected that the Operators Work Station might support a simplified programming panel for generating these Recipe Tasks, if this capability was allowed. As an elaboration, this form would lend itself to compiling, in parallel, an illustrative Sequential Function Chart display, analogous to the analogous Idiom Loop Statement iconic illustration. The listings below, illustrates the kind of capability expected: simple control Task activation and Calling, simple process re-configuration, minimal parallel action, and automatically generated Sequential Function Charts:

STYRENE_PLANT

Page: Simple Procedures

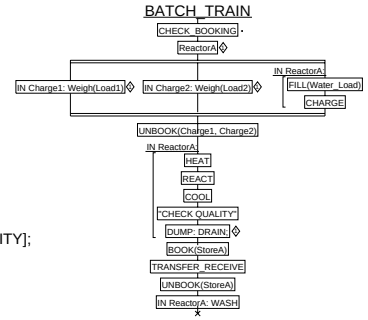
```
STEAM
    IN HEAT_RECOVERY: CONTROL_HX2
    IN FURNACE:
        FSHV, OPEN
        CONTROL_FURNACE_TEMPERATURE
        CONTROL_STEAM_FLOW
        CONTROL_NITROGEN
            RAMP FSF FROM 0 TO 50 CFS IN 30 MIN
            RAMP FNF TO 0 CFS IN 30 MIN
```



BATCH_PLANT. BATCH_TRAIN: [2]

Page: Simple Procedures

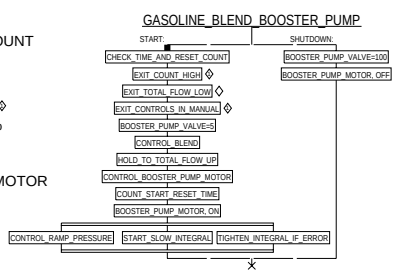
```
· CHECK_BOOKING·
◊ReactorA◊
◊   IN Charge1: Weigh(Load1)◊
◊   IN Charge2: Weigh(Load2)◊
    IN ReactorA:
        FILL(Water_Load)
        CHARGE
UNBOOK(Charge1, Charge2)
IN ReactorA:
    HEAT
    REACT
    COOL
    "CHECK QUALITY"; READY; [QUALITY];
◊   DUMP: DRAIN; END◊
BOOK(StoreA)
TRANSFER_RECEIVE
UNBOOK(StoreA)
IN ReactorA: WASH
```



GASOLINE_BLEND_BOOSTER_PUMP

Page: Simple Procedures

```
START
START:
    CHECK_TIME_AND_RESET_COUNT
◊   EXIT_COUNT_HIGH◊
◊   EXIT_TOTAL_FLOW_LOW◊
◊   EXIT_CONTROLS_IN_MANUAL◊
    BOOSTER_PUMP_VALVE = 5 %
    CONTROL_BLEND
    HOLD_TILL_TOTAL_FLOW_UP
    CONTROL_BOOSTER_PUMP_MOTOR
    COUNT_START_RESET_TIME
    BOOSTER_PUMP_MOTOR, ON
        CONTROL_RAMP_PRESSURE
        START_SLOW_INTEGRAL
        TIGHTEN_INTEGRAL_IF_ERROR
SHUTDOWN:
    BOOSTER_PUMP_VALVE = 100 %
    BOOSTER_PUMP_MOTOR, OFF
```



Restrictions:

The Tasks defined on the Recipes Page would be support only the following usages:

- At most one conventional argument List, all of whose arguments would reference single Attribute data with all its Contexts and with extended Context, and one State definition expression argument List.
- No use of the Task termination commands, except to distinguish normal and abnormal exits from called Tasks.
- Simple or Recipe Calls of any Task which did not include more than one argument List in the Call.
- Only Sequential and Parallel Activity Brackets could be used.
- No Parallel Activities nested within Parallel Activities and no more that a small number (4?) statements or Activities contained within a Parallel Activity.[57]
- Only Discrete Assignments to a named State, and Real Assignments to a constant state could be used.
- State Prefixes could be used, but without Local State Environment Declaration statements; only Global, System, and Operation States, or States declared in the argument List could be referenced.
- Scoping Prefixes are allowed within the body text.
- Theme Statements would be allowed.

Sequential Function Charts:

The examples illustrate the translation to Sequential Function Charts (SFCs). The translation would support the normal usages:

- Sequential Activities and their usages translate to vertically arranged steps on a single path.
- Parallel Activities translate to several vertical (Sequential) paths running between top and bottom pairs of horizontal doubled lines.
- Sets or alternative conditional cases (not illustrated above but normal represented in ICL by an initial Local State Environment Declaration followed by a set of State Prefixed statements) translate to several vertical (Sequential) paths running between top and bottom pairs of horizontal lines.

ICL usage suggests the following extensions to the Sequential Function Chart as illustrated above:

- Diamonds, asterisks, and dots, carried over from the ICL listing, to represent special exits or other statement/Activity relationships.
- The translation of Scoping Prefixes (as in **IN ReactorA:** above), and their nested Activities to the included Prefix in the Sequential Function Chart, with a vertical bracket delineating the set of SFC steps encompassed by the Prefix.
- State Driven Activities translated to several vertical (Sequential) paths running between top and bottom pairs of horizontal dashed lines, with each path labeled at top with its appropriate State Prefix. The thick vertical line from the top dashed line indicates the initial State path. This usage provides a more structured usage to represent many of the application situations

[57] Not only to simplify the diagram; but to fit it across the page!

Freezing prevents any change to the variable unless the variable is unfrozen or a **FREEZE** Prefix containing the variable as an argument precedes the associated command. The first form illustrated simply Freezes the value of the two variables (there can be any number of variables), the second form Freezes the value to the specified States or values. Both the **BOOK** and the **FREEZE** commands return States **_/FAIL**.[55] Both functions can be enclosed in a Local State Environment Declaration without lasting execution to test whether their potential execution was currently possible. When such a Local Environment is established, a command **EXECUTE**, within the sequence of statements separated by semicolons, will cause the enclosed Booking and Freezing actions to become permanent.[56]

```
[BOOK(UNIT1); FREEZE(V100, OPEN)];
WAIT NONE FAIL; EXECUTE
```

The value of Freezing is that it permits process elements like blocking valves to be set in tacit agreement with other independent processing tasks which may try to take the same action. The system reacts adversely only when the other task wants to set the element differently. At the same time it guarantees the other tasks that the element will not be changed. Both Booking and Freezing can be done directly in State assignments (e.g. **FNHV, FREEZE**; **FURNACE, BOOK**) except that the remaining Path Task characteristics will not then be invoked. Only the initial application of the Book function is considered to Book that element; later Book actions fail until the first one is undone. But every Task which Freezes an element (to consistent values) is assumed to participate in the Freezing; the Freezing is undone only after the last Freezing action is itself undone.

With this background, Path Tasks operate by taking any desired process re-configuration actions and setting up any desired control Tasks. At that point either of these commands may be used to restrict appropriate access to the resulting controls. Once these statements are encountered, the associated Booking and Freezing States will apply until separately undone (by analogous **UNBOOK**, **UNFREEZE** commands), or the Path Task has terminated. (The Task must remain active for the duration of the associated production task.) These commands can be nested in Activities several levels deep by using a similar notation to the **END** command asterisk and diamond, and **NEXT** command dot usage described earlier. For example the notation, `||·▌BOOK(FURNACE, REACTOR)·`, causes the Activities between the two dots to be grouped with the **BOOK** command; the Parallel Activity then becomes the effective Path Task, terminating the Booking implicitly only when that Task terminates.

On the other hand, if some conflicting Task succeeds in exercising the **OVERRIDE** Prefix capability or changing the Booking or Freezing State of some affected variable, then the Path Task itself is notified of that fact through its **_/OVERRIDE** State. The normal engineer's panel search function can then find which (**OVERRIDE**) statement then caused the Override. The system must support system calls which can return this same information, for operator notification and access.

Discussion:

A Path Task is any Task containing a **BOOK** or **FREEZE** statement.

An object Frozen of containing a Frozen element can be Booked, but a Booked object or its elements cannot be otherwise Frozen by an external element. Any already Frozen element, Booked or not, can be unFrozen by its Freezing elements. Thus if Booking takes place with respect to an externally Frozen element, the Booking element accepts the limitation imposed by the Freezing.

The archiving, listing, and entry of these statement forms is conventional ASCII/Keyboard based.

No Path Task may be Called (as an instance Modeled in a particular Operation) when it is already active.

A Path undoes its Booking actions on termination. Freezing is more complicated. A path undoes its Freezing actions on termination or under the UnFreeze command, but only with the termination or UnFreezing of the last Path Task to have Frozen that particular variable. The Task unFreezing is independent of any exercise of interim Freeze/UnFreeze State assignments to that variable.

A Frozen element cannot be Booked nor a Booked element Frozen, except by the Task which previously Froze it.

Simplified Task Format for Operator/Recipe Usage; Sequential Function Charts

Conceptual Role and Purpose: To represent and support procedures simplified for Recipe definition and operator usage.

Detailed Role:

This section describes a tentative proposal for a variant form of Task available for operator programming. Its explicit purpose is to allow simpler programming of Recipe level procedures using a subset of the language. At the same time it could be used to encourage maximum readability of high level outline application Tasks. In general, it would eliminate the computational aspects of the language, and include only the most readable procedural recipe forms. These Tasks would be included on a Simple Procedures Page. They could be called as normal Tasks, but would

[55] **_/FAIL** instead of the narrower States **BOOK/UNBOOK** and **FREEZE/UNFREEZE** allow simultaneous testing of both functions as above.

[56] In the special case where no Book or Freeze action would fail, the affected objects are held available (prevented from being Booked or Frozen) until an statement is encountered without semicolons.

asterisk can be moved to a selected State Driven Activity by repeated carriage returns (and **<<<++>>>** Archival Format notation) as with the **END** commands. The actually chosen Bracket is determined cyclically by those repetitions. The **NEXTSTATE** command only applies to Activities, not from Task Calls.

Typical Formats:

Shown above.

Keywords:

END, **END1**, **END**2, …, **<<<END∆>>>**, **NEXT**, **NEXTSTATE**, **<<<+>>>**, **<<<++>>>**, … .

Discussion:

The general Archival Format **END** statement consists of the Listing Format keyword **END**, optionally including (without separating space) a number, optionally followed by an off-line Archival Format triple bracketed keyword including one or more **+** signs; or of the Listing Format keyword **END**, followed by **<<<∆>>>**.

The Archival Format **NEXT** statement consists of the **NEXT** keyword optionally followed by an off-line Archival Format triple bracketed keyword consisting of the brackets and one or more **+** signs.

The Archival Format **NEXTSTATE** statement consists of the **NEXTSTATE** keyword optionally followed by an off-line Archival Format triple bracketed keyword consisting of the brackets and one or more **+** signs.

The usages are nearly identical, and except for the numbered States largely equivalent. the user need not learn the nuances of the asterisks, periods, and diamonds until well after he has fully used them. Their inclusion is largely there for visual emphasis of the different cases.

Archival/Listing Format Relationships:

The key behavior is the use of repeated carriage returns to cycle through the legal cases. This usage allows only legal cases to be selected by the user. This cycling includes the cycling of position of the asterisks or periods and the selection of diamonds (abnormal termination) vs. asterisks (normal termination) for a left most symbol.

Note that the number entered after the **END** command, and included in the Archival Format does not become part of the Listing Format **END** keyword. Instead it is incorporated into the paired diamond symbols.

Archival/Entry Format Relationships:

END statements are entered directly in Entry Format as **END**©. When appropriate, the **END** keyword may be entered with a directly following number (**END1**©). The asterisk or numbered diamond is automatically positioned before the right most Activity Bracket in the listing. Repeated carriage returns then cause the asterisk or diamond to cycle to earlier Brackets, and then (if an asterisk) if before the left most Bracket to change to a diamond, and then to recycle back to the original position as the original icon.

At the end of a sequence of position changing ©s, the entry of a further statement, or the initiation of any other edit action, terminates this selection and causes the added statement to be included on a new line (starting a fresh line in the edit area).

On edit, the statement can be selected. The carriage returns are not part of the edit output; they are encoded in an internal state and represented by the asterisks or diamonds, and their positions. Repeated carriage returns can then be used to change the termination State as before.

Path Tasks

Conceptual Role and Purpose: To represent and manage temporary combinations of processing equipment and associated setups or controls.

Detailed Role:

Path Tasks (or Operations) represent a special kind of Task(or Operations), distinguished by their use of the **BOOK** or **FREEZE** statements. Their purpose is to control process configuration so that low level conflicts between independent production tasks are avoided. They are used whenever a process is to be reconnected or re-configured for control using shared process equipment or variables. They operate using Booking and Freezing. An Operation, Block, or Variable can be Booked by a BOOK statement:

BOOK(FURNACE, REACTOR)

This command sets the **BOOKED/UNBOOKED** State of its arguments (there can be any number of arguments.). When this happens, no other Operation may change any variable, parameter, or State (including the **BOOKED/UNBOOKED** State), or Call any Task within the Booked object, unless the associated command contains an **OVERRIDE** prefix naming that Operation as one of its arguments:

OVERRIDE(FURNACE, REACTOR): FNHF, CLOSE

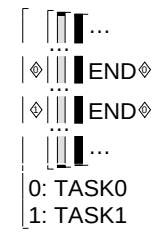
Booking is used to protect the controls of an Operation when they must be changed unpredictably. A process variable can be frozen by a **FREEZE** statement:

FREEZE(FNHV; FSHV) or **FREEZE(FNHV, OPEN; FSHV, OPEN)**

simplest form of the **END** command (illustrated contained in two Sequential, one Parallel, and one Continuous nested Activities) combines the **END** command with a highlighting pair of asterisks: |||*|END*. The paired asterisks delimit the command and Bracket of the terminated Activity. This case indicates the termination of the lowest level (Continuous) Activity; that Activity containing the command. The corresponding Archival Format for this statement would be: **END**. The command would be entered by typing **END**© (like any other statement terminated by a carriage return[54]). The simplest form of a Task call, so terminated, simply exits on completion, as with a normally exited Sequential Task or any other simple statement.

There are additional needs for controlling the termination of Activities and Tasks:

- Provision for termination of nested containing activities. This is supported by moving the asterisk (to its left) so that it includes all of the intended terminated Activities (i.e. their Brackets): ||*||END*, |*|||END*, etc. The corresponding Archival Formats are **END <<<+>>>**, **END <<<++>>>**, etc. They would be entered as **END**©©, **END**©©©, etc. If the asterisk is moved all the way to the left most position (*||||END*), the result represents the normal overall Task exit (its Archival Format is **END <<<+++>>>**; its Entry Format is **END**©©©©).
- Provision for distinguishing between a normal and an abnormal or otherwise distinguished exit from a Task (or Operation). This form of exit is indicated with a diamond instead of an asterisk: ◊|||END◊. Except for distinguishing the underlying exit condition, the function is the same as *|||END*. The corresponding Archival Format is **END <<<Δ>>>**. It would be entered as **END**©©©©©©. Further ©s (carriage returns) would cause the asterisk position and diamond forms to recycle.



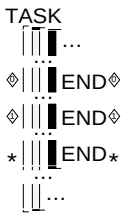
- Provision to further distinguish multiple exit conditions, by passing an exit State that can be tested upon exiting. This is done by including a number with the **END** command (within the diamond): ◊|||END◊, ◊|||END◊. In this case, the zero numbered diamond corresponds to the normal (asterisked) exit. The full usage might appear as shown to the right.

  In this case, the inner nested Activities have two exit conditions, numbered 0 and 1. On exit, the two State Prefixed statements select whether to execute **TASK0** or **TASK1** based on which exit State is returned. The corresponding Archival Format (including the three nested Activities being terminated) for the two **END** statements is **END0 <<<++>>>** and **END1 <<<++>>>**. Their keyboard entry would be: **END0**©©© and **END1**©©©.

- Provision to extend the termination of Tasks so that their Task Calls can controllably terminate containing Activities or pass exit States. This gives rise to several cases:
  - Simple Calls without exit controls: The Task exits normally, whatever the exit condition, like a normal statement. The Archival Format and keyboard entry are the same as any statement.
  - Simple Calls with both normal (indicated by a Task definition terminating **END** statement with asterisk) and abnormal (indicated by a second Task definition terminating **END** statement with diamond [with or without State numbers]) conditions: In this case, a corresponding pair of diamonds (with numbers included only if there is a single numbered State) is positioned, one after the Task Call and one to the immediate left of the Bracket, corresponding to the highest level containing Activity to be terminated.

    The normal Task termination causes the Task to exit as with the previous simple Call. An abnormal (or zero numbered diamond) exit condition causes the Task to terminate along with all of the Activities up to the marked activity. Thus, in the example to the right, the Task has State exits numbered 0 and 1 and a normal exit. In the calling Activity the normal **TASK** exit terminates to the Parallel Activity, whereas the numbered exits both terminate all Activities up through the marked Sequential Activity. Following the terminations, the exit States can then be used to select the **TASK0** or **TASK1** through the corresponding State Prefixes.

    The Archival Format for the Task call (covering both normal and abnormal terminations) is analogous to the corresponding **END** statement, in this case being: **TASK <<<++>>>**. The corresponding keyboard Entry Format is: **TASK**©©©.



