- If a RAMP Theme Statement Footnote is terminated by a semicolon, the following line contains a RAMP Theme Statement Footnote.

Other usages (such as Idiom Loop Statements) are controlled by the choice of configuration Paragraph or, alternatively, the special Archival Format keywords:

<<<BLOCK>>> **<<<KEY NAMES>>>** **<<<DEFINITIONS>>>** **<<<HEADINGS>>>** **<<<ENTRY>>>** **<<<FOOTNOTES>>>**
<<<FILT/COMP>>> **<<<LOOPS>>>** **<<<IDIOMH>>>** **<<<THEME STATEMENTS>>>** **<<<PROCEDURES>>>**
<<<SEQUENTIAL>>> **<<<PARALLEL>>>** **<<<LOOPING>>>** **<<<CONTINUOUS>>>** **<<<STATE_DRIVEN>>>**
<<<ENDSQ>>> **<<<ENDPL>>>** **<<<ENDLP>>>** **<<<ENDCT>>>** **<<<ENDSD>>>**

normal (more natural) human usage and allow User Names, which are identical to these Keywords, to be used unambiguously without any associated underscores:

- Not immediately followed by a decimal point or period (dots) with or without intervening spaces[55],
- Not followed directly by an equal sign with or without intervening spaces[56],
- Not followed directly by a comma, itself not enclosed in matching parentheses (Discrete assignment sign), and
- Not occurring between parentheses matched on a single line.[57]

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 (including multi-line Theme Statements) initiated by a particular associated Independent Keyword.
- User Name: a sequence of letters, digits, and isolated (single internal) underscores/spaces[58], 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[59] 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.
- State Name: same as User Name but allowing, in addition, all ICL operator symbols (but not the slash '*/*', comma ',', or 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 or preceding 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**.[60]

With these definitions the lexical classification of (correct or incorrect) statements might proceed subject to the following rules:

- A legal statement starts after a carriage return, colon, or semicolon, or after the start of an Activity.
- A Prefix is indicated if a statement includes a colon, and consists of everything from the start of the statement to that colon.
- If a Prefix starts with the Keyword IN it is a Scoping Prefix; otherwise it is State Prefix or Truth Table entry.
- If the immediately preceding statements are State Prefixed and terminated by semicolons back to a Truth Table Heading, the Prefix is a Truth Table entry.
- Otherwise, if a statement includes an equal sign it represents a Real Assignment.
- If a statement starts with a (square) bracket it is an LSED.
- If the statement has an Initial TT Keyword it is a Truth Table Heading.
- If the statement has an Initial RAMP Keyword it is the first line of a RAMP Theme Statement.
- Phrases within a RAMP Theme Statement must have Dependent Keywords as their first word. These determine the Phrase form: FOR, TO, FROM, RAMP. Certain Phrases may have secondary Phrases, terminated internally by commas, and also having a Dependent Keywords as their first word: TO after FROM, IN after TO, and FOR after AT.
- RAMP Theme Statement RAMP Phrases terminate with a colon, and all other RAMP Theme Statement Phrases except the final one terminate with a semicolon.
- If a RAMP Theme Statement (or RAMP Theme Statement Footnote) is followed directly with a statement starting with an asterisk or dagger the rest of that line contains a RAMP Theme Statement Footnote statement of type determined by the above rules.

[55] Permits unambiguous User Name in dot reference expressions.

[56] Permits unambiguous first User Name in assignments.

[57] 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.

[58] An isolated underscore or space is one that occurs by itself between letters or digits.

[59] Or by a preceding terminated Symbol. The number of underscores isolated does not alter the meaning of the returned Null Symbol.

[60] This might be argued as being incompatible with allowing output to other (FORTRAN, C) language readable files. However the resulting translation is easy to impose, and ICL is not expected to output directly to general text files. We can argue the point.

Carriage returns (indicated as ©) terminate statements. Spaces (indicated as Δ) and underscores have a special dual role in the definition and termination of user names; they represent the same symbol, but spaces internal to names will be indicated as underscores. ICL user defined names allow any combination of letters, digits, and single internal spaces (including purely digital names with spaces, as in **123_456**). In greater detail the following rules apply; ICL symbols include:

- Letters: **A B C D E F G H I J K L M N O P Q R S T U V W X Y Z** (Lower case letters are assumed converted to upper case in the Small System Language.).
- Digits: **0 1 2 3 4 5 6 7 8 9 .**(as Decimal Point).
- Operators: **+ - *** (as Multiply Operator) ***** (as Footnote Operator) **/ ^ = ,** (as a Discrete Assignment Operator) **.** (as a Reference [or Dot] Operator).
- Comparison Operators: **< >**
  plus (in context): **=**
  plus the particular composite operators: **<= >=**.
- General Punctuation: **: , ;**.
- Definition Punctuation (in context): **/**.
- Idiom Punctuation (in context): **/.**
- State and States Expression Punctuation (in context): **/ ,**.
- Quotes: **"**.
- Parentheses/Brackets: **( ) [ ]** (Left and Right Parentheses and Brackets will always be matched).
- Special Composite Archival Format Symbol delimiters: **<<<**, **>>>**.

The composite symbols below all must be properly initialized, either by starting immediately at the start of a new line (or after a carriage return), bypassing any intervening spaces or underscores; or by starting after a properly terminated prior symbol, bypassing intervening spaces or underscores. A composite symbol is terminated by the occurrence of any character (or System Name) not allowed as part of the symbol.

The composite ICL symbols are defined below. 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 or underscores) 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.[50] But statements are pretty-printed to remove any resulting visual ambiguity: Spaces/Underscores embedded in names are pretty-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[51] as a decimal point and as a dot reference notation operator. The disambiguation is straightforward: If it is directly next to other digits in a Number[52] 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, 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[53]. 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.[54]
  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. These restrictions reflect

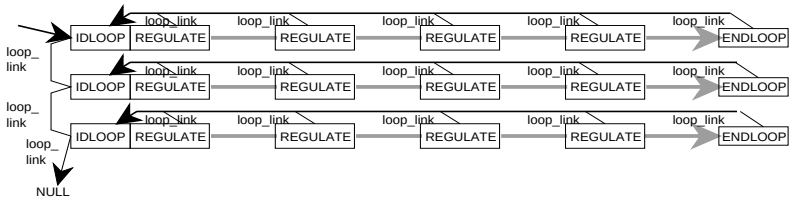
[50] With the indicated exceptions using underscores to set off User Names ambiguous with Keywords and to indicate "blank" User (or State) Names. Even within this exception they are represented internally as equivalent; the user Name is identical to the Keyword and the "blank" User Name is empty. The exception simply distinguishes the identically named cases as input or listed.

[51] Identical to that in C, but enforced both at the syntactic and at the lexical levels.

[52] And not repeated in the same Number.

[53] System Names cannot contain spaces or underscores. This allows User Names to include pieces identical to System Names, even when spaced (pretty-printed as an underscore). The restriction on preceding or following underscores extends the capability to allow User Names which are identical to System Names in those rare cases where it may be required. The obscurity of this restriction is felt to be reasonable because normal good practice will not use User Names which are identical to System Names. On listing, a following underscore will be pretty-printed (like the pretty-printing of embedded spaces as underscores) to disambiguate any such user name.

[54] 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 out regard to any internal spacing in preceding User Names. This objective is still preserved if the above definition is extended to allow 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) if they occur as the first word in a statement (or after a prefix).



The run time execution involves not only the forward passage of data from I/O, and the variables and their associated setpoints, but a reversed order backward computation, and the use of scaling Attributes to maintain a neutral scaling within the algorithms. As indicated earlier, the pcode computation thus has two parts, the forward and backward executing parts. Each IDLOOP pcode element includes its own stack which is used to control and pass data to this backward executing part. The normal I/O buffers also pass the variable Value, setpoint, and scaling data, to support both forward and backward computations. The basic Idiom control parameters are stored in the main pcode element.