- Provision to cause a containing Activity to continue processing as if a nested Activity (or set of nested Activities) had terminated, allowing the nested Activity to complete its function, in parallel, in its own good time. This is carried out by the **NEXT** command. The usage is similar to the simple **END** command usage, as shown: |||·|NEXT·, and entered as **NEXT**©. The paired dots take the place of the asterisk in the **END** command usage, and they can be moved over to encompass several Activity Brackets as with the **END** command usage: |·||NEXT·. The usage ·|||NEXT· makes no sense, except in a Task definition, where it would indicate that the Task returned control in parallel to the calling Activity while continuing to operate. The corresponding Archival Formats are: **NEXT**, **NEXT <<<+>>>**, **NEXT<<<++>>>**; and the keyboard entries are: **NEXT**©, **NEXT**©©, **NEXT**©©©.
- Activities (or Tasks) operating inside one or more State Driven Activities can cause one of those Activities to advance its State (to the next State listed as a State Prefix State) by using the command **NEXTSTATE**. The system generates a pair of asterisks as before, positioning the left most one to the immediate right of the affected State Driven Activity as in the above (Tasks section) Styrene Plant example. Such a command inherently terminates all included Activities, since it terminates the current State. The

[54] Recalling that © represents a carriage return and Δ represents a space.

STEAM
- IN FURNACE:
  - CONTROL_STEAM_FLOW
  - CONTROL_NITROGEN
    - RAMP FSF FROM 0 TO 50 CFS IN 30 MIN
    - RAMP FNF TO 0 CFS IN 30 MIN

Operation/Task/Activity System States

The earlier Operations Page showed System States for the Operation. The Activities and Tasks will generally have similar built-in System States. These States control the normal operation of the particular element. The States can also be viewed as Operating Modes. The Operating Modes or States shared by all such elements are:

Operating Modes:

Mode	Theme t'mnt State Code	Activity State	Meaning
RUN	1	≠0	The Activity or Related Form is fully running.
SUSPEND	2	≠0	The Activity suspends any sequenced processing until Continued or Aborted. At the same time any process I/O and regulatory control is continued. In nesting or nested Activities or statements SUSPEND overrides RUN.
MANUAL	3	≠0	All automatic control (but not the process I/O) within the Activity ceases. In nesting or nested Activities or statements MANUAL overrides SUSPEND or RUN.
CONTINUE	4	≠0	The Activity retransitions from the SUSPEND (or MANUAL) mode back to the RUN mode.
AUTO	6	≠0	A Manual Activity retransitions back from the MANUAL mode to the SUSPEND mode.
ABORT	7	≠0	The Activity is prepared to terminate; the Activity then transitions to the END mode.
END	8	=0	A top level (unnested) Activity or Statement is terminated and completely inactive.

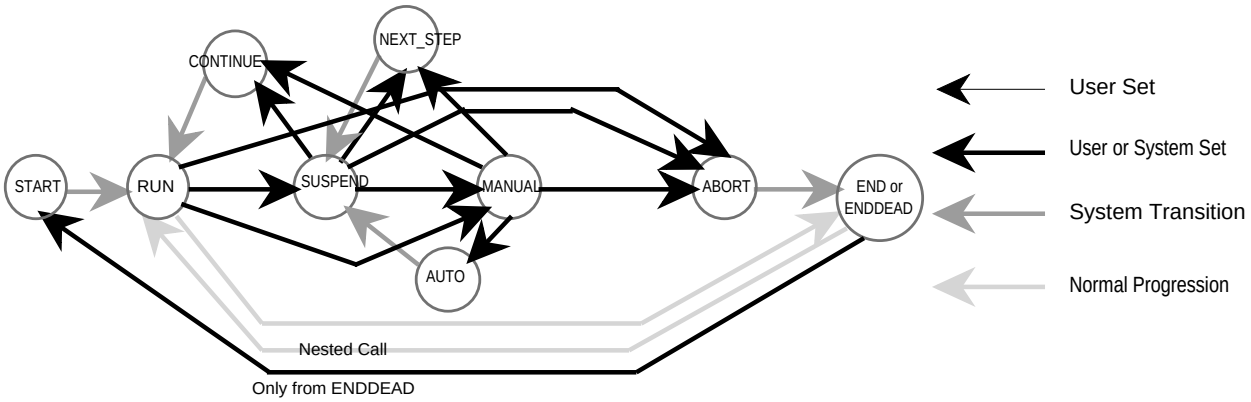
This initial State is needed for Tasks:

NEXT_STEP	5	≠0	The Activity executes one step (sample time) in RUN mode from SUSPEND (or MANUAL) mode and retransitions to the SUSPEND mode.

These States have been added for small system Tasks, where the extra states are needed to differentiate from prior States not supported by internal State differentiators:

START	0	=0→≠0	The Activity or Related Form is set (by the user or system) to initiate running; it will then transition to RUN as part of the processing. As indicated in the figure this State can be entered manually only from the ENDDEAD State (by an independent or un-nested Activity).
ENDDEAD	9	=0	A nested Activity or Statement is terminated and completely inactive. Its purpose is to distinguish the terminated nested Activity or statement and prevent their manual transition to START.

The State Diagram defines the State Transitions for these States:



Activity/Task Termination Commands

Conceptual Role and Purpose: To support the flexible termination and continuation from nested Activities and Tasks.

Detailed Role:

Sequential Activities terminate naturally. The other Activities sometimes need explicit commands for terminating like the above **END** command. Non-Sequential Activities also complicate the coordinated termination of nested Activities. The

```
INTRM = Remainder
...
–: ERROR = –ERROR
INITIALIZE: INTLG = XFB–ERROR/PB
AUTO: OUT = (ERROR+INTLG)/PB
ENDSQ
ENDP
```

Keywords:

Tasks and Activity statements, names, and argument Lists require no special Keywords. The Activity Bracket Keywords are entered in this order:

Initial Activity Bracket Keywords: **<<<SEQUENTIAL>>>**, **<<<LOOPING>>>**, **<<<PARALLEL>>>**, **<<<CONTINUOUS>>>**, **<<<STATE_DRIVEN>>>**.

Terminal Activity Bracket Keywords: **<<<ENDSQ>>>**, **<<<ENDLP>>>**, **<<<ENDPL>>>, <<<ENDCT>>>**, **<<<ENDSD>>>**.

Discussion:

An Activity definition is initiated with one of the Initial Activity Bracket Keywords, followed by any number of statements or included Activities, followed by the corresponding Terminal Activity Bracket Keyword. A Task definition is an Activity definition, directly preceded (on the preceding line) by a (unique) name, itself optionally followed by one or more argument Lists, which may be parenthesized State definition expressions. There may be multiple argument Lists, or equivalently, nested List arguments.

As indicated earlier, *Single Statement* and *Null Statement Tasks* can be defined without Activity Brackets, for special usages.

Operations used in Task Calls???????.

Archival/Listing Format Relationships:

The matched Activity Bracket keywords define the beginning and end of each Activity. The corresponding Bracket is drawn to delineate the corresponding statements and nested Activities. These statements and Activities are displayed displaced to the right sufficiently to allow for the display of the Bracket and of any interposed diamond, period, or asterisk termination symbols.

On the Listing Format Procedures Page, the initial unnamed Activities and consecutive Tasks are separated (and each Task name and body is grouped) sufficiently to distinguish them as independent Tasks definitions rather than as intermixed statements and Activities.

Archival/Entry Format Relationships:

Tasks and Activities are entered on the Procedures Page.

A Task definition may not occur nested within some other Task or Activity.

The normal statements within the Activities are entered in directly and edited as text. The engineer's work station panel described later includes an edit text entry window into which all text is entered or edited (in Entry Format), this is translated into Listing Format in a listing display.

An Activity (and its Bracket) is initiated by entry of an unmatched **(**©. The result is the start of a Sequential Bracket. Repeated entry of © (carriage return) causes the Bracket choice to cycle (from Sequential) to Looping, to Parallel, to Continuous, to State Driven, and back again.

The first line of a Task definition consists of the Task name, the argument Lists, and a final unmatched **(**©, including any additional ©s which act in the same way as with Activities, to specify the basic Activity Bracket associated with the Task:

```
STEAM
┌
```

At the end of a sequence of Bracket changing ©s, the entry of a further statement, or the initiation of any other edit action, terminates this Bracket selection and causes the added statement to be included on a new line (continuing after the start of the Bracket in the Listing Format, starting a fresh line in the work station edit area).

Whenever statements are entered within an Activity (or Task), the accumulated vertical Bracket components are included in the listing before the statement text, automatically.

```
STEAM
┌IN FURNACE:
│┌CONTROL_STEAM_FLOW
│CONTROL_NITROGEN
│└RAMP FSF FROM 0 TO 50 CFS IN 30 MIN
```

The bottom of an Activity is defined by entering in an unmatched **)**©. The bottom of the Bracket is then drawn on the listing and the next line positioned to. The edit area of the work station is cleared for a new line.

```
STEAM
┌IN FURNACE:
│┌CONTROL_STEAM_FLOW
│CONTROL_NITROGEN
│└RAMP FSF FROM 0 TO 50 CFS IN 30 MIN
└RAMP FNF TO 0 CFS IN 30 MIN
```

Multiple Activities may also be terminated together with an unmatched **)...)**©, the number of **)**s matching the Activities.

<<<PROCEDURES>>> STYRENE_PLANT
<<<SEQUENTIAL>>>
[FURNACE; REACTOR; HEAT_RECOVERY; FEED_TANKAGE; SEPARATOR];
ACTIVE: "END ALL OPERATIONS"; END <<<END>>>
<<<PARALLEL>>>
<<<CONTINUOUS>>>
(CONSOLE_SHUTDOWN)
<<<ENDCT>>>
<<<STATE_DRIVEN>>>
STARTUP
STARTUP:
<<<SEQUENTIAL>>>
PREPARE_STARTUP <<<END 3>>>
PURGE <<<END 3>>>
STEAM
REACT <<<END 3>>>
NEXTSTATE <<<END>>>
<<<ENDSQ>>>
HOLD:
<<<PARALLEL>>>
(FURNACE, REACTOR, HEAT_RECOVERY, FEED_TANKAGE, SEPARATOR) <<<END 3>>>
<<<CONTINUOUS>>>
"READY TO RUN? SET FEED RATE."; READY; NEXTSTATE <<<END>>>
<<<ENDCT>>> <<<ENDPL>>>
RUN: (FURNACE, REACTOR, HEAT_RECOVERY, FEED_TANKAGE, SEPARATOR) <<<END 2>>>
EMERGENCY_SHUTDOWN:
<<<SEQUENTIAL>>>
PREPARE_EMERGENCY_SHUTDOWN
EMERGENCY_STEAM_SHUTDOWN
FINAL_SHUTDOWN <<<END 3>>>
<<<ENDSQ>>>
SHUTDOWN:
<<<SEQUENTIAL>>>
SHUTDOWN_FEED
SHUTDOWN_TEMPERATURE
SHUTDOWN_STEAM
FINAL_SHUTDOWN <<<END 3>>>
<<<ENDSQ>>> <<<ENDSD>>> <<<ENDPL>>> <<<ENDSQ>>>
<<<ENDP>>>

In the following sections, reference is again made to a Details Page PID Block Call. The earlier Details Page and Parameters Page discussion refer to this structure. While the PID definition would normally be built in, if it were instead user defined, the definition might take the following form with argument List:

GLOBAL Page: Procedures

PID((SET), (MEAS), (OUT), (XFB), (PB), (INT), (DER), (INTLG), (INTRM), (DERLG), (DERRM))
(+/–, MANUAL/AUTO)

ERROR = SET - MEAS
...
INTLG = (XFB*INT+DelT*INTLG+INTRM)/(INT+DelT)
INTRM = Remainder
...
– : ERROR = –ERROR
INITIALIZE : INTLG = XFB–ERROR/PB
AUTO : OUT = (ERROR+INTLG)/PB

For future discussion, the algorithm includes Auto/Manual action, loop gain sign (**+/–** instead or **Increase/Decrease**), and the ability to respond to the system Initialize State. The corresponding Archival Format is

<<<PROCEDURES>>> <<<GLOBAL>>>
<<<TASK>>>PID((SET), (MEAS), OUT), (XFB), (PB), (INT), (DER), (INTLG), (INTRM), DERLG), (DERRM)) (+/-, MANUAL/AUTO)
<<<SEQUENTIAL>>>
ERROR = SET – MEAS
…
INTLG = (XFB*INT+INTLG+INTRM).(INT+1)

Sequential Activities that never execute their final statements, without causing the pointless immediate termination of final non-terminating statements. The caveat applies only to Sequential Activities.]

[XXX Problem: Non-terminated Statements or Activities following a Statement Prefix. In a Sequential Activity these are treated as conditional executions, ignored if the test fails, continued indefinitely if the condition once succeeds (and continuing any Sequential Activity in which they occur as the last Statement/Activity). But in a Continuous (or State Driven) Activity, where the Prefix is continuously tested, the non-terminating Statement/Activity is activated and Aborted appropriately whenever the Prefix first succeeds or first fails. XXX]

- Looping Activities execute their delineated statements and Activities, starting at the topmost one as in the case of Sequential Activities. But on completing the execution of the last or lowest statement or Activity, they return to executing their first or topmost one, continuing indefinitely until explicitly terminated with an **END** command. Successive loopings to the Activity always start in the next sample time calculation.
- Parallel Activities execute their delineated statements and (nested) Activities, in parallel. The Activity terminates normally after the completed execution of all of its delineated statements or Activities.
  [In fact, each delineated statement or Activity is executed to completion, or suspension, sequentially within the first sample time. Thereafter, suspended statements or Activities are tested in their suspension conditions and execution resumed when permitted, again sequentially within the sample time. The Parallel Activity terminates in that sample time, with its last executing delineated statement or Activity.]
- Continuous Activities execute their delineated statements and Activities, repeatedly each sample time, and in parallel. Normally this covers the function. However if a delineated statement or nested Activity suspends in the middle of execution (other delineated statements and Activities continue normally) it is tested for resumption each succeeding sampling time and resumed when appropriate. After its normal termination it is re-executed in the following sample time, repeating indefinitely, until the Continuous Activity is explicitly terminated.
- State Driven Activities independently execute all those delineated statements or Activities, not prefixed by a State Prefix (in parallel, as in the case of Parallel Activities). Whenever a system or user defined Operation State (including the State of a parent Operation) changes to a State included in one or more of the State Prefixes preceding a delineated statement or Activity, those statements or Activities are executed independently (in parallel, as in the case of Parallel Activities). At the same time, whenever the States which have caused delineated State Prefixed statements or Activities to execute cease to apply (or in the case of un-Prefixed statements or Activities, whenever the State is changed to permit execution of State Prefixed statements or Activities) the affected statements or Activities are aborted (executing any nested **ABORT** Prefixed statements or Activities before terminating). Any continuing abort related activities continue independently of the rest of the execution. The abort activity for a State Driven Activity starts in the same sample time that the change in State is recognized. The initiation of the new State activity starts in the following sample time, unless that State has been terminated when it comes time to start execution.

When an Activity is terminated for any reason, any internally nested statements or Activities are aborted (executing associated **ABORT** Prefixed statements or Activities before terminating). Thus, when any Activity terminates, any still active Task Call (defined below) also terminates. But if the Task is to be reCalled in the next sample time, then instead the Call continues as if never terminated.

[????Problem: How to identify these continuations: Use Qualifications: no argument Lists, explicit keywords [**CALL TASK()**, **CONTINUE TASK()**, …]? **????]**

Tasks are named Activities. The name is included on the line above and before the top of the Activity Bracket, as illustrated by the CONTROL_FEED Task example earlier in the Procedures Page discussion. A Task without arguments is essentially a standard operation applicable to the Operation. But Tasks can have arguments, to parameterize that standard operation, or permit it to act as a generic model for a number of distinct control objects. Following the Task name in a Task definition, on the same line, one or more argument Lists may be placed. As developed further in the section on Lists, the argument List may be a parenthesized list of named items, or it may be a parenthesized State definition expression. In the latter case, the defined States are added to the built-in States of the Task itself. These States are associated with the Task Call, accessible by dot reference in the case of Named Calls, or when there is only a single (unique) implied Task Call (within a given Activity, and no explicit Task Calls).[52] Inside the Task definition, the State names may be used in State Prefixes.

An argument List may also consist of the single word **ALL**, meaning that it includes data spaces for all of the Task referenced values not already included in some other argument List. Every (definition) List argument or element must be defined within the Operation, as representing a defined unambiguous value or object (a SuperVariable element, a Task, named Call, or (sub)Operation).[53] As described in greater detail, a normal argument List can be preceded by a name and a colon to provide a name to the List for internal reference.