Theme Statements; Ramp Statement

Each Theme Statement has its owns special structure of Phrase and Footnote elements. For example, the Ramp Theme Statement has:

- The initial or main RAMP (RAMP T101) Phrase,
- The subordinated RAMP (RAMP T102) Phrase, and
- The actual ramping Phrases which contain:
  - A FOR subPhrase (to indicate the length of time for which a constant value is to be held) or
    IN subPhrase (to indicate the length of time for which a ramping action is to be carried out),
  - An optional AT subPhrase (with a [following] FOR subPhrase) to indicate the constant value which the variable is to be held
    or
    An optional TO subPhrase (with a [following] IN subPhrase) to indicate the initial value from which the variable is to be ramped, and
  - (In the case of a [following] IN subPhrase) A TO subPhrase to indicate the final ramped value in the Phrase.

```
RAMP T101:          FOR VENTTIME;†[1]
                    TO COOKTEMP+BIAS, IN HEATTIME;
                    AT COOKTEMP, FOR COOKTIME;
          RAMP T102: TO 0*[2], IN COOLTIME
```

---

†[1] T102 = COOKTEMP

*[2] T101 = COOKTEMP * (T102 / COOKTEMP)^2

As a matter of convenience, the subPhrases are grouped to form the following pcode elements, each corresponding to an action completely describing the behavior within a single interval of time:

- The main RAMP pcode element, defining the initial ramped variable,
- The subordinate RAMP2 pcode element, used to indicate later ramped variables,
- The RAMP_FOR pcode element, defining a hold of the current value for a specified interval of time,
- The RAMP_AT_FOR pcode element, defining a hold of a specified constant value for a specified interval of time,
- The RAMP_TO_IN pcode element, defining a ramping to a specified value for a specified interval of time,
- The RAMP_FROM_TO_IN pcode element, defining a ramping from a specified value to a specified value for a specified interval of time.

The RAMP_FOR, RAMP_AT_FOR, RAMP_TO_IN, and RAMP_FROM_TO_IN pcode forms can each support up to two (optional) Footnotes. The first, with its indicating asterisk embedded before the final subPhrase, describes an action to be repeated every sample time as long as the Phrase is being carried out. The second, with its indicating asterisk positioned at the end of the Phrase, describes an action to be executed once when the Phrase terminates to move on to the next Phrase. The Footnotes are listed at the end of the Statement, but their pcode elements are positioned right after the associated Phrase pcode, the terminal pcode positioned last. A ftnt_code element in the Phrase pcode element indicates which Footnotes have been defined. Compilation consists of generating the pcode elements and Footnotes and merging the appropriate pcode elements in the appropriate order. Back Compilation requires that the Statement and Footnote pcode elements be Back Compiled themselves and sorted for the correct display.

ICL Small System Symbols and Names

The rules for defining basic language Symbols below are designed to allow the classification of statement types before those statements are analyzed in detail. Thus the statement may be recognized as a given type even if it is incorrectly formed in detail. This simplifies error messages. ICL symbols include: letters, digits, operators, punctuation, quotes, parentheses, system Keywords, numbers, and user defined States, units, and variable names.

corresponding length element is used to control the loop which sets the variables to the assigned value.

Footnotes and their Relation to SuperVariables and Including Statements

As indicated earlier, Footnotes have a language and in implementation role. As an implementing concept, the Footnote represents the concept of interspersing the control computation pcode elements between the SuperVarable Attribute elements at their point of scan execution. At a language level there are Filter/Compensations (with a specialized role)_and general computing Footnotes. The general computing Footnotes exist in two contexts:

- As Footnotes in the Footnotes Paragraph, and therefore Footnotes associated with an associated SuperVariable Attribute (marked by a numbered asterisk (or dagger)). Footnote Paragraph Footnotes are translated to pcode elements occurring directly after the corresponding Attribute. The Footnote pcode can consist of any number of pcode elements as long as they are the natural translation of one or more complete statements or Activities (not including Attributes, Idioms or Idiom Loop Statements). In compilation, each numbered Footnote is translated from the paragraph and inserted after the corresponding asterisked Attribute. In Back Compilation, all SuperVariable Footnote references are reinserted in the Definitions (SuperVariable) Paragraph as superscripted numbered asterisks. The Attribute Footnotes, themselves, are then collected and Back Compiled from the SuperVariable section of the pcode string, and grouped consecutively with preceding asterisk and Footnote number superscripts in the Footnotes Paragraph. A Footnote can contain multiple statements (each on its own line) and Activities (grouped collectively after the single numbered Footnote asterisk or dagger), with the corresponding pcoded structures occurring consecutively between the appropriate pair of Attribute pcodes.
- As Footnotes to a Theme Statement. Certain Theme Statement Phrase forms and corresponding pcode elements may support Footnotes. In this case, the pcode element will include a code field whose value indicates the nature of any following associated Footnote pcode structures. The Footnote will consist of a single statement or Activity. On compilation, the Footnote references are recognized by manually entered asterisks (or daggers) in the appropriate positions in the main Theme Statement text. The computational meaning of each reference is defined as a Footnote preceded by a corresponding numbered mark (asterisk or dagger) in the Footnote area located below the body of the Theme Statement; the individual Footnotes fall in the same order as their indicating marks and both references and Footnote definitions are numbered correspondingly. On compilation, the Footnote is translated and inserted immediately after the corresponding Phrase pcode; the Phrase code field is set appropriately. On Back Compilation, the Footnotes must be sorted from their pcode location, numbered automatically (with asterisk or dagger, as appropriate),[49] and grouped together at the end of the complete Statement (set off by any standardized legend line or spacing).

Filter/Compensations Footnotes

Filter/Compensations Footnotes apply only to associated Attributes. In that application they behave like regular computational Footnotes. (Indeed the two forms of computation and pcode can be mixed in a single Footnote, the associated Paragraph being assigned based on the first Footnote form.) Each Filter/Compensations Footnote statement consists of a single Filter/Compensations Idiom, programmed, each on a separate line, as the single name of that Idiom. Each Idiom translates to an individual pcode element. For these Footnotes a dagger replaces the asterisk of the general Footnotes in references and definitions. With this exception, compilation and Back Compilation are similar to the compilation and Back Compilation of general Footnotes. Filter/Compensations Footnotes and general Footnotes are numbered separately. As indicated earlier, each Filter/Compensations Idiom pcode elements takes input data from the meas_buf I/O buffer, carries out the desired compensation, and returns the result to the buffer. In this way the Footnote adds its computation to the existing stream of I/O Attribute computations eventually setting the variable Value Attribute.

Simple Idiom Loops and Idiom pcodes

Idiom Loops are expressed as sequences of process variables each with a set of subscripted Idiom operators:

$$T101_{\substack{\text{REGULATE [1]} \\ \text{FEEDFWD F50 [1]}}}\ F101_{\text{ REGULATE [2]}}V101$$

In the ICL IA Block Small System Language the variables correspond to a SuperVariable definition line. The Idiom computations will make use of the I/O, Value, Scaling, and Set Attributes, and so the Idiom Operator pcodes are inserted directly after the corresponding SET Attribute. The SuperVariable definitions must be ordered in the same order as any intended variables used in any intended Idiom Loop Statement (and vice versa). As a consequence, compilation of Idiom Loop Statements consists of locating the associated SuperVariable definitions, checking that they are correctly ordered, and inserting the appropriate Idiom Pcode structures after the corresponding SET Attributes. The Back Compilation consists of locating the variable definitions in a given Loop and, for each variable, the associated Idiom operators, and then listing them all.

The system allows overlapping Loop/SuperVariable structures. Each such structure is set off by a corresponding Idiom Loop (IDLOOP) pcode entered before the first Idiom in the Loop (after the SET Attribute), and terminated by an End Loop (ENDLOOP) pcode element entered immediately after the Value Attribute of the final (manipulated) variable in the Loop (or after each final variable in a fanned out Loop). Each Idiom and End Loop element (and variable) is associated with its initial Idiom Loop pcode element by a pointer to that element in its own pcode structure.