Typical Formats:

The above Listing Format example has the following corresponding Archival Format listing:

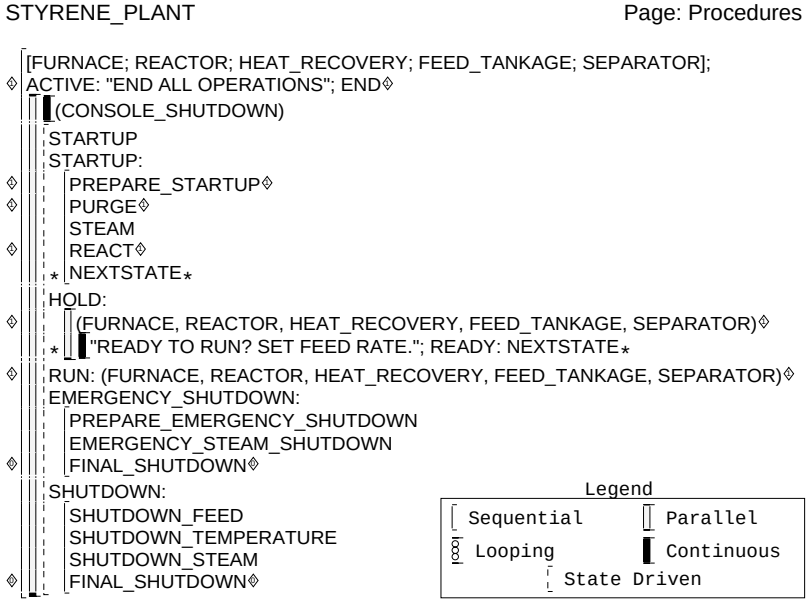
[52] Task Calls and State references should not normally be mixed in a program particularly when the Calls are re-entrant, because of the resulting reference confusion. However Task States can always be referenced and changed within a Task definition. Further, Task Calls unique within an Activity may be referenced as to their State within that same Activity. And uniquely named Task Calls (Block or Recipe Calls) can always be referenced for their States.

[53] But capable of representing some similar object, either as a SuperVariable modeling a similar SuperVariable or constant, or a Task or Operation modeling one of several modeled Tasks or Operations.

ICL can group statements into executing lists in two ways: by use of semicolons, and in Activities and Tasks. Statements separated by semicolons can be grouped on the same line or on consecutive lines.[50] Statements grouped on a same line are executed sequentially. Embedded statements which suspend or run for several sample times cause the rest of the line to wait their completion before continuing, as with any other kind of sequenced execution. In this sequenced interpretation, State Prefixes encountered at the start of a line, or after a semicolon are interpreted as setting off a conditionally executed statement. The failure to meet the condition is viewed as the completion of the remaining line, not its suspension.[51]

Consecutive lines separated by semicolons are executed sequentially, but each line is initiated independent of the suspension of the previous line; the line executions are thus ordered in the same way as the nested elements of the Parallel Activity below. The underlying purpose of the semicolon is to group statements subject to some related data environment. Usually this is the set of States that affect their conditional execution.

As illustrated in the earlier examples, Activities and Tasks define the working computations and their computational order. An Activity is a sequence of statements or nested Activities delineated immediately to its left by a Bracket, as illustrated below. While one might imagine entering the positioned iconic and asterisk symbols in some way as direct keyboard symbols in the Entry Format, this is quite awkward; these symbols are instead controlled by using repeated entry of the terminating carriage return to cycle their choice and position. The Bracket entry will be described at the end of this section; the asterisk diamond positioning will be described in the next section.



This Activity represents the main Activity, for the Styrene Plant example, invoked whenever that Operation is called or activated. The legend defines the five kinds of computational orders for the delineated statements:

- Sequential Activities execute their delineated statements and (nested) Activities, sequentially, starting with the topmost one, each to completion before the next. The Activity terminates normally after the execution of its last or lowest delineated statement or Activity.

  [The Sequential Activity may execute to completion in a single sample time depending on its delineated statements or Activities. Alternatively, any individual statement or Activity may suspend in the middle of its execution, suspending the execution of the Activity as a whole. In that case, each succeeding sample time will cause the testing of the suspension conditions until the delineated statement or Activity can resume execution, and with it the Sequential Activity.]

  [Nested Activities (in a Sequential or Parallel Activity), which have no programmed termination and are not the last/lowest nested Activity in a Sequential Activity, allow the continued execution of following statements or Activities once they have been initiated. This includes the following kinds of non-terminating Activities:

  - Idioms or other Continuous/Continued Statements.
  - Continuous, Looping, or State Driven Activities, without explicit **END** commands exiting them directly (not to a higher level Activity).
  - Sequential Activities, with terminal non-terminating Activities, as defined here, or terminal **CONTINUED** commands.
  - Parallel Activities with nested non-terminating Activities, as defined here.
  - Calls to Tasks based on the above kinds of Activities, not normally terminating, and without normal **END** command conditions (**END 0** or **END** with an asterisk), exiting the Task directly (not at a higher level Activity).

  Note the recursive character of the non-terminating items so defined. The purpose of this caveat is to prevent the definition of

[50] Semicolons are also used to group phrases in Theme Statements and other kinds of listed language items. Each of these forms has its own special usage

[51] If a conditional execution is intended to include a wait for that condition, this must be explicitly programmed. The **WAIT** (Theme) Statement represents one way of waiting until an intended condition has been met.

STYRENE_PLANT Page: Definitions

NAME	IN	MIN	MAX	UNITS	VALUE	SET	HI	LO	DEV
FST	1	0	800	DGC	_	_	780	300	20
TRF	2	0	100	GPM	_	_	_	_	20
TFF	3	0	100	GPM	_	_	_	_	20
TF	_ †1	0	100	GPM	_	_	_	_	20
POL	4	0	300	FT	_	_	30	10	20
WOL	5	0	300	FT	_	_	30	10	20

The corresponding Comments Pages and entries would be numbered as in the case of the Details Page. This would be most visible with respect to the Archival Format, since the Listing Format behavior could be linked naturally without explicit reference to the numbers.

Typical Formats:

The corresponding Definitions Page entry would take the Archival Format:

<<<ENTRY>>> TF _<<<COM 12.1>>> 0 100 GPM _ _ _ _ 20 ;analogous to the similar Footnote entry.

The Comments Page might look as follows in the Listing Format:

STYRENE_PLANT Page: Comments

†1 THIS VARIABLE CORRESPONDS TO THE TOTAL FEED FLOW.

The Comments Page might look as follows in the Archival Format:

<<<COMMENTS>>> STYRENE_PLANT. DEFINITIONS_PAGE. <<<PAGE 12>>>
<<<COMMENT 1>>> THIS VARIABLE CORRESPONDS TO THE TOTAL FEED FLOW.
<<<ENDC>>>.

The Comments Pages would be numbered, Comments to Definitions Pages first, then Comments to Procedures Pages, then Comments to Details Pages.

Keywords:

Mandatory: **<<<COMMENTS>>>**, **<<<PAGE @>>>**, **<<COMMENT #>>>**, **<<<ENDC>>>**, on the Comments Page, with the @ sign replaced by the Page/statement number, and the # sign replaced by the comment number as discussed under the Details Page; **<<<COM @.#>>>** as the corresponding reference code on the associated Definitions or Procedures Page.

Discussion:

Within an Operation, all subOperation, Task, and State Names must be mutually unique.

Archival/Listing Format Relationships:

Comments are listed on the Comments Page in order of their Page/statement and comments number. Both numbers are automatically generated under online configuration.

Archival/Entry Format Relationships:

The † symbol is entered in manually in the appropriate positions on the appropriate Pages.

The Comments Page is selected, for display, by a panel button. Text comments can then be entered manually for each reference.

Simulations Page

Conceptual Role and Purpose: To support process simulations in association with their controls.

Detailed Role:

The Simulations Page is identical with respect to configuration and programming, but for its name, to the Procedures Page. It is inactive except under the Simulations mode. Under that mode, it executes exactly like the Procedures Page (and with it). Under that mode, the system can be taken through the same subModes, as under the Operate Mode. But the process I/O is not then enabled. This allows a process simulation to be incorporated on that Page, reading and driving the process I/O variables instead of the I/O. The Simulations page incorporates only the existing ICL statements, making no other concessions to simulation. It is expected that only the simplest kind of operational simulation will be needed for program shakedown.[49]

Tasks and Activities; Tasks with Arguments

Conceptual Role and Purpose: To represent combined or named computations and controls.

Detailed Role:

[49] The system might also support a standardized software connection to an external simulation package.

Entered in this order:

Mandatory: **<<<SUMMARY>>>**, **!**, **!!**, **<<<ENDS>>>**.

Optional: **<<<ENTRYS>>>**, **EXPRESSION**, **CALL**, **MESSAGE**, **<<<TREND>>>**.

Discussion:

The Summary Page is initiated by the **<<<SUMMARY>>>** keyword. It contains one or more entries, and is terminated by a **<<<ENDS>>>** keyword.

The **<<<SUMMARY>>>** keyword must be immediately followed by a legitimate Operation reference, for the Operation containing the Summary Page.

Each simple entry is initiated by an **<<<ENTRYS>>>** keyword and consists of an appropriate sequence of items, up to the next **<<<ENTRYS>>>** or **<<<ENDS>>>** keyword.

Each Trend entry is initiated by a **<<<TREND>>>** keyword, and followed by the identical sequence of items as would apply for a simple entry.

A single variable entry includes the variable name, its value, and, (optionally on input, always on output) its units.

Multiple entries for the same variable append an occurrence number after a dot after the variable name.

An expression lists the **EXPRESSION** keyword, followed by the parenthesized expression, followed by the calculated value.

Multiple expression entries append an occurrence number after a dot after the **EXPRESSION** keyword.

A single Call entry includes the **CALL** keyword followed or the Task name.

A single Block entry includes the Block name.

Multiple Call entries to the same Task are numbered with a dot and occurrence number after the Task name.

Multiple Block entries to the same Block are numbered with a dot and occurrence number after the Block name.

A single Message entry includes the **MESSAGE** keyword, followed by the quoted Message, if the Message has occurred, or a Blank otherwise.

Multiple Message entries are numbered with a dot and occurrence number after the keyword.

On output (and in the Listing Format), all variables are listed first, followed by expressions, followed by unnamed Calls, followed by Blocks, followed by Messages.

All named variables and Calls are listed alphabetically, by their variable, Task, or Block name.

Trend entries create an appropriate recording of the necessary data.

Archival/Listing Format Relationships:

The **<<<SUMMARY>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

The **<<<ENTRYS>>>** or **<<<TREND>>>** keywords together with **EXPRESSION**, **CALL**, and **MESSAGE** keywords identify the corresponding line entries in the Listing.

Simple entries are listed as described.

Trend entries are listed graphically, as an analog trend for Real data, as a step trend for Discrete data, and as event marked time lines for Calls and Messages.

Archival/Entry Format Relationships:

The **!** and **!!** symbols are entered in manually in the appropriate positions on the Definitions and Procedures Pages.

The Summary Page is selected, for display, by a panel button. No further manual entry is currently needed.[48]

Idioms Page

Conceptual Role and Purpose: To summarize the complete set of currently active Idiom controls.

Detailed Role:

The Idioms Page is fully systems generated. Off-line, it summarizes all Idiom Loop statements, showing where they conflict and overlap. On-line, it summarizes only those statement which are actually running, thus summarizing the current state of the Idiomatic continuous controls.

Comments Page

Conceptual Role and Purpose: To provide for natural language commentary associated with all program statements and entries.

Detailed Role:

The Comments Page allows explanatory information to be set aside on a statement by statement or entry by entry basis. The Comments Page also follows a Footnoting strategy, marking the point in the program where a relevant comment has been set aside with an appropriate mark (here we will use the dagger † superscripted), for example as in the figure:

[48] However, later definition may call for various modifying filtering actions to be configured on the Summary Page.

from a Variable consists of only that (Real or State) data involved in the current computation. The system can file a copy of the current Summary Page under manual or program control, and under a user or system generated name. The necessary commands for doing this and tieing the file name to any batch Recipe or Lot names are ???????.

Reference Conventions:

The purpose of the Summary Page is to develop a record of the operation, for operator or archival use. It is not a substitute for the normal variable storage or operating display. Thus the Summary Page's reference structure is principally used by external software systems. Nevertheless, the data can be individually accessed by a dot reference expression consisting of any reference to the Operation, followed by a dot, followed by the Keyword **SUMMARY**, followed by a dot, followed by the reference displayed after the exclamation mark in the originally marked Page.

Typical Formats:

The above Figure illustrates the Footnote like usages. In this case there is no Footnote, and no Summary Page user programming, to go with the Footnote. However, the screen Listing should show the summary data and the Archival Format should redundantly summarize that display. The above listing in Archival form becomes:

```
<<<PROCEDURES>>> STYRENE_PLANT. REACTOR
CONTROL_REACTOR
<<<CONTINUOUS>>>
RSF<<<SUM>>> = RSR * TF<<<SUM TF.1>>>
RSF /REGULATE/LOLIMIT RSV
R1PIT /REGULATE FST
RX1PT /REGULATE RX1SV
RX2PT /REGULATE RX2SV
"MONITOR R1POT, R2POT, RPOP, RPOA, RPIP. ADJUST RX1PT, RX2PT SETPOINTS AS NEEDED" <<<SUM MESSAGE.5>>>
<<<ENDCT>>>
<<<ENDP>>>
```

The corresponding Summary page might take the following form (with added references and messages):

STYRENE_PLANT.REACTOR Page: Summary

NAME	VALUE	UNITS	
RSF	22.4	CFS	
TF.1	30.5	GPM	
TF.2	30.5	GPM	
TF.3	_	GPM	
MESSAGE.1			"OPEN VENT HAND VALVE"
MESSAGE.2			"MONITOR OUTLET TEMPERATURE"
MESSAGE.3			"PREPARE MIX"
MESSAGE.4			"CLOSE VENT HAND VALVE"
MESSAGE.5	"MONITOR R1POT, R2POT, RPOR, RPOA. ADJUST RX1PT, RX2PT SETPOINTS AS NEEDED."		

Each variable is listed with its name and occurrence number (if there are several), its value, and its units. Each expression is listed spelling out the expression with an occurrence number (if there are several) and its value. Each Call is listed with the **CALL** keyword and occurrence number (if there are several), its Call or Task name, or a Blank if the Call has not been encountered. Each Message is listed with the **MESSAGE** keyword and occurrence number (if there are several), and the Message or a Blank (if the Message has not been encountered).

The corresponding Archival Format might appear as:

```
<<<SUMMARY>>> STYRENE_PLANT. SEPARATOR. REACTOR
<<<ENTRYS>>> RSF 22.4 CFS
<<<ENTRYS>>> TF.1 30.5 GPM
<<<ENTRYS>>> TF.2 30.5 GPM
<<<ENTRYS>>> TF.3 _ GPM
<<<ENTRYS>>> MESSAGE.1 "OPEN VENT HAND VALVE"
<<<ENTRYS>>> MESSAGE.2 "MONITOR OUTLET TEMPERATURE"
<<<ENTRYS>>> MESSAGE.3 "PREPARE MIX"
<<<ENTRYS>>> MESSAGE.4 "CLOSE VENT HAND VALVE"
<<<ENTRYS>>> MESSAGE.5 "MONITOR R1POT, R2POT, RPOP, RPOA, RPIP. ADJUST RX1PT, RX2PT SETPOINTS AS NEEDED."
<<<ENDS>>>
```

As shown, the Summary Page contents is largely developed automatically from the control program exclamation marks. It will probably be desirable to include additional record specification within the Summary Page for certain classes of data.

Keywords:

corresponding to named Discrete variables in argument Lists will be represented like all other parameter Entries.

This **<<<HEADINGM>>>** entry will initially duplicate the original Procedures Page call form. This State (including Parameters) will also be restored whenever the Call is first made after being inactive. However, in a running program the Parameters and States may be changed by programmed, engineer, or operator action. ???General question of which values are permanent parts of the data base and which change by operator action. Also question of commands to reset values to their data base values, or to reset the data base values to new values.???

Following the Block heading entry, the Listing may include **<<<ENTRYM>>>** entries for every parameter (argument in the defining argument List). The entry will contain first the (argument List) parameter name, then its current value, and then, optionally, its units. The set of entries may contain any parameter defined in the argument List, at most once. On output, the complete list of entries will be generated, with units.

Following the final parameter entry, the Page Listing will be terminated by the **<<<ENDM>>>** keyword.

Archival/Listing Format Relationships:

The **<<<PARAMETERS>>>** Archival Format keyword identifies the Parameters Page for the Listing Format.

Its Operation reference identifies the required Listing Format Operation.

The **<<<HEADINGM>>>** keyword identifies the Block Parameters Page heading and its Listing contents.

Similarly, the **<<<ENTRYM>>>** keyword identifies the parameter entries and their Listing contents.

Parameters Pages are to be ordered, page numbered, and listed (Archivally) lexicographically by Block (and Instance) name.

Archival/Entry Format Relationships:

The Parameters Page is selected, for display, by a panel button. The outline structure will have been already generated automatically.

Remaining entries are entered as text spaced and carriage returned, or inserted in by normal edit actions. The Listing Format resulting is then pretty-printed before re-display.

Summary Page

Conceptual Role and Purpose: To record and display, for general access the overall State of the Operation Unit or Plant, including the history of the immediate production run. To generate Recipe records.