---

[49] The asterisk is used to indicate a Footnote which is run continuously for the interval of time that the Phrase is active, whereas the dagger is used to represent a Footnote which is run once at a transition in operation. In this case the two kinds of Footnote are listed intermixed, and numbered together.

Real Expressions and their Relation to Including Statements[45]

The system includes a Real Expression pcode element to represent normal single real valued statement/expression computations:

A * 23 * (4 + C / sin(X + Y)).

The expression is programmed by the user in the normal infix form as shown, constrained as follows:

- Supports the normal binary operations: addition/plus (**+**), subtraction/minus (**–**), multiplication (*****), division (**/**) and raising to a power (**^**).
- Supports unary minus (but not if applied directly after another operator as in **A*-B**; use **A*(-B)** instead), but not unary plus.
- Supports sine (**sin**), cosine (**cos**), natural log (**ln**), and exponent (**exp**) (all small letters).
- Supports nested parentheses; although the expressions are assumed to be computed from a compiled reverse polish, a dummy function is used to represent parentheses as entered by the user. The parentheses are Back Compiled as entered by the user.

Real Expression pcode Implementation

The pcode element (REXPR_CODE = 54) is designed to contain a translated reverse polish representation of a normal real valued (infix) expression. The pcode computes the result of the expression. The representation takes the form of a sequence of REFERENCE structures[46] and operator codes (see pcode.h, pcode.c, ICL Block Individual Routine Descriptions, and Introduction and Data Structuring Practices for the ICL IA Block).[47] The REFERENCE structures are distinguished by an initial code less that OP_CODE (= 192). All higher codes correspond to operators or functions: 192→+, 193→–, 194→∗, 195→/, 196→^, 197→dummy function operator (to represent parenthesis), 198→unary minus, 199→sin, 200→cos, 201→log, 202→exp.

Codes lower than 192, composited out of packed code fields, according to the earlier REFERENCE struct discussion, distinguish the nature of the values being operated on:

- Real data (0 with mask 15) distinguished from other data types.
- Constant data (0 with mask 192) distinguished from other expressions, the operators themselves, or
- Reference pointer (64 with mask 192). Any such reference pointer points to the actual value of the intended data, to support quick computation.

These pointers are further distinguished, to facilitate Back Compilation, in terms of:

- Direct References (0 with mask 48), pointing directly to some program referenced data Attribute,
- Context References (16 with mask 48), pointing to a context appropriate Attribute or field associated with the program referenced Attribute (e.g. a State field associated with a Real valued VALUE Attribute).
- Idiomatic (32 with mask 48), pointing to a Setpoint SET Attribute used in control of the program referenced Attribute.
- External (48 with mask 48), pointing outside the program (or IA Block).

The pcode element thus contains the expression in a form supporting both fast computation and effective Back Compilation. It also contains the value field storing the final result, where it can then be accessed directly from a remote REFERENCE structure.

The Real Expression pcode is used anywhere some other statement form allows for calculated values. The pcode element for such a statement form will contain a REFERENCE structure, coded to point to the associated Real Expression pcode element. All Real Expression pcode elements will be located directly before the statement form which uses them, and in their order of use. This allows their predictable interpretation and Back Compilation.[48]

Real Assignments

The Real Assignment is the simplest example of a pcode structure making potential use of a Real Expression pcode element. It generally takes the usual language form:

D = A * 23 * (4 + C / sin(X + Y)) ,

and is represented by a Real Expression pcode element followed by the Assignment pcode element which can include a pointer to the value resulting from the computation of the associated Real Expression and a REFERENCE structure pointing to the variable to be set by the assignment (reversed from their order in the above statement).