Detailed Role:

This Page summarizes a current production operation, for a selected process Operation or path. The summarized data may be specified in the Operation, or in a Recipe. In the latter case, the data includes the Recipe setup (formula) data. More generally, any SuperVariable reference or message may have its current value recorded on the Summary Page. The usage is similar to a Footnote, except that the marking asterisk is replaced by a superscripted exclamation mark. For example, a Procedures Page statement can be altered to cause a current value to be recorded on the Summary Page by following that value by an exclamation mark (which may be numbered, by the system or the user, if multiple recordings of the same variable are involved):

STYRENE_PLANT. REACTOR Page: Procedures

CONTROL_REACTOR

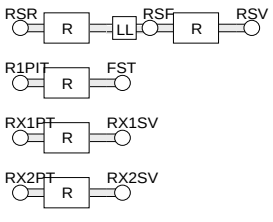
RSR REGULATE LOLIMIT RSF REGULATE RSV

R1PIT REGULATE FST

RX1PT REGULATE RX1SV

RX2PT REGULATE RX2SV

"MONITOR R1POT, R2POT, R3POT; RPOP, RPOA, RPIP. ADJUST RX1PT, RX2PT SETPOINTS AS NEEDED"



A Variable, parenthesized expression, unnamed Call, named (Block or Recipe) Call, or Message can be marked for recording. If a given Variable or Call (or an expression or message) occurs only once, only the exclamation mark will be needed to mark it. If the marked object has a name (Variable, Task, or Block name), and it is marked more than once, the exclamation mark will be followed by that name, a dot, and a reference number. Multiple expressions, Calls, or Messages are marked by the exclamation mark followed by the word **EXPRESSION**, **CALL**, or **MESSAGE**, followed by a dot followed by a reference number. User entered codes or numbers must conform to the above convention except that the reference number may be replaced by any unique User Name or number.

If the same statement is encountered repeatedly, only the last value is recorded. If that variable (or expression) is to be trended instead, a double exclamation mark is used (as in **RSF$^{!!}$ = RSR*TF$^{!!TF.1}$**). An expression, Call, or Message trended is represented by a count of the number of occurrences, and a time stamp value for each. The data recorded

<<<SEQUENTIAL>>>
MEAS = STP
******
XFB = OUT
STV = OUT
<<<ENDSQ>>>

The Parameters Page for the earlier Details Page Compiled Idiom takes the following Listing Format:

STYRENE_PLANT. SEPARATOR Page: Parameters

PID: STP	+	AUTO
SET	0.0	%
MEAS	0.0	%
OUT	0.0	%
XFB	0.0	%
PB	20.0	%
INT	30.0	MIN
DER	0.0	MIN
INTLG	0.0	%
INTRM	0.0	%
DERLG	0.0	%
DERRM	0.0	%

The corresponding output Archival Format is (The final units name in each **<<<ENTRYM>>>** row is optional/redundant on input.):

<<<PARAMETERS>>> STYRENE_PLANT. SEPARATOR. DETAILS_PAGE. <<<PAGE 42>>>
<<<HEADINGM>>> PID: STP + AUTO
<<<ENTRYM>>> MEAS 0.0 %
<<<ENTRYM>>> OUT 0.0 %
<<<ENTRYM>>> XFB 0.0 %
<<<ENTRYM>>> PB 20.0 %
<<<ENTRYM>>> INT 30.0 MIN
<<<ENTRYM>>> DER 0.0 MIN
<<<ENTRYM>>> INTLG 0.0 %
<<<ENTRYM>>> INTRM 0.0 %
<<<ENTRYM>>> DERLG 0.0 %
<<<ENTRYM>>> DERRM 0.0 %
<<<ENDM>>>.

Of course, the **<<<ENTRYM>>>** rows must be consistent with the original Task (or algorithm) definition argument List, and with the associated variable definitions.

Keywords:

Entered in this order: **<<<PARAM @>>>**, **<<<PARAMETERS>>>**, **<<<PAGE @>>>**, **<<<HEADINGM>>>**, **<<<ENTRYM>>>**, **<<<ENDM>>>**.

Discussion:

The **<<<PARAM @>>>** Keyword is added after the appropriate Blocks Call to indicate the reference to a corresponding numbered Parameters Page, with @ replaced by the actual page number generated by the system.

Any Parameters Page entry will be ignored unless it corresponds to a Block Call in the same archival file (, or is preceded by a **<<<MAKE>>>** Keyword as explained later under the **MAKE_RECIPE** command). The Parameters in that entry must correspond in Parameter name and engineering units to the parameters as ordered in the Task definition argument List. But once accepted as having such a Call or Keyword the entry is accepted at face value; its individual argument values are entered as listed.

A Parameters Page Archival Listing starts with the **<<<PARAMETERS>>>** keyword, followed by a Operation and Page reference (defining the Operation containing the Parameters Page), including the final **.<<<PAGE @>>>** Keyword expression.

Following this initial entry, the Listing includes the **<<<HEADINGM>>>** keyword followed by a Task (or algorithm) name followed by a colon followed by the corresponding Block name.

The Block name may include an appended Block Instance name separated from the initial name component by a dot operator. This allows several Blocks to share a common variable name: **STP. STARTUP**, **STP. RUN**, **STP. SHUTDOWN**. There may be more than one such Block name.

Following the Block Call (or Instance) name, the entry will include a list of State names, drawn from the original Task (or algorithm) definition States Expression argument Lists, defining the current State of the Block Call. This will include an entry for every State value defined in the Task definition State definition expression argument Lists.

Note that this is the distinct way of including this kind of Discrete valued data in the Parameters Page. These States are viewed as being distinctly associated with the Task (They are dot referenced through the Task name.) rather than with argument variables. State data

A Footnote entry starts with the keyword **<<<FOOTNOTE #>>>**, followed by a statement or Activity representing the computation to be carried out whenever the Footnote reference is encountered; the # symbol is replaced by the consecutive number corresponding to the reference and Footnote.

A Compiled Idiom entry starts with the keyword **<<<COMPILED #>>>**, followed by a statement or Activity representing the computation to be carried out whenever the Idiom reference is encountered; the # symbol is replaced by the consecutive number corresponding to the reference and Footnote.

An invoked Local State Environment Declaration Statement Truth Table entry starts with the keyword **<<<INVOKED #>>>**, followed by a statement or Activity representing the computation to be carried out whenever the Idiom reference is encountered; the # symbol is always replaced by one, since each Declaration can only invoke one Truth Table.

Following all Details Page entries, the Page is terminated with a **<<<ENDL>>>** keyword.

Archival/Listing Format Relationships:

The **<<<DETAILS>>>** Archival Format keyword identifies the Details Page for the Listing Format.

Its Details Page reference identifies the required Listing Format Operation, and the referencing Page and Statement or Entry.

The **<<<FOOTNOTE #>>>**, **<<<COMPILED #>>>**, or **<<<INVOKED #>>>** keyword identifies the form of Details Page entry, and, in the case of Footnotes, defines the appropriate asterisk entry and Footnote number. The corresponding Listing Format relationships are selected through Work Station Mouse and panel button function.

Archival/Entry Format Relationships:

The Details Page is selected, for display, by a panel button. Most entries will have been already generated in rough form automatically.

Remaining (headings and) entries are entered as text spaced and carriage returned, or inserted in by normal edit actions. The Listing Format resulting is then pretty-printed before re-display.

Parameters Page

Conceptual Role and Purpose: To operationally parameterize and display Recipes, controls, and control tasks.

Detailed Role:

A Parameters Page contains the control parameters for Blocks, whether named and associated with Block Calls or associated with simple re-entrant Calls (described later). Each call, whether direct from the Procedures Page, or compiled from the Details Page, will have a Parameters Page, associated with the Call and referenced by any Block name. Block Calls with different (Block) names will have different Parameters Pages, but a Page may be shared by several Calls if their Blocks have the same name (and instance name).[46] The Parameters Page parameters take their names and data types from the definition argument Lists of the called procedure. For example, the PID algorithm might be defined to have the following Task name and argument List heading:

PID((SET), (MEAS), (OUT), (XFB), (PB), (INT), (DER), (INTLG), (INTRM), (DERLG), (DERRM)) (+/–, MANUAL/AUTO)

The first parenthesized list contains the named parameters.[47] These represent the normal parameters: Setpoint, Measurement, Output, External Feedback, Proportional Band, Integral, and Derivative. In addition, the list includes the integration and derivative filter lag and remainder values. The second parenthesized expression contains a States Expression (as described before) listing the states that may be called (or defaulted) in the Block Call. These represent the Output to Measurement gain sign, and the Auto Manual state. The first value for each State is its default value. The Details Page Compiled Idiom example shows a Block Call for such a PID. The corresponding Parameters Page entry would then define the values for each of the intended parameters. The Parameters Page must match the Task definition argument List in data type and engineering units, but it can include edited values for the parameters.

Alternatively, a Parameters Page entry may be set aside as later described by a **MAKE_RECIPE** command/statement.

Typical Formats:

Recalling the earlier Details Page Block Call (Listing Format):

```
PID: STP + AUTO
    SET = STP.SET
    MEAS = STP.VALUE
    **
    XFB = OUT
    STV = OUT
```

with Archival Format:

<<<BLOCK CALL>>> PID: STP <<<BLOCK>>> + AUTO <<<PARAM 42>>>

[46] Calls allow for several different named Blocks (particularly for Recipe use). When this is the case the corresponding Parameters Page will be built by combining the data of its separately named Blocks, even when those same Blocks are shared by other Calls; the Parameters Pages represent displays of the data, rather than redundant storages of the data.

[47] Each parameter is itself parenthesized, meaning that it can only be passed by name as a value (from the block).

PROCESS. TANK Page: Details

LCH	LCM	LCL	Result
	Lo	Lo	Empty
Lo	Hi	Lo	Low, Failed
Hi	Lo	Hi	Full, Failed
	Lo	Hi	Low, OK
Lo	Hi		Full
Hi	Hi		Filled

When State Prefixes are recognized associated with a Local State Environment Declaration Statement, having States that that Declaration could not return, a Truth Table with the appropriate headings is invoked on the Details Page. The user is expected to fill in the appropriate table entries to define the computations leading to the appropriate Result values.[43] The above Listing Format invoked Truth Table is expressed in the following Archival Format:

<<<DETAILS>>> PROCESS. TANK. PROCEDURE_PAGE. <<<PAGE 47>>>
<<<INVOKED 1>>>
<<<HEADINGTT>>> LCH LCM LCL <<<RESULT>>> Result
<<<ENTRYTT>>> <<<SPACE>>> Lo Lo Empty
<<<ENTRYTT>>> Lo Hi Lo Low, Failed
<<<ENTRYTT>>> Hi Lo Hi Full, Failed
<<<ENTRYTT>>> <<<SPACE>>> Lo Hi Low, OK
<<<ENTRYTT>>> Lo Hi <<<SPACE>>> Full
<<<ENTRYTT>>> Hi Hi <<<SPACE>>> Filled
<<<ENDTT>>>
<<<ENDL>>>.

The initial invoked form of Truth Table has no entries. Its archival representation (as generated automatically before user entry) takes the following form:

<<<DETAILS>>> PROCESS. TANK. PROCEDURE_PAGE. <<<PAGE 47>>>
<<<INVOKED 1>>>
<<<HEADINGTT>>> LCH LCM LCL <<<RESULT>>> Result
<<<ENDTT>>>
<<<ENDL>>>.

The Details Pages would be numbered, Footnotes to Definitions Pages first, then Footnotes to Procedures Pages, then Footnotes to Details Pages, themselves.

Keywords:

<<<DETAILS>>>, **<<<ENDL>>>.** Any reference to a Details Page is coded in the archival listing by the **<<<REF @.#>>>** keyword where the @ symbol represents a unique number code for the referring statement or entry (the Details Page number), and the # symbol represents a consecutively numbered Details Page entry and reference from the referring statement or entry (reference number)[44]. The corresponding Details Page (and referencing statement or entry) is indicated, in Archival Format, by a page code keyword **<<<PAGE @>>>**, where the @ symbol is replaced by the corresponding number code. The individual Details Page entry (and statement or entry reference point) is set off by the appropriate **<<<FOOTNOTE #>>>**, **<<<COMPILED #>>>**, or **<<<INVOKED #>>>** keyword, where the # symbol are replaced by the number matched in the reference. Because the Details Page permits the same programming usages as the Procedures Page, it includes all Activity, Task and Statement keywords.[45] Apart from Activity Bracket keywords, the example includes the archival Truth Table keywords: **<<<HEADINGTT>>>**, **<<<ENTRYTT>>>**, **<<<ENDTT>>>**.

Discussion:

A Details Page Archival Listing starts with the **<<<DETAILS>>>** keyword, followed by a Page reference (i.e. the statement or entry referencing the Details Page).

This reference is a dot reference expression, starting with the proper Operation dot reference expression, followed by a dot operator and the **DEFINITIONS_PAGE**, **PROCEDURES_PAGE**, or **DETAILS_PAGE** name (depending on referencing Page), followed by a dot operator and the **<<<PAGE @>>>** keyword, with appropriate unique number code replacing the @ symbol.

The reference is followed by any number of Details Page entries, each either a Footnote, or a Compiled Idiom, or a Local State Environment Declaration Statement invoked Truth Table.

[43] As indicated later in the discussion of Truth Tables, empty entries in a column indicate Don't Care conditions; the topmost line in the Table, which matched the process state, defines the result.

[44] Allowing for more than one reference (Footnote) on any Details Page, from the same statement or entry.

[45] The Details Page may also have its own Details Page, and so on!

Typical Formats:

The following examples illustrate the three kinds of Details Page (Each draws from usages which may be described in greater detail later.):

Footnote:[39]

STYRENE_PLANT Page: Details

*1 TF = TRF+TFF

This Page is expressed in the following Archival Format:

<<<DETAILS>>> STYRENE_PLANT. DEFINITIONS_PAGE. <<<PAGE 24>>>
<<<FOOTNOTE 1>>>
TF = TRF+TFF
<<<ENDL>>>.

Compiled Idiom:[40]

STYRENE_PLANT. SEPARATOR Page: Details

PID: STP + AUTO
SET = STP.SET
MEAS = STP.VALUE
**
XFB = OUT
STV.VALUE = OUT

This Details Page entry is assumed to represent a default Block Call compilation of the associated Idiom call; the user is permitted to edit it. The above Listing Format compiled result is expressed in the following Archival Format:

<<<DETAILS>>> STYRENE_PLANT. SEPARATOR. PROCEDURE_PAGE. <<<PAGE 36>>>
<<<COMPILED 1>>>
<<<BLOCK CALL>>> PID: STP <<<BLOCK>>> + AUTO <<<PARAM 42>>>[41]
<<<SEQUENTIAL>>>
MEAS = STP
****
XFB = OUT
STV = OUT
<<<ENDSQ>>>
<<<ENDL>>>.

Local State Environment Declaration Statement Truth Table:[42]

---

[39] Footnote from the global Styrene Plant Definition Page:
The TF entry would take the Archival Format:
<<<ENTRY>>> TF _ <<<REF 24.1>>> 0 100 GPM _ _ _ _ 20

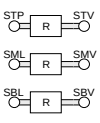
STYRENE_PLANT Page: Definitions

NAME	IN	MIN	MAX	UNITS	VALUE	SET	HI	LO	DEV
FST	1	0	800	DGC	_	_	780	300	20
TRF	2	0	100	GPM	_	_	_	_	20
TFF	3	0	100	GPM	_	_	_	_	20
TF	_*1	0	100	GPM	_	_	_	_	20
POL	4	0	300	FT	_	_	30	10	20
WOL	5	0	300	FT	_	_	30	10	20

[40] Idiom Compiled from the top statement of the Styrene Plants Procedure Page Control Separator Task (**$STP_{REGULATE}$ STV**):
The Archival Format for this statement is:
STP/REGULATE <<<REF 36.1>>> STV

STYRENE_PLANT. SEPARATOR Page: Procedures

CONTROL_SEPARATOR
$STP_{REGULATE}$ STV
$SML_{REGULATE}$ SMV
$SBL_{REGULATE}$ SBV



[41] The **<<<PARAM 42>>>** Keyword will be discussed later under the Parameters Page; the **<<<BLOCK>>>** and **<<<BLOCK>>>** Keywords under the Block Call.

[42] This example is derived from a Styrene Plant Appendix discussion. The Truth Table is invoked from an example Task:
The Archival Format for this statement is:
[LCH, LCM, LCL] <<<REF 47.1>>>;

PROCESS. TANK Page: Procedures

TEST_TANK
[LCH, LCM, LCL];
EMPTY: TKV, OPEN;
◊ FULL: END;◊
FILLED: TKV, CLOSE

ABORT:
<<<SEQUENTIAL>>>
(TRPp, TFPp), STOP
(TRV, TFV), CLOSE
END <<<END 2>>>
<<<ENDSQ>>> <<<ENDCT>>>
CONTINUED
<<<ENDSQ>>>
CONTROL_FEED
<<<CONTINUOUS>>>
TF/REGULATE/SPLR TRV TFV
TRL/REGULATE/OVERRIDE TRV
<<<ENDCT>>>
<<<ENDP>>>

Keywords:

<<<PROCEDURES>>>, **<<<ENDP>>>**; Also any Activity, Task, and Statement keywords. Apart from Activity Bracket keywords, the example includes the **END** statement keyword **<<<END 2>>>** to be explained later.

Initial Activity Bracket Keywords: **<<<SEQUENTIAL>>>**, **<<<PARALLEL>>>**, **<<<LOOPING>>>**, **<<<CONTINUOUS>>>**, **<<<STATE_DRIVEN>>>.**

Terminal Activity Bracket Keywords: **<<<ENDSQ>>>**, **<<<ENDPL>>>**, **<<<ENDLP>>>** , **<<<ENDCT>>>**, **<<<ENDSD>>>**.

Discussion:

The Procedures Page contains one unnamed Activity followed by any number of (uniquely) named Tasks.
The Activities are set off by Activity Brackets (listed above, and described later).
The Procedures Page is initiated, in Archival Format, with the **<<<PROCEDURES>>>** keyword; followed by an Operation reference, identifying the containing Operation, followed by the unnamed Activity (there may be more than one, all executed independently on call to the Operation), set off an initial Activity Bracket keyword and terminated by the corresponding Terminal Activity Bracket keyword; followed by the named Tasks, each set off by a name, optional argument Lists (as described later), and Activity (with its paired initial and terminal Bracket keywords); and terminated by a final **<<<ENDP>>>** keyword.
The named Tasks may also take the form of *Single Statement Tasks* (without the Activity Brackets): set off by a name, optional argument Lists, and single statement definition.
Null Statement Tasks can be defined with name and argument List, but without any following definition. Their use is to allow Block Calls which set aside operational Block data without doing any thing with it. This data can then be acted on by other programmed statements.
It is intended that the Tasks be listed in lexicographical order on the Procedures Page listing, under pretty-printing.

Archival/Listing Format Relationships:

The **<<<PROCEDURES>>>** Archival Format keyword identifies the Procedures Page for the Listing Format.
Its Operation reference identifies the required Listing Format Operation.
Bracket keywords identify the necessary Activity Brackets.
Because the Listing Format is the result of pretty-printing, the Parameters Page must accommodate when there is insufficient space for an entry on a line. Essentially the second and following lines of a statement so continued will be offset. However certain statements may need their own rules for effective pretty-printing of multi-line continuations.

Archival/Entry Format Relationships:

The Procedures Page is selected by a panel button.
Remaining (headings and) entries are entered as text spaced and carriage returned. The Listing Format resulting is then pretty-printed before display. Normal edit actions can be carried out to alter the resulting text.
Special Entry Format conventions for Activity Brackets and **END** commands will be developed later.

Details Page

Conceptual Role and Purpose: To permit suppressed detailed computations associated with other pages, and separately represent them.

Detailed Role:

The Details Page represents manually entered or automatically compiled details for an associated statement (each such statement or Definitions Page variable definition has a single associated Details Page). The entries on the Page may include: Footnotes, entered by the user; compiled Idioms, compiled by the system but editable by the user; and Local State Environment Declaration Statement Truth Tables, to be filled in by the user. In greater detail: Footnote entries consist of single statements, and compiled Idioms take the form of Block calls.

Attribute in the preceding heading, each of appropriate data type.

Archival/Listing Format Relationships:

The **<<<DEFINITIONS>>>** Archival Format keyword identifies the Definitions Page for the Listing Format.

Its Operation reference identifies the required Listing Format Operation.

The **<<<HEADING>>>** keyword identifies the SuperVariable Format line with the intended list of consecutive Attribute headings.

The **<<<ENTRY>>>** keyword identifies the individual SuperVariable entry with the intended list of consecutive Attribute values.

Although the above listings do not show this, it is intended that all Definition Groups (groups of SuperVariable entries sharing the same Format and Headings) be lexicographically (alphabetically) ordered in their group, as part of the standard pretty-printing.

It is intended that all distinct Definitions Groups sharing a Format (set of headings) in common be merged under a common set of headings, as part of the pretty-printing.

It is intended that the set of all Definitions Groups be itself lexicographically ordered.

Because the Listing Format is the result of pretty-printing, the Definitions Page has special requirements if there is insufficient space for an entry on a line. In this case, the entire Definition Group must be split, so that the first few columns are printed together in a set with their headings, and then a second set of columns are printed spaced vertically from the first and offset from the left hand margin to distinguish the sets, and then a third, etc.

Archival/Entry Format Relationships:

The Definitions Page is selected by a panel button.

Lines are entered with their column entries separated by spaces or tabs.

The first line in the Definitions Page is interpreted as a heading (SuperVariable Format; parsed for correct form). Following lines are interpreted as SuperVariable entry lines.

Lines are terminated by carriage returns.

A Definition Groups is terminated by a repeated carriage return, after which the next line will be interpreted as a new SuperVariable Format heading line.

The Listing Format resulting is then pretty-printed before display. Normal edit actions can be carried out to alter the resulting text; single carriage returns leading to a new entry line, double carriage returns leading to a new Group.

Procedures Page

Conceptual Role and Purpose: (Together with Simple Procedures Page) To represent controls and procedures associated with an Operation.

Detailed Role:

The Procedures Page lists the Operation's unnamed main Activity and all of its named Tasks. The Page definition depends on the later definition of component Activities, Tasks, and Statements.

Typical Formats:

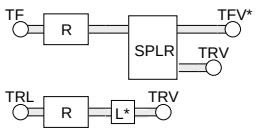
The Styrene example shows the following Feed Tankage Operation Procedures Page, in Listing Format:

```
STYRENE_PLANT. FEED_TANKAGE                    Page: Procedures

(TRPp, TFPp), START
  CONTROL_FEED
  ABORT:
    (TRPp, TFPp), STOP
    (TRV, TFV), CLOSE
◊   END◊
CONTINUED

CONTROL_FEED
  TF  REGULATE SPLR  TRV   TFV

  TRL REGULATE OVERRIDE  TRV
```



The Archival input or output Format of this same page is:

<<<PROCEDURES>>> STYRENE_PLANT. FEED_TANKAGE
<<<SEQUENTIAL>>>
(TRPp, TFPp), START
<<<CONTINUOUS>>>
CONTROL_FEED

The Definitions Page constitutes a listing of all of the Operation's Variables, divided into Definition Groups each sharing the same headings and Attributes, each Group listed in its own table. It also includes all explicitly declared Lists. The later SuperVariable and List sections will define the details of their Definitions Page declaration.

Typical Formats:

The Styrene example shows the following Furnace Operation Definitions Page, in Listing Format:

STYRENE_PLANT. FURNACE Page: Definitions

NAME	STATE	STATES	MANUAL_STATE
FSHV	_	OPEN/CLOSED	_
FSVHV	_	OPEN/CLOSED	_
FNHV	_	OPEN/CLOSED	_

NAME	DOUT	STATE	STATES
FFIg	1	_	START/STOP

NAME	DIN	STATE	STATES
FFFI	1	_	LIT/COLD

NAME	OUT	VALUE	MIN	MAX	UNITS	STATES
FSV	1	_	0	100	%	CLOSED=0 / OPEN=80
FNV	2	_	0	100	%	CLOSED=0 / OPEN=100
FFV	3	_	0	100	%	CLOSED=0 / OPEN=50

NAME	IN	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
FSF	6	_	0	100	CFS	_	_	_	_
FFF	7	_	0	100	CFS	_	_	_	_
FNF	8	_	0	100	CFS	_	_	_	_
FSP	9	_	0	500	PSIG	_	_	_	_

As expressed in output Archival Format, this would become:

<<<DEFINITIONS>>> STYRENE_PLANT. FURNACE
<<<HEADING>>> NAME STATE STATES MANUAL_STATE
<<<ENTRY>>> FSHV _ OPEN/CLOSED _
<<<ENTRY>>> FSVHV _ OPEN/CLOSED _
<<<ENTRY>>> FNHV _ OPEN/CLOSED _
<<<HEADING>>> NAME DOUT STATE STATES
<<<ENTRY>>> FFIg 1 _ START/STOP
<<<HEADING>>> NAME DIN STATE STATES
<<<ENTRY>>> FFFI 1 _ LIT/COLD
<<<HEADING>>> NAME OUT VALUE MIN MAX UNITS STATES
<<<ENTRY>>> FSV 1 _ 0 100 % CLOSED=0 / OPEN=80
<<<ENTRY>>> FNV 2 _ 0 100 % CLOSED=0 / OPEN=100
<<<ENTRY>>> FFV 3 _ 0 100 % CLOSED=0 / OPEN=50
<<<HEADING>>> NAME IN VALUE MIN MAX UNITS SET HI LO DEV
<<<ENTRY>>> FSF 6 _ 0 100 CFS _ _ _ _
<<<ENTRY>>> FFF 7 _ 0 100 CFS _ _ _ _
<<<ENTRY>>> FNF 8 _ 0 100 CFS _ _ _ _
<<<ENTRY>>> FSP 9 _ 0 500 PSIG _ _ _ _
<<<ENDD>>>.

The Archival entry Format would be the same except for any added white space or comments.

Keywords:

Archival: **<<<DEFINITIONS>>>**, **<<<HEADING>>>**, **<<<ENTRY>>>**, **<<<ENDD>>>**.

ICL Listing: These include all of the SuperVariable Attribute names: **NAME**, **IN**, **DIN**, **OUT**, **DOUT**, **VALUE**, **STATE**, **TIME**, **COUNTS**, **MANUAL_VALUE**, **MANUAL_STATE**, **MIN**, **MAX**, **UNITS**, **STATES**, **SET**, **HI**, **LO**, **DEV, ACCUM**.

Discussion:

The Definitions Page would be initiated, in Archival Format, with the **<<<DEFINITIONS>>>** keyword and terminated with the **<<<ENDD>>>** keyword.

The **<<<DEFINITIONS>>>** keyword would be immediately followed by a legitimate Operation reference to the Operation for which the Page is to be defined. It would then be followed directly by one or more headings each followed directly by one or more entries. The final entry is followed by the **<<<ENDD>>>** keyword.

A heading consists of an initial **<<<HEADING>>>** keyword and by any number of SuperVariable Attribute name heading keywords.

An entry consists of an initial **<<<ENTRY>>>** keyword, followed by column entry values, one per each SuperVariable

SuperVariables may have Attributes and States added. But no Attribute (except a Name Attribute) may be added directly before an Attribute of identical Attribute Type Name in the SuperVariable Format.[38] And all State names must be unique.

SuperVariable Attribute values may be freely changed, except for Name Attributes which can only changed using the above renaming declaration convention.

Operation Calls

Operations differ from Tasks, defined later, in that they have data associated with them. Unlike the Task, an Operation brings all of its own data to every call made on it; it is inherently not re-entrant. Thus only one Operator call may be sensibly active at a time. Nevertheless, the Operations can be called, as one means of activating them. And we use the Operation Call with arguments as a way of defining Operations intended to represent temporary combination Process Objects, such as batch lines (trains, paths). The assumption is that a given plant will be set up with certain sets of duplicated units from which temporary production trains can be grouped for the duration of the production of a particular batch. This could be accommodated by booking the needed process unit Operations in a train. However, booking does not create a recognizable composite entity, supported by all of the ICL reference conventions, whereas an argument list Operator call does. It is assumed that the maximum number of trains supported for each style of production will be naturally obvious for each multi-unit process.

An Operator, intended to be used with an Operator call with arguments, lists its dummy arguments, in parentheses, separated by commas, on the line after the title line. As with Tasks, later, multiple argument Lists may be used (to permit development of independent groupings). The actual call arguments must be modeled after the definition arguments.

TRAIN_1(STYRENE_PLANT) Page: Operation
(REACTOR, HEAT_RECOVERY, SEPARATOR)

SYSTEM STATES: CONFIGURE/SETUP/SIMULATE/OPERATE,
RUN/SUSPEND/CONTINUE/ABORT/ END,
ACTIVE/ INACTIVE, BOOKED/ UNBOOKED, _/INITIALIZE

USER STATES:

TASKS: EMERGENCY_STEAM_SHUTDOWN, FINAL_SHUTDOWN,
PREPARE_EMERGENCY_SHUTDOWN, PREPARE_STARTUP,
PURGE, REACT, SHUTDOWN_FEED, SHUTDOWN_TEMPER-
ATURE, SHUTDOWN_STEAM, STEAM

SUBOPERATIONS: FURNACE, REACTOR, HEAT_RECOVERY,
SEPARATOR, FEED_TANKAGE

The example defines a Train based on the Styrene Plant, in which the Furnace and Feed Tankage are shared, but the Reactor, Heat Recovery, and Separator Units are selected by argument. The archival form is similar, placing the argument list separately on its own line without distinguishing keyword:

<<<OPERATION>>> TRAIN_1(STYRENE_PLANT)
(REACTOR, HEAT_RECOVERY, SEPARATOR)
<<<SYSTEM STATES>>> …

The structure can be extended to allow the pre-allocation of several identical Operations (to represent multiple Operation Trains and Calls) which can then be "re-entrantly" Called, up till the point which all of the Operations have been used up. The extension is based on the later described concept of Lists. In this case the Operation name (and any parenthesized model Operation name) is followed (in a List Declaration) by a colon (see Naming Prefixes) and parenthesized List (or a bracketed number indicating the number of identical Operations in the Operation List, as below):

BATCH_PLANT. BATCH_TRAIN: [2*] Page: Operation
(ReactorA*(Reactor): (Reactor1/Reactor2), Load1, Load2, Water_Load, StoreA(Store): (Store1/Store2))

The example includes two identical **BATCH_PLANT_TRAIN** Operations in its List. The asterisk after the number 2 is keyed to the asterisk after the **ReactorA** argument, indicating that the Reactor argument choice is the limiting hardware resource, defining the maximum number of Trains that can run at a time. This information can be used to limit re-entrant Calls to the Train, so that the application is not left with available reactors without correspondingly available Train Operations. The individual (Listed) declarations can be referenced by any of the List conventions later described, of by any explicit element names. The Operation Call obeys all related conventions on State Assignments and argument Lists described later.

Definitions Page

Conceptual Role and Purpose: To declare and define all Operation Variables and process I/O.

Detailed Role:

[38] This restriction generally ensures that SuperVariable (Zipper) references will be unaffected by inheritance, **if normal usages are followed**.

or operators.[35]

The **<<<TASKS>>>** keyword is followed by the list of Task names, complete as the time of entry, separated by commas. Task names must be User Names.

The **<<<SUBOPERATIONS>>>** keyword is followed by the list of subOperations names, complete as of the time of entry, separated by commas.

Archival/Listing Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword identifies the Operations Page for the Listing Format.

Its Operation reference identifies the required Listing Format Operation.

The other Archival keywords identify the appropriate Listing Format rows and underlined row headings.

Archival/Entry Format Relationships:

The **<<<OPERATION>>>** Archival Format keyword corresponds to a engineer's workstation panel new Operation button selection.

The engineer's work station panel Current Operation entry window (see appropriate section) allows entry of the new Operation's name.

The other entry positions are selected by arrow keys or carriage return, and entered as text terminated by carriage return.

Modeled Operations:

An Operation can be modeled after an existing Operation. When this is required, the normal Operation reference is immediately followed by a second Operation reference (to the already defined model Operation), in parenthesis, as in: **MODEL_UNIT(PROCESS_UNIT)**, and:

Styrene_Plant_Var_1(STYRENE_PLANT) Page: Operation

SYSTEM STATES: CONFIGURE/SETUP/SIMULATE/OPERATE,
RUN/SUSPEND/CONTINUE/ABORT/ END,
ACTIVE/ INACTIVE, BOOKED/ UNBOOKED, _/INITIALIZE

USER STATES:

TASKS: EMERGENCY_STEAM_SHUTDOWN, FINAL_SHUTDOWN,
PREPARE_EMERGENCY_SHUTDOWN, PREPARE_STARTUP,
PURGE, REACT, SHUTDOWN_FEED, SHUTDOWN_TEMPER-
ATURE, SHUTDOWN_STEAM, STEAM

SUBOPERATIONS: FURNACE, REACTOR, HEAT_RECOVERY,
Separator_Var_1(STYRENE_PLANT. SEPARATOR),
FEED_TANKAGE

When such a modeled Operation is created, it will default to a copy of the model Process, whose subOperations are distinct from the corresponding model subOperations, but copies modeled after them. Any such subOperation can be altered as long as the result is still an Operation modeled after the corresponding original subOperation. Note that the subOperation reference must be formed, consistent with the already defined Dot Reference scoping rules of the language, to reference the actual model subOperation, as referenced from within the new Operation.[36]

Model Element Deletion/Redefinition Restrictions:

The purpose of model modification restrictions is to ensure that all correct references and Task calls, defined for the model Operation, continue to work unchanged in effect when applied to the modeled operation, unless explicitly overridden.

Any named element (variable, State, Task, subOperation) in the model Operation must be preserved in (cannot be deleted from) the modeled Operation.

Any new named element (variable, State, Task, subOperation) may be added to the modeled Operation.

Any named element in a modeled Operation can be renamed from its counterpart in the model Operation by following its declared name in its normal declaration by the name being replaced in parentheses. For example, in the above figure: **Separator_Var_1(STYRENE_PLANT.SEPARATOR)**, representing a variant Separator unit.[37]

Model element names, corresponding to renamed elements in the modeled Operations, act as synonyms for the renamed elements within the scope of the renaming, being compiled into the new name in all modeled Operation declarations and being interpretively renamed in other applications.

Any Task can be overwritten by a new Task having the same (or renamed) name, as discussed later.

---

[35] This is allowed to permit the representation of certain system choices.

[36] The notation **NAME1(NAME2)** will occur in a number of different contexts, throughout the language, as a way of expressing that **NAME1** is a user defined named entity modeled after the similar named entity already defined. Depending on context, the modeling may involve the identical function with a new name, a copied but identical function with a new name, or a similar function with restricted modification and a new name.

[37] Since all of the SubOperations are modeled, the existing notations for the other SubOperations are effectively shorthands for: **FURNACE(STYRENE_PLANT.FURNACE), REACTOR(STYRENE_PLANT.REACTOR), HEAT_RECOVERY(STYRENE_PLANT.HEAT_RECOVERY), FEED_TANKAGE(STYRENE_PLANT.FEED_TANKAGE)** respectively.

At any point in the program in which an Operation can be referred to by name, the Operation (illustrated in the following by the Operation named **STYRENE_PLANT**) supports the following dot reference capabilities:

- Any subOperation can be referred to by adding the dot and the subOperation name: **STYRENE_PLANT. FURNACE**.
- Any contained Task can be referred to by adding the dot and the subOperation name: **STYRENE_PLANT. PREPARE_STARTUP**.
- Any Variable can be referred to by adding the dot and the variable name (with any further SuperVariable reference notation): **STYRENE_PLANT. TF**.
- Any Block (named) call can be referred to by adding the dot and the Block name (with or without any further parameter reference): **STYRENE_PLANT. F100**, **STYRENE_PLANT. F100. MEAS**. The Block call supports further conventions to distinguish ambiguous references.[32] Certain other (Theme and Idiom Loop) statement forms, having associated data structures may also be referenced in terms of an associated variable or user name and statement keyword.
- In some cases, an Operation's Pages may have internal data, referenceable through the Page name and value name: **STYRENE_PLANT. SUMMARY. TF.** 1.
- As a sometimes more efficient reference, the language supports a Scoping Prefix, which permits a following statement or Activity to be written is if it had been defined in the scope of a named Operation. The Scoping Prefix takes the form of an initial **IN** keyword, followed by a dot reference to an Operation, followed by a colon. The Prefix precedes the statement or Activity whose scope it sets. Thus, in the Styrene Plant example, from within the Styrene Plant, the Discrete assignment statement:

  IN REACTOR: (RSV, RX1SV, RX2SV), CLOSE

  has the same meaning as:

  (REACTOR.RSV, REACTOR.RX1SV, REACTOR.RX2SV), CLOSE.

  Outside that Styrene Plant, the same statement might be expressed as:

  IN STYRENE_PLANT.REACTOR: (RSV, RX1SV, RX2SV), CLOSE.

  meaning:

  (STYRENE_PLANT.REACTOR.RSV, STYRENE_PLANT.REACTOR.RX1SV, STYRENE_PLANT.REACTOR.RX2SV), CLOSE.

Typical Formats:

The earlier Relation between Program Representations section showed typical Operations Page formats.

Keywords:

Archival: Entered in this order: **<<<OPERATION>>>**, **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>** and, **<<<ENDO>>>**.

Mandatory: **<<<OPERATION>>>**, **<<<ENDO>>>**.

Optional: **<<<SYSTEM STATES>>>**, **<<<USER STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**.

Redundant: **<<<SYSTEM STATES>>>**, **<<<TASKS>>>**, **<<<SUBOPERATIONS>>>**, but if they are made here they must be complete and consistent with the built in system character, or with the later declarations.

Discussion:

As indicated earlier, within an Operation, all subOperation, Task, and State Names must be mutually unique.[33]

The **<<<OPERATION>>>** keyword must be immediately followed by a legitimate Operation reference[34]. This reference includes, as its final subOperation name, the name of the new Operation, preceded by the higher level Operation reference to the intended Operation including the new Operation as a subOperation. Operation names must be User Names, as defined above.

However there is an implied **GLOBAL** operation, containing definitions which are globally available to define these pages (and look at the Tasks and Operations listed Globally) the corresponding Operation reference is replaced by the Keyword **<<<GLOBAL>>>** as in: **<<<OPERATION>>> <<<GLOBAL>>>**.

The **<<<SYSTEM STATES>>>** keyword must be immediately followed by the exact list of system State names as shown in the typical Operations Page formats although they can be entered in any order, in a States Expression, separated by the appropriate operators as described next.

The **<<<USER STATES>>>** keyword is followed by a States Expression, including all of the user named and defined States. States and State Expressions will be defined in greater detail in the Section on the State named data type. For now a States Expression is a list of the intended State names, where States which must be mutually exclusive are listed separated by a slash (*/*) operator, and where sets of these mutually exclusive States must be separated by a comma (,) operator. ICL allows States (and only States) to assume the names of any valid word including keywords

---

[32] In rare cases where the shared SuperVariable and Block Attribute/Parameter names define an ambiguity, the SuperVariable takes precedence and a **BLOCK** keyword can be used to disambiguate the reference: **STYRENE_PLANT. F100. SET** vs. **STYRENE_PLANT. BLOCK. F100. SET**. The **PARAMETERS** (Page) locates the Block Parameter; its name can similarly be used to disambiguate the reference: **STYRENE_PLANT. PARAMETER. F100. SET**. Similar ambiguities may affect references to Loops and other structures. Their references can be disambiguated by using the names of the Pages in which the data is declared: **DEFINITIONS**/**PROCEDURES**/**PARAMETERS**/ **IDIOMS**. Alternatively, to avoid confusion, the more specific object type distinguishing keyword can be used as above: **BLOCK**/**LOOP**/**IDIOM**.

[33] The lexical rules allow User Names, identical to Keywords, but this is not a generally a good idea; and names of all elements within a given Operation should generally be mutually unique, as a matter of good practice.

[34] User Name or dot reference expression, defining the intended reference, containing only the names of nested Operations as defined above.

punctuators) and integer Numbers as well.

Corresponding to the Symbol defining rules are some pretty-printing rules for reconstructing names and numbers. These are consistent with the above rules; pretty-printed listings can always be consistently re-entered.

- Between System Names (including Keywords) and their directly following Numbers or User Names there will always be printed one space.
- After every dot operator there will always be printed one space.
- After every User Name, ambiguous with a System Name (or Keyword), there will always be printed one underscore, before any additional space or symbol.
- Between a Number and a following dot operator there will always be printed one space.
- Between two Numbers or User Names there will always be printed two spaces (including any (above) pretty printed underscore).
- In ICL, on input or as pretty-printed, a real number in floating point will always be expressed with algebraic operators, as in: **1.23*10^5** or **1.23/10^5**.

Dot Reference Notation

Although not part of the lexical processing, it is appropriate at this point to describe the dot reference notation. The ICL program (i.e. Operation) is defined as one large data structure, in which any User Defined and named object, may be directly referenced and accessed. The reference is based on the Operations hierarchy but extends that into lower level objects within the lowest Operation referenced: Tasks, Blocks, and Recipes, within Operations; Parameters within Blocks; SuperVariable Attributes within SuperVariables within Operations. The details of this reference will be developed within each affected definition, and within the discussion of the Zipper reference for SuperVariables. At the Operation level, an Operation is always referenced starting with the name of a subOperation in the current environment[30], followed by a dot operator followed by one of that subOperation's subOperations, and so on until the desired referenced Operation name is reached (e.g. **STYRENE_PLANT. FURNACE**).[31]

One of the purposes of the language is to provide externally accessible data structures for operating and supervisory purposes, based on its structure; to make all the application data directly accessible in every natural data grouping. The system must provide system calls to allow external software access to every dot referenced component, in a standard Data Format.

When dot references are pretty-printed, they go through their own pretty-printing process; a dot reference, once compiled, is a reference to the indicated element however the name of that element may change. Therefore, all program or files under the control of the ICl system are effectively updated to any name changes of ICL elements. It is thus desirable to place external references under the control of the system, to ensure that this updating takes place.

Operations and the Operation Page

Conceptual Role and Purpose: To represent combinations of plant equipment: their support data, controls, and control tasks.

Detailed Role:

As indicated the Operation is the most complete, computational ICL object, defined in terms of its Pages, and in turn of its SuperVariables and Tasks. A new Operation is declared in an off-line listing by the **<<<OPERATION>>>** keyword declaring the new Operations Page as above. The remaining Pages are then declared, with their component elements, to complete the Operation's programming. The specification of each of these elements will be described in its appropriate section, as follows. The Operation Page is itself the header Page for the Operation, summarizing its States, Tasks, and subOperations.

Reference Conventions:

At any point in the programming of an Operation, any of its named direct component objects can be referenced (for assignment, computation, or Call): SubOperations, Tasks, variables (SuperVariable Zipper references) whose definition entries occur on its Definitions Page, Blocks, Page contents. In addition, unless a corresponding object has an identical name to an object defined in the Operation, any accessible objects in the Operation defining the current Operation as a subOperation are also directly accessible. Similarly any objects defined in any Operation defined as a parent Operation is accessible by name from the current Operation unless there is an identically named object in a lower level parent which would take precedence.

---

[30] The Operation in which the current program element (Task) is defined, or the Global (top level) environment; for manual or configured commands.

[31] There is a strong natural resistance in the industry to hierarchical naming strategies. This is a consequence of inexperience with the consequences of trying to write complex programs without strict hierarchical references coupled with a long history of highly disciplined use of tagging systems. I have considered allowing unique Operation names occurring anywhere in the Operation hierarchy to be used globally, and the specific definition of **GLOBAL NAME**s for SuperVariables. The more I consider, this the less secure I am with it. The specification already allows the definition of any desired Operations, SuperVariables, and Tasks at the global level and the free form naming of tags. Continuous processes can be operated without hierarchical reference, at the expense of losing their benefits in describing the process logically in terms of its parts. Multi-unit batch processes with multiple similar production lines become much more cumbersome if the Recipes cannot be defined about standard component names which are the same for every production line.

the expression and for any matched Parentheses, Brackets, and Braces within that expression. Other mathematically equivalent rules can be constructed.

- White Space (Space and Carriage Return indicated symbolically): ∆ (Space), _ (Underscore), © (Carriage Return; New Line).
- Ellipsis Symbols: two or more dots whenever they occur in succession without any separating White space: .., ..., ..., etc.

More complex or composite ICL symbols are defined, based on the following definition rules. The definitions assume a right to left character scan in which new Symbols are initiated by one set of conditions, and terminated by another set of conditions.

- Every Symbol is properly initiated if it occurs as the first Symbol in a Statement, or immediately (but for intervening spaces) after a properly terminated prior symbol.
- Every Symbol is terminated by the occurrence of any character not allowed as part of the Symbol.
- Spaces and Underscores are normally equivalent whether embedded in a Symbol name or occurring as a separator between words.[21] But statements are pretty-printed to remove any resulting visual ambiguity: Spaces/Underscores embedded in names are printed as Underscores. Otherwise they are ignored; Statements are pretty-printed according to separate rules.
- In ICL, the decimal point or period has a special dual role[22] as a decimal point and as a dot reference notation operator. The disambiguation is straightforward: If it is directly next to other digits[23] it is a decimal point, otherwise it is a dot operator.

The remaining definitions are:

- Number: a sequence of digits (at least one, other than a decimal point) containing at most one decimal point, and terminated by any other character (including a second "." [now a dot operator]), but not if without a decimal point and followed directly by a letter, or by one space and a letter or digit.

  For example: **123.456** in **123.456SAM**, or **24236** in **24236+SAM**, or **123.** in **123..SAM**, or **.345** in **JOE.345SAM**, but not **234** in **234SAM** or **123** in **123 456**.
- System Name: a predefined system name,[24] starting with a letter, followed by any number of letters or digits, terminated by any other character, but not if preceded or followed directly by an explicit underscore. In addition, any sequence of characters, not initiated but otherwise constituting a properly terminated System Name, preceded by a space, becomes a System Name, terminating any preceding Symbol.

  For example: **RAMP** in **123.RAMP.** or **123 RAMP .** but not in **_RAMP** , or **SIN123** in **234.SIN123.456**, but not in **SIN123_**.
- Keyword: a System Name, whose definition, as a System Name, is subject to additional restrictions:
  - Not immediately followed by a decimal point or period (dots) with or without intervening spaces[25],
  - Not followed directly by an equal sign with or without intervening spaces[26],
  - Not followed directly by a comma (Discrete assignment sign), itself not enclosed in matching parentheses, and
  - Not occurring between parentheses matched on a single line.[27]

  For example: **RAMP** if spaced apart from any other name, but not in **RAMP.SET** , or **SIN123** in **234.SIN123** , but not in **SIN123_** .

  Each Keyword is either Independent or Dependent as defined below.
- Independent Keyword: a Keyword, which is only defined as such when it occurs as the first non-white-space word in a statement (or after the colon in a Prefix, or after a semicolon separating two in-line statements).
- Dependent Keyword: a Keyword, which is only defined for statements initiated by a particular Independent Keyword.
- User Name: a sequence of letters, digits, and isolated (internal) underscores/spaces[28], including at least one letter or internal underscore/space, terminated by (but not including) any other character type, or by a terminating Keyword as defined above. Underscores isolated by spaces[29] are allowed to represent a "blank" or 'Null" User Name.
- Function (or Task) Name: a System or User Name which is immediately followed by a left parenthesis (itself not included in the Function Name), without intervening spaces. System Function Names are restricted to being defined only for the condition under which the left parenthesis is not spaced from the Name. The language will allow spacing of User Function Names from its following left parenthesis when no ambiguity results, but will not recognize such a usage at the lexical processing level.
- State Name: same as User Name but allowing, in addition, all ICL operator symbols (but not the slash '*/*', comma ',', or

[21] With the below indicated exceptions using underscores to set off User Names ambiguous with Keywords and to indicate "blank" User (or State) Names. Even within this exception they are represented internally as equivalent; the user Name is identical to the Keyword and the "blank" User Name is empty. The exception simply distinguishes the identically named cases as input or listed.

[22] Identical to that in C, but enforced both at the syntactic and at the lexical levels.

[23] And not repeated in the same Number.

[24] Since System Words are pre-defined in the system, this rule defines a restriction only affecting the system builder when he adds new System Words (also reflected in the implemented design of the parser). Its value is that it allows a single space to introduce or terminate any System Word. With this objective, System Words consisting of spaced multiple word elements (e.g. **MAKE_RECIPE**) (or otherwise freed to the full generality of the User Name Symbol definition rules given below) can be permitted if they occur as the first word in a statement (or after a prefix).

[25] Permits unambiguous User Name in dot reference expressions.

[26] Permits unambiguous first User Name in assignments.

[27] Further simplifies the distinction of User Names which may be lexically identical to Keywords, where their direct use is unambiguous within the inherent structure of the language.

[28] An isolated underscore or space is one that occurs by itself between letters or digits.

[29] Or by a preceding terminated Symbol. The number of underscores isolated does not alter the meaning of the returned Null Symbol.

for an incorrect assignment statement.[15] This simplifies the user's recognition of how the system will interpret a statement. This section addresses the lexical definitions.

ICL is designed to permit maximum freedom of user names.[16] On the other hand, one can follow the standard conventions (of FORTRAN or C) and not go wrong.[17] The definitions and rules below generalize those usages while supporting the following goals:

- The system should allow names of any realistic length that the user might chose (conveniently up to 128 or 256 characters).
- For readability, it should permit spaced multi-word names, using the space key (for ease of entry, but displayed with all internal spaces replaced by the underscore, for clarity. Thus: **REACTOR FEED FLOW** on input and **REACTOR_FEED_FLOW** on output.).[18]
- It should, from the opposite point of view, allow arbitrary user Tagging conventions, any combination of letters and digits (e.g. **1IAX23Q_42Z**, including pure number tags (like **123_456**) where these can be distinguished from numerical values.

More broadly, the priorities are: readability of the resulting code first, and then, maximum flexibility to support user conventions, accommodation of traditional expectations, and ease of program entry. Attention at this level is not our usual forte, but these issues have real material benefit. A large application may have thousands of process or variables names. Readable names can save days (**Those are real man-days!**) per reader in getting on board, whether achieved by using natural English or by conforming to a consistent user convention.[19] In addition, accessible names permit casual readers to understand the control program who would otherwise be shut out of that understanding. The core of the "software problem" is the inaccessibility of programming descriptions to all but the writer. The more people that can understand, the more casual redundancy available in real or accidental walk-throughs, the fewer errors in the result.[20]

Apart from the off-line archival form described above and the special on-line icons, all forms share the following character primitives:

- Letters: **A** ... **Z**. Upper and lower case permitted when available. But the case may be set to be ignored; the initial off-line keywords **<<<UPPER>>>**, **<<<LOWER>>>** and **<<<BOTH CASES>>>** control this state.
- Digits: **0** ... **9**.
- Decimal point: (in context, by that name) **.**.
- Arithmetic Operators: **+**, **–**, **^**, plus, in context, *****, ***/***.
- Assignment Operators: **=**, plus, in context, **,**.
- Comparison Operators: **<**, **>**, plus, in context, **=**, plus the particular composite operators: **<=**, **>=**.
- Highlight Symbols: ******, plus, in context, *****.
- List and Statement Punctuation: **:**, **;**, and **Tabs**, plus in context, **.** (called then a dot operator) and **,**.
- Footnoting punctuation: **!**, **!!**, **?**, †, plus, in context, *****, ******.
- Quotes: **"**.
- Parenthesizers (paired): **()**, **[]**, and **{}** (as available). List and Statement Punctuation, Quotes, and Parenthesizers always occur individually.
- Special Composite Symbol delimiters: **<<<**, **>>>**.

Parenthesis Rule: When used in later definitions this is the rule that says that, for a particular expression: The total number of **(**'s exactly equals the total number of **)**'s; and, between the beginning of the expression and any other point, the number of **)**'s will never exceed the number of **(**'s. Matching parentheses are parentheses whose contained expression obeys the Parenthesis Rule. The analogous rule for Brackets and Braces will be similarly named. The **Generalized Parenthesis Rule** requires that the Parenthesis, Bracket, and Brace Rules apply simultaneously for

[15] Many statements have an associated initial Keyword (after a Prefix). Such a statement is interpreted as being a (legal or illegal) statement of the corresponding type. Any statement which includes an equal sign, or comma outside of matching parentheses, not initiated by a valid Keyword is interpreted as a (legal or illegal) assignment of the appropriate kind.

[16] Subject to the earlier name uniqueness constraints.

[17] A general ICL design goal is to support casual users working from simplified user's guides with only those language elements that they need. For example, the following simplified naming rules work; the user can then learn the refinements in his own good time:

Casual User Naming Rules

- Normal FORTRAN alphanumeric names work (initial letter followed by any number of letters or digits). But generalized as follows:
  - Any pair of number of letters (up to some large limit).
  - Allows initial digit, as long as a letter is later included.
  - Allows single spaces between words and numbers, which are displayed as underscores: **FEED_FLOW**.
- Legal user names, keywords, and operators can always be unambiguously separated by a single space. For example, a decimal point can be distinguished from a dot operator, by the intervening space: **123.** vs. **123 .**.
- Any items (legal or not) can always be separated by a double space.

[18] On the other hand, long names should be used for names that occur mainly in short statements: simple calls. Complex statements (like algebraic expressions) become more readable if based on abbreviated names (like the process variable names in the Styrene Plant example). In support of this usage, ICL allows multiple names for SuperVariable referenced data.

[19] Other language features, such as the default sharing of variable names with their associated blocks or between Real and Discrete Attributes of a SuperVariable, are designed to support natural language usage, and minimize the need to learn arbitrarily redundant additional names.

[20] The necessary lexical processing routines have already been written and exercised in C.

components of the model Object must validly apply to the modeled Object. Within a modeling Operation Task definition, an argument list can pass data in a Task call if the corresponding SuperVariable contains the same elements as the model, with added States or Attributes only if these do not change the reference to existing ones. It may pass a Task, Operation, Call or statement name only if the the argument has all of the elements of the model, ??? and the procedural part of the Task or Operation is unchanged ??? .

The Language Concepts, Structures, and Elements, in Detail

Language Outline and Subject Index:

Basic Symbols and Naming Conventions (*8*)

Objects/Structures:

Operations and SubOperations (*11*)

Pages:

Operation*11* Definitions*14* Procedures*16* Details*17* Parameters*20* Summary*22* Idioms*24*
Comments*24* Simulations*25*

Tasks and Activities; Tasks with Arguments *25*

Activity/Task Termination Commands: *29*

END NEXTSTATE NEXT

Path Tasks *31*

Simplified Task Format for Operator/Recipe Usage; Sequential Function Charts *32*

Task Call Forms: *33*

Simple Calls*34* Block Calls*35* (Recipe Calls*36*) Idiom Calls*37* Path Calls*42*

Built-in Block Calls: *42*

PID*43* PIDX*43* PIDE*43* Ratio*43* FeedForward*43* FeedFwdX*43* LeadLag*43*
DeadTime*43* Function*43*

Object/Task Call Support Commands *43*

Make Recipe *43* UnMake *44* Archive *44* Restore *44*

Footnotes; Footnote References *45*

States and State Computation *46*

Prefixes; State Prefixes, Scoping Prefixes, Naming Prefixes and Postfixes *47*

SuperVariable, and SuperVariable Attributes *48*

Lists *51*

Data Types, Basic Computational Statements and SuperVariable Attributes: (*55*)

Real *56*

Assignment *56*

State Named (Discrete) *57*

Assignment*59* Truth Table *60*

Time (and Date) *62*

Counting *62*

Other SuperVariable Attributes *63*

Higher Statements and Element Forms: *63*

Messages *63*

Extended Calls; Local State Environment Declaration Expressions/Lists and Statements *63*

Theme Statements *65*

Wait*65* Ramp*66* Blend*67* Sequencing/ Timing*67* Stream*68* Profile*69*
…

Idioms and Idiom Loop Statements *69*

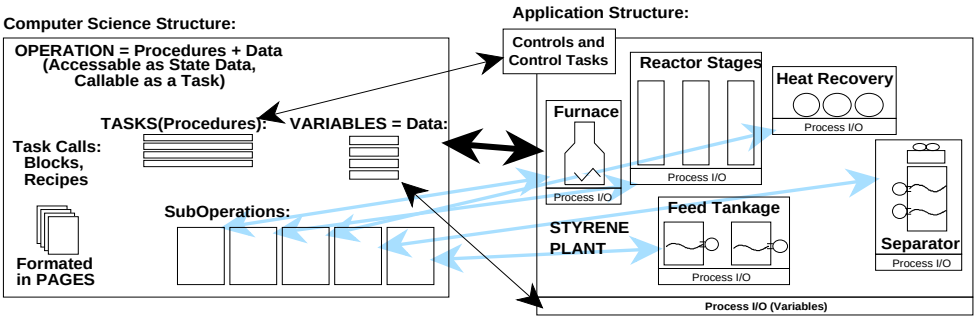
Regulate*75* RegulateX*75* Function/M*76* Feedforward*77* FeedfwdX*77* Hi/LoLimit*77* Hi/LoConstr*77*
Decouple*78* SplitRange*78* Backup*79* Blend*79* Combust'nControl*80* MOC*80*

Operator Executed Commands *80*

ICL Keywords, Symbols, and Names

In the interest of clarifying language error messages, the lexical (word forming) and syntactic (Statement analysis) specification of the language are sharply separated. For the same reason, the rules for recognizing statements of a given kind, then recognizing that those statements are error free, and and then parsing those statements in detail are also sharply distinguished; for example, ICL not only has a formal definition for a correct assignment statement but

Operation (and of its subOperations, ad infinitum), describing the parts of that process.[13] The different aspects of each Operation (and subOperation) are described separately in appropriately formatted Pages. The Operation includes the definition of all associated process Variables, defined as SuperVariables on a Definitions Page, and all associated control Tasks (computations), defined on a Procedures Page. Other Pages describe other aspects of the control computation. The mapping between the process structure and the ICL Objects maps Processes and their units to ICL Operations, Process I/O and Variables to ICL Variables, and Controls and Control Tasks to ICL Tasks and their calls.



The Operation is the basic ICL language object. It has three central roles: It is the main language object. It is the most powerful language object, containing data and function in their most general form, a "full featured procedure". And it is the basis for modeling of the process elements and their controls, within the language. All other objects provide data and function in more restrictive, less general ways. All other Objects are part of some Operation, dependent on it for some aspect of their operation. On the simplest plane, as data, its States may be set by an assignment, just as any other data value. But the Operation contains variable definitions which can be operated on by its own function or by any contained Task. As function, it can be called like any other procedure.

The Tasks within an Operation contain only function, and the necessary States to control and monitor that function; they may be called[14] ???What about setting System States of multiple Task calls of the same Task????, but directly only within their own Operation, and only directly acting on the data of Variables of that Operation. A Task definition (and the corresponding call) may also have arguments, providing data for it to act on. The variables and the special (e.g. Recipe and Block) calls within an Operation, contain the data of that Operation, as well as the essential functions to maintain that data: I/O scan, scaling, alarm, control.

Although a Task depends on its Operation for most of its data and all of its data definitions, it is still desirable to allow a single Task definition to apply to several similar Operations. The problem in so doing is to ensure that the Operations are sufficiently alike that they are consistent with that Task. To support this behavior, and in so doing, to allow the formal definition of classes of similar processes and process units, and the abstraction of process characteristics, ICL allows one or more Operation to be modeled after another Operation. When such a relationship applies, the modeled Operation contains by default all of the variable and Task if the model Operation. It also contains subOperations modeled after the subOperations of the model Operation. It may also be modified in certain restricted ways, containing, for example, additional variables, Tasks, and subOperations.

Operations have user defined names for external access. Of the Operation's major components, so do Tasks (for external call); certain Task calls: certain (Block and Recipe) calls, certain (Theme and Idiom Loop) statements, and SuperVariables. Of these, the Operation, Task calls, statements, and SuperVariables represented named control objects, with data and permanent operational existence. The named (Block and Recipe) calls and statement names have default values, overridden only by explicit user name assignment.

Effective naming requires that subOperation, Task, Recipe, and SuperVariable names be unique within any Operation (or within the Global Environment). The Block and statement names may be redundant with an associated SuperVariable name, but must otherwise be unique. Multiple Blocks associated with a single variable may have a distinguishing (unique) dot reference tag in their name: **F100. STARTUP**, **F100. RUN**.

Modeling, Inheritance, and Type Checking

ICL permits a free passing of arguments in its various calls. The concept of type checking envisioned is based on Object oriented inheritance thinking. Object inheritance allows an inheriting Object to add data and methods, but not to delete them, and to change method definitions but not change their call. This ensures that any message to the inheriting Object will always work. As translated to ICL, any ICL Object (Operation or its contained elements) may be *Modeled* after a similar Object (in a similar Operation); it may have additional elements, but any reference to

[13] Operations having subOperations are defined to be parents of those subOperations, and to be (higher level) parents of all subOperations whose parents are those subOperations.

[14] Alternatively, Tasks, not otherwise called from within other Tasks and Activities may also activated directly by setting their System State appropriately. In this sense, every such Task may be thought to have a built-in call.

visual/human perspective for readability and ease of use. The other representations serve to enhance the effective interpretive and system integrating character of the language.

Relation to Small Systems Versions

Currently the concept supports three simpler versions: a Single Loop Controller, a Local Equipment Controller, and an IA ICL Block. Each of these presumes an implementation which is simplified in some respects. All would eliminate any concern for the general, parenthesized, multivariable Idiom forms. All would be based on the SuperVariable scan for their computation, higher level functions being embedded in the SuperVariable as Footnotes. This permits a simpler data organization into a single data block, which is locally packed for editing. All would also reduce the hierarchical character of the language, and generally not include embedded Task definitions. The State Prefix mechanism would be used to provide the appearance of Task Calls within the resulting control object.

A Single Loop Controller would be organized about five kinds of listed structures: a single extended SuperVariable Format, the corresponding extended SuperVariable definition, simple computational assignment statements, Idiom Loop statements (with fan-out Idioms), and Parameter Tables for each Idiom. The Local Equipment Controller and IA ICL Block would replace the ICL Pages by a single Listing with separated Paragraphs corresponding to Pages: Heading Paragraph, Variables Paragraph, Footnotes Paragraph, Filters/Compensations Paragraph, Loops Paragraph, Theme Statements Paragraph, Procedures Paragraph. The Theme statement would be emphasized. Its configuration would be associated with corresponding configuration menus: Top Level Menu, Variable Format Menu, Variables (Definition) Menu, Footnote Menu, Filters/Compensations Menu, Loops Menu, Parameters Menu, Theme Statements Menu, Procedures Menu. The menus would be designed to lead the user through any desired edit or configuration step.

The IA ICL Blocks would emphasize including the basic computational capability of the existing HLBL language. But this would be extended to include illustrative elements of the other major language features, including Single Loop Idioms with fan-out. It would certainly include the all of the Activity Brackets (i.e. including parallel tasking) and State oriented logical capability. By its implementation it would support internal variables. It would probably not have internal Tasks, using State oriented approaches to accomplish the same function.[12] Originally the ICL Blocks were intended to operate independently; but it would not be difficult to expression hierarchical character by allowing the ICL Block to incorporate other Blocks grouped with it as Compounds. this would provide a Compounding capability in which the ICL Block "included" its subBlocks of all kinds, while also having its own computation. If this was done, it would support a full hierarchy of ICL Block Compounds of Compounds, etc.

The Language Elements and Structure, in General

The ICL design trades off conflicting goals: computer science goals of minimizing and generalizing the language elements, and the application goals of defining unambiguous process control programs. A general purpose language is designed with language elements that are as powerful as possible. But power comes from being able to support many roles, and function with many possible roles is function whose immediate intended use is thus ambiguous. The language accommodates the conflict by emphasizing the modeling of a minimum number of basic process and control information modeling objects, and providing general access and control of the resulting objects.

Specialized functions are included where these functions are universal to process control or where significant processing efficiencies occur if they are integrated in the language. Where specialized functions and low level actions on the modeled data elements are required (e.g. process or path booking for batch processes), the language generalizes these to support the broadest range of useful function without losing the clarity of language usages. Where specialized technology and expertise is required for the proper execution of a normal control function (e.g. Idioms), this expertise is modeled in the language in a way that clarifies its purpose in application programs. The language relies on external packages to address specialized, higher level operational perspectives, carried out through commands to the language elements.

As a computer language, ICL is designed to provide a full range of functional objects, from powerful and complex to restricted and simple. The range of objects is intended to permit user selected trade-off between the demands made by the particular application problem and the complexity acceptable to its specification. As an application language, each of these different functional objects is represented in terms of an expected application role. The expected roles of the different objects define a default attitude for the development of application programs and a basis for readable application programs, with predictable meaning, according to a holy writ. The computer science generality of these objects allows their perversion from writ when necessary to solve the actual application problem. The language will be described in terms of the intended conceptual model, but the accompanying computing role and structure of the objects described will be of equal importance.

ICL is a language for representing the control of processes at their different levels of unit or equipment. An ICL program consists of an Operation, describing the process as a whole, and any number of subOperations of that

[12] True ICL Block subroutine Tasks, particularly with reentrancy, would only make sense if the same Task could run in several ICL Blocks. This requires a separate Task store as well as some part of the ICL modeling structure to support type checking.

white space and comments freely to lay out his program in the clearest possible way.[10] These comments should not contain any explanatory data as they will be lost once the program is compiled or edited in the on-line system.

Keywords special to the Archival Format (translated to icons or other screen manifestations in the online version) will always be set off between triple brackets ("<<<" and ">>>"), with no optional internal white space. All other ICL text (keywords and symbols or user defined names) will take the same form in both formats. Edited white space will be lost in compilation, being reconstructed to the language system's pretty-printing rules when the Archival Format is regenerated. While the language permits distinguished upper and lower case text, the examples and keywords will use only upper case.

The Archival Keywords will generally be subject to strict syntactic combinations. For example, a new Operation (described shortly) will be declared archivally within the strict syntax:

<<<OPERATION>>> STYRENE_PLANT <<<USER STATES>>> STARTUP/ HOLD/ RUN/ EMERGENCY_SHUTDOWN/ SHUTDOWN <<<ENDO>>>.

When loaded into a full Work Station configurator, as part of a complete program, this statement would be translated to the Operation Page for the particular Operation [including the Task and subOperation names added (automatically) as these elements have been defined elsewhere]:

STYRENE_PLANT Page: Operation

SYSTEM STATES: CONFIGURE/SETUP/SIMULATE/ OPERATE,
RUN/SUSPEND/CONTINUE/ABORT/END,
ACTIVE/INACTIVE, BOOKED/ UNBOOKED, _/INITIALIZE

USER STATES: STARTUP/HOLD/ RUN/EMERGENCY_SHUTDOWN/SHUTDOWN

TASKS: EMERGENCY_STEAM_SHUTDOWN, FINAL_SHUTDOWN,
PREPARE_EMERGENCY_SHUTDOWN, PREPARE_STARTUP,
PURGE, REACT, SHUTDOWN_FEED, SHUTDOWN_TEMPER-
ATURE, SHUTDOWN_STEAM, STEAM

SUBOPERATIONS: FURNACE, REACTOR, HEAT_RECOVERY,
SEPARATOR, FEED_TANKAGE

The Listing Format includes automatically generated redundant information (from the system state data and other user entered subOperation definitions.[11] The Reverse Video indicates a running program with the indicated States and subOperations active. The entry of a same (new top level) Operation manually from the engineers Work Station configurator, would require selecting a New Operation (mouse) button to set up the window. The resulting skeleton Operation Page requests the Operation name, and shows the Page type, built in system States, including the reverse video-ed current states (as being configured), and underlined headings:

<<*Enter Operation Name Here*>> Page: Operation

SYSTEM STATES: CONFIGURE/SETUP/SIMULATE/OPERATE,
RUN/SUSPEND/CONTINUE/ABORT/ END,
ACTIVE/ INACTIVE, BOOKED/ UNBOOKED, _/INITIALIZE

USER STATES:

TASKS:

SUBOPERATIONS:

The remaining essential data could be entered on-line. The Entry Format would thus consist of the initial work station panel button selection of a new Operation, followed by entry of each item terminated with a carriage return (arrow keys allowing reconsideration of any section). This same data would be incorporated in the archival listing generated by the system, reformatted to maximize the readability, even though in Archival Format (The redundant data would be checked for consistency on re-entry.):

<<<OPERATION>>> STYRENE_PLANT
<<<SYSTEM STATES>>> CONFIGURE/SETUP/SIMULATE/OPERATE, RUN/SUSPEND/MANUAL/CONTINUE/AUTO/ ABORT/END, ACTIVE/INACTIVE, BOOKED/UNBOOKED, _/INITIALIZE
<<<USER STATES>>> STARTUP/ HOLD/ RUN/ EMERGENCY_SHUTDOWN/ SHUTDOWN
<<<TASKS>>> EMERGENCY_STEAM_SHUTDOWN, FINAL_SHUTDOWN, PREPARE_EMERGENCY_SHUTDOWN, PREPARE_STARTUP, PURGE, REACT, SHUTDOWN_FEED, SHUTDOWN_TEMPERATURE, SHUTDOWN_STEAM, STEAM
<<<SUBOPERATIONS>>> FURNACE, REACTOR, HEAT_RECOVERY, SEPARATOR, FEED_TANKAGE
<<<ENDO>>>.

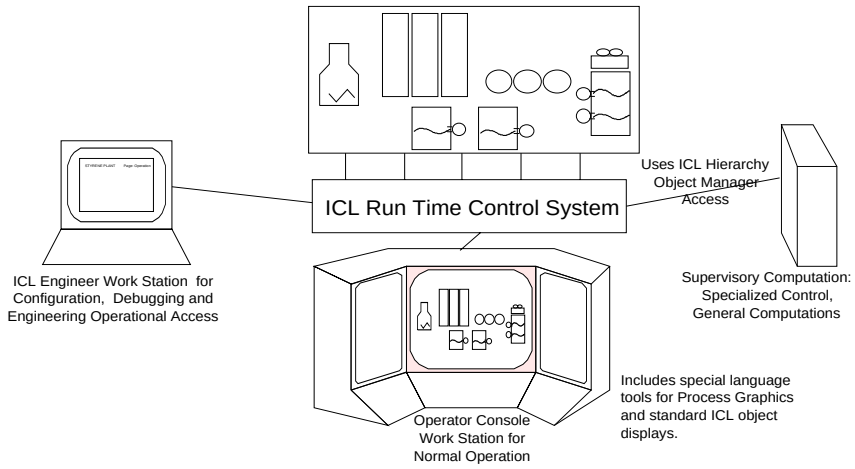
The Archival Format will thus define the language, with the Listing and Entry Formats representing the equivalent

[10] However, white space cannot include an underscore or carriage return in contexts where it would normally alter an ICL statement or word; a carriage return, internal to a statement, but not intended to end it, must be preceded by a back slash (\). Comments are set off as in C.

[11] The full Listing Format specification, for any Work Station, defines: the font size and style, of each entry, the precise position of the Operation Page, and entry names, the maximum number of characters in a line, the position of line continuations, etc. This level of detail is deferred to a final implementation but it should be made explicit at that time.

temporary result of an on-line command.

The operator interface is a separate software system, which should ultimately evolve to have the same data access to the programmed control system as the configuration Work Station. But it would have its own language/tools for supporting graphic process representations and operator interfacing. It would use the ICL access hierarchy to locate and update control data for its own operating displays. These displays would include standard displays for monitoring different levels of process and control: loop displays, unit displays, process summary displays, or standard tasks: recipe management. They would also include special user defined displays for special tasks; e.g. recipe management.



Computing Resources Requirements

The external resource requirements of the Language system as described can be divided as follows.

- GUI to construct graphic Engineer Work Panel. The associated software to support these panels should not be major. (On the NeXT it would be about two man weeks to create the described panel.)
- Operator's Work Panel. This is one of the major opportunities presented by ICL, still requiring its own engineering to what ever level we determine.
- PostScript. The language listing depends on a clean graphic screen/printer language to generate ICL listings. This would be best supported by Display PostScript which is available on the Sun. This would supply a single format for screen and printed listings. Failing this the bit map capability seems adequate, to support the screen.
- Process I/O, to be provided through IA, but the full SuperVariable language support with I/O Attributes would require a separate treatment.
- "Object Manager" function. This requires a rewrite of the Object Manager to support the more general ICL access. Ideally this would be based on the Object Manager's locating the main CP location from the top level of the reference, and then turning over the rest of the interpretation to the ICL system to get the actual program data.
- Sampling time computation. The ICL interpreter would be run at the normal block sample time, taking as much time as needed or available each sample time. However, for most computations, it would adjust its internal sample time to whatever multiple of this time was needed to complete the sample times computation. This adjustment would be dynamic as needed, with operational notification of the current sample time; the control algorithms would adjust themselves to the changes. As a consequence, for the normal application there would never be a situation where the system flat sample time overloaded; it would just sample less frequently. For several reasons, more compact algorithms, less redundant code, more appropriate computational strategies, the system would normally carry out computations faster than our current systems.

Apart from these considerations, the language system itself should fit in 64k, and making use of the normal IA/Sun file system.

Relation Between Program Representations

From the normal language specification point of view, the most visible program representations are those for graphic display (the Listing Format), off-line editing (the Archival Format), and keyboard entry (the Entry Format). Normal language specification tends to emphasize a compiler oriented view. But ICl is intended to reflect a more interpretive on-line character. This distinction is involved in these three distinct forms. The Entry and Listing Formats emphasize the on-line character, whereas the Archival Format form best reflects the off-line character. At the lowest level the Listing Format reflects the specialized graphic icons, but its more important effect is in organizing the display of program elements in window related pages. These special iconic or organizational forms are only visible in the on-line versions; their indication must be converted to keywords for pure text off-line representation.

For this reason, the Archival Format will be closest to traditional keyword based compiler specification (e.g. Bachus Naur form), and the natural starting point for formal syntax specification. The Listing Formats can then be defined as they would be translated from the Archival Format. For this reason we will first highlight key elements in the Archival Format. This format can be generated directly by a user on an off-line editor. In that case, the user will be able to use

The specification will include:

- Specification of the language elements, as normally expressed, in the intended graphic, iconic, configurator window listing formats: the Listing Format.[7] In this respect, the configurator representations and paper listing format should be developed together, and reflect such aesthetic concerns as pretty-printing principles and automatic page numbering.
- Specification of the intended (ASCII compatible) keyboard usages: the Entry Format. Because of the use of special icons and icon/text layouts, natural keyboard entry strategies for controlling the icon placement are needed.[8] Each such key action generates immediate feedback which is translated and displayed in the more natural icon form as soon as possible.
- Specification of a standard ASCII counterpart: the Archival Format; suitable for editing in an off-line editor.
- Specification of the file format for external data access: the Data Format. This is the format (corresponding to the C printf and scanf formats) for any data requests from outside (C) programs. It is necessary to allow the language system to manage data access. This format will be outlined in an appendix.
- C or C++ coding standards to permit user written, C coded general Tasks, Blocks, and Idioms.
- Outline of a compiled binary core and file format, for fast reload, analogous in function and format to the Checkfiles: the Object Format. Because it will be compiler implementation dependent, this format will be outlined only in an appendix.
- Intended Engineering Work Station and Configurator environment. This will describe the desired information to be provided in the different support windows and menus of the intended Work Station configurator, without insisting on any particular display formats (beyond the already defined listing formats).
- Intended Operator Work Station. This will describe the desired information to be provided in the different configuration support windows of the intended Operator Work Station Console, without insisting on any particular display formats.
- General implementation discussion. The overall language is intended to permit implementation as a big system within IA, and as little single loop and small multi-loop product elements. The implementations of these smaller systems is expected to be different from the large system design, being based on a single SuperVariable embedded in a limited memory. This same implementation is envisioned for an initial form of IA ICL Blocks, where the memory limitations are now required to keep the full ICL program within a single Block. In the long run we would like it to be feasible to accommodate any customer desires to second source ICL capability. This requires that the language be portable. For process control applications this requires the kind of isolated Run Time Control System shown in the Figure, allowing the separate implementation of the language processing system, the general computational capability, and the proprietary operating system and process I/O processing. This later function can be the only part of the system that needs to be particularized to the competitive process control platform.

Recall that the displayed format of an ICL program is intended to be computer determined (pretty-printed) without user controlled white space or embedded comments (explanatory comments are separately controlled on the Comments Page). This requires that every one of the specified formats is fully translatable to every other, automatically, unambiguously, and without loss of information. The language is implemented in an interpretive format, so that the data is directly accessible for operational purposes, but back compilable to the natural Listing Format. This same internal format is executed by the ICL Run Time Control System which will supply the access to the process I/O, and the operating system to carry out the different (Sequential, Parallel, Looping, Continuous, State Driven) Activity execution orders.

Relation Between ICL, the Computing Environment, and Work Stations

ICL is intended to evolve piecemeal within the existing I/A system until it addresses all of the configuration issues of systematic process control. To do this, it is provided with language structures for representing process structure, control procedures, and working control functions. The role of ICL in a complete system is to provide the control system description and the resulting computational control objects,[9] and to support effective operational interfacing to them. The resulting control programs then run in a larger environment which includes the ICL run time function, but also includes the Work Station software for configurating and compiling the ICL, higher level supervisory programs which may schedule or otherwise modify the operation of the control programs, and an operator interface.

The configurator software supports the creation of process descriptions and control programs, the compilation of these programs, and their debugging, including simulated process elements. It can also function on-line, providing engineer Work Station access to the running system. In this case, it will access any described process unit, set that unit in any appropriate operating state (including Simulate), and interpret and execute any appropriate ICL command applicable to any processing element. Within the process hierarchy and Page window framework, it can monitor and tune any process/control language element in real time, whether that element is a permanent part of the program or a

[7] The Styrene Plant example listings were generated in PostScript. The iconic listing specification process should eventually include specified listing PostScript routines and usages (or the equivalent for any other chosen graphic display implementation vehicle).

[8] For example:

- The Brackets are entered as open left parentheses with repeated carriage returns causing a cyclic selection of the appropriate bracket icon; the Brackets are closed by the matching open right parenthesis.
- Diamonds associated with an **END** command, initially positioned to the left of the most nested Bracket, are similarly moved (to the left) by repeated carriage returns (an arrow key could also be used). The start of the new lines text would complete the selection.
- Loop statement idiom subscripting is indicated by entering the "/" key.

These alternative uses of carriage return are legitimate since the user is not allows to control white space use in the program.

[9] Continuous control, logic, sequencing; but not general programming or supervisory control, not optimization or scheduling.

control procedure concept is needed to define controls which may be run permanently, as regulatory controls, or intermittently, as alternative continuous controls, or temporarily, as startup/shutdown or recipe procedures.[5] For security reasons, these procedures must be restricted to operation on the process units (either real or generic) for which they were designed. Accordingly these procedures should be included as part of the definition of their associated process unit, dependent on it.

A procedure definition, by itself, is a passive object, defining only the potential for action, not the action. It becomes active only as it is activated by some manual operational act or some programmed statement. In conventional programming, this activation corresponds to a subroutine call, which may be supported by argument list data. The control counterpart may take several forms. Because a control procedure is defined specifically in terms of the associated process unit and its process variables, it may be called (e.g. as a startup procedure) without arguments, all variables directly related to the particular process unit. But a more generally designed procedure may be defined with an argument list, to carry out a broader function in which there may be some user discretionary parameters (e.g. as in a recipe with its formula).

Process control related argument lists, are likely to be much more complex than normally encountered computationally (if containing all the variable elements in a recipe or all the parameters in a control block). And unlike conventional computational argument lists, process control argument lists and calls may be named, for operator access. In this case, their full operation will need sophisticated parameter entry and passing structures, tying the parameters to data structures in a calling program or in an operator interface. A recipe is inherently tied to a process related task. Its call represents a transient action by the operator from outside the process and control system, terminated naturally whenever the batch is completed. The same structure can support system and user written control blocks. In this case, the procedure is the control algorithm and the call is more permanently installed and executed in the control program. Its argument list now constitutes the block parameters, subject to operator call for display and tuning.

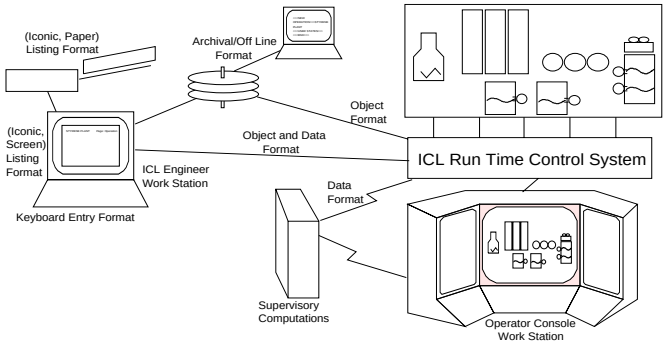
One central consequence of all of this is to define a control program as a single enormous data and reference structure, permitting computational and operational access to any desired process data, function, or call. The reference from within an ICL program looks like any C structure reference: **STYRENE_PLANT. FURNACE_UNIT. FUEL_FEED. SET**. This is another difference between the Process/ICL object view and the Object Oriented view: the process structure is designed to support structured access to an open structure; Object Orientation is designed to hide data.[6]

In any case, the above discussion defines the three basic kinds of process entities needed in a fundamental process control language:

- Objects or entities representing the parts of the process, including all data and control procedures, hierarchically organized according to containment.
- Entities representing the computations or control procedures appropriate for different parts of the process, including: simple process control procedures; partial or complete, generic or specific, recipe procedures; block algorithms.
- Entities or objects representing the parameterization and history of a given execution of a given procedure call, including: simple control task calls, recipe calls with formula and lot record; block execution calls, embedded in a control program.

Objectives

The specification will be complete, relative to the actual language elements, and suggestive, relative to the expected programming environment:



[5] In ICL, these are called Tasks.

[6] The structure of Object Oriented Programming has evolved from a history different in detail from the requirements of process control.

- Objects were developed to manage CRT displays and to manage different kinds of broadly similar, dynamically created graphic objects. Such an environment benefits from classes and inheritance, and has no need to structure the graphics in terms of its parts. This is distinct from the hierarchy of process elements.
- The separate definition of classes and objects allows a compiler based system, in which program elements (in the class definitions) can be pre-compiled, whereas the objects can be created dynamically at run time. Process control has a more interpreter based history. This allows for the interpretation of full language expressions on line.
- As indicated, Objects favor hiding, process control needs general data access.

Large System ICL; Rough Specification (Last Printing 1/9/94; Edit 10/13/95)

Content Outline

Background *1*
Objectives *2*
Relation Between ICL, the Computing Environment, and Work Stations *3*
Computing Resource Requirements *4*
Relation Between Program Representations *4*
Relation to Small Systems Versions *6*
The Language Elements and Structure, in General *6*
Modeling, Inheritance, and Type Checking *7*
The Language Concepts, Structures, and Elements, in Detail *8* **(with further indexing)**
Intended Engineer's Work Station *82*
Intended Operator Interface Function *83*
Appendix I. Data File Access Formats *85*
Appendix II. Binary Core/File Formats *85*
Index of Command Keywords and Symbols *86*
Index of Concepts *86*

Background

Idiomatic Control Language is designed for the integrated modeling and configuration of control systems, to emphasize ease of use and system integration. As currently evolved, it supports variants optimized for different levels of operational sophistication, from field devices to plant level control.[1] This document addresses ICL from the large control system focus, appropriate to continuous and batch plants, serving as a companion document to the Styrene Plant Example document and drawing from it many of its listing examples.[2] The Specification is not an substitute to these companion documents to understand the language; the companion documents should be read first.

The language structure is based on the process, to define a reference structure for the plant and its parts within a single program. From a process structure point of view, each process element is defined by a name, by its named components and by the named process variables whose I/O provides the process state data to the control system. The associated active computational elements are either activities dedicated to controlling the process, or I/O operations and conversions (including alarms) associated with process variables.[3]

Process control systems require the kind of information modeling capability suggested by Object Oriented Programming, but not precisely the same modeling structure. Conventional Objects emphasize a hierarchy of variant classes, about which Object instances are modeled. In contrast, process models need to emphasize a reference hierarchy of plant or process components, as described above. The classic continuous process is typically built up out of the minimum number of unique process units, each sized to accomplish the intended production scale. Each Object is thus unique, not an instance of some more general Class.

However, particularly in Batch processing, there are plants which achieve their production volume (and flexibility) by operating several similar production lines, or similar process units in parallel. In this case, ICL permits definition of abstract or generic process elements (like Classes), from which real (Instance) process elements can be defined about a minimum common model. ICL thus emphasizes the modeling of individual process units in a plant hierarchy[4] while permitting abstraction to generic units, in contrast to Object Oriented Programming, which emphasizes the abstract Objects, subordinates the instances, and subordinates any component hierarchy into the methods and pointers associated with particular Object definitions.

The classic process control system envelops the process with a regulatory control system which can almost be viewed as part of the process. Control blocks are based on this perspective, biasing toward continuous control in its most rigid implications. Modern control, and particularly batch control needs a more flexible view of control. A

[1] Inserted Sections and Footnotes will define related small system usages to indicate needed accommodations. Current thinking envisions a transitional IA block form based on the smaller system versions.

[2] The example illustrates the integration of the different facets of control. Other documents illustrate the details of other aspects of the language:
- "Idiom Loop Statements"
- "Idiom Loop Statement Demo in BASIC"
- "Improved State Diagrams for Documentation, with ICL Asides"
- "SuperVariable Process Data Definitions"

[3] As developed later, ICL models the process with the Operation, and the variables and I/O with the SuperVariable.

[4] Current thinking eliminates the hierarchical character for the small system versions and for transitional ICL I/A blocks. In that case, the individual controlled process elements are independent of each other, expressed as completely independent but mutually communicating process control "Objects".