In ICL, assignments can be used to set more than one variable to the same value by replacing the single assigned variable by a parenthesized list of variables (separated by commas):

(D, E, F) = A * 23 * (4 + C / sin(X + Y)) ,

In this case, the corresponding pcode element is made to include several REFERENCE structures, each set to point to a corresponding variable. This makes the Real Assignment pcode element a variable length element; the

[45] The Real Expression is also used to compute Time and Counts valued expressions. The Real Expression has its own pcode element because it may be used to represent Real values in any appropriate statement. Discrete Computations also support combined computations but in ways that require more specialized implementations. For this reason, all such computations are expressed in specialized structs each limited to use in particular Discrete pcode elements.

[46] As described below (the footnote asterisk), REFERENCE structures are designed to contain either a constant value, a direct reference to a remote value, or a reference to some further expression pcode element.

[47] Reverse Polish is a rearrangement of the expression, as entered by the user, so that all data precedes the operator operating on it; the above expression **A * 23 * (4 + C / sin(X + Y))** translates to **A 23 * 4 C X Y + sin / + δ *** , where the δ represents the dummy parentheses function described above. The internal representation then translates all resulting operators into codes greater than 191, and all variables and values into REFERENCE structures.

[48] The current design includes full language Compilers and Back Compilers as well as a simpler pcode editor/lister for directly manipulating pcode in a text script form. When working with pcode script, expressions are pushed as encountered on an expression stack. This makes their location available (by popping) when needed to configure their linking pointers for their using pcode element.

Attribute) [15],

- *Associated with control setpoints and alarms:*

RSET (representing the SET Attribute when used with a Real variable (VALUE Attribute)) [16],
SSET (representing the SET Attribute when used with a Discrete variable (STATE Attribute)) [17],
TSET (representing the SET Attribute when used with a Time variable (TIME Attribute)) [18],
CSET (representing the SET Attribute when used with a Counts variable (COUNTS Attribute)) [19],
HI [20], LO [21], DEV [22],

- RENAME pcode used to create locally applicable user names for Attributes [23],
- Prefix Control pcodes: LSED (Local State Environment Declaration) [24], STP (State Prefix) [25], NLP (Null Prefix, representing multiple sequenced statements on a line) [26], SCP (Scoping Prefix) [27]. The Prefix pcode elements each have a program counter pointer allowing their control of the sequenced execution of included pcode elements, represented multiple statements (separated by semicolons) on a same line.
- Activity Header pcodes: SEQUENCE [28], PARALLEL [29], CONTINUOUS [30], LOOPING [31], ST_DR (State Driven) [32]. These pcode elements all have an Operating State, to control operation; a program counter pointer, to control the basic (generally sequential) execution of their included pcode elements, and in some cases a second continuous program counter pointer to control the execution of any continued included pcode elements.
- Activity END pcode [33]. This pcode indicates programmed (early) termination to its including element. It includes a code indicating the number of levels of nested Activities to be affected.
- Filter/Compensations Idiom pcodes: LEADLAG [34], TOTLZER[??], DEADTIME [35], (Breakpoint Function) FUNCTION [36]. These will generally include an Operational State and any needed parameters. The passing of input and output data to a Filter/Compensations Idiom is carried out automatically by buffers, relating a variable Input and its normal data location.
- Idiom Control pcodes: IDLOOP (Idiom Loop; locates the beginning of a Loop Statement) [37], ENDLOOP (locates the end of the loop and manipulated variable) [38]. These elements contain a Loop Link pointer, in the first case linking the Loops together, in the second linking the end of the Loop to its associated Idiom Loop pcode element. The Idiom Loop element also contains an Operating State, controlling the Loop as a whole (including a Loop Local/Remote subState which controls remote computer control through the primary Setpoint); a Cascade State, defining the level of Idiom at which the Operator is currently operating manually; and dynamic data reflecting the current computed loop behavior. The Idiom Loop element contains a stack, with stack pointer, which controls the backward path computation of the Idioms in the Loop. Data is passed between successive Attributes and Idioms in Loop Statement through the I/O data buffers, the associated IDLOOP pcode element, and the stack.
- Idiom pcodes:

REGULATE [39], REGULATEX (with EXACT) [40],
FUNCTIONL (Loop Breakpoint Function applied to the output of a controller External Feedback Back Calculation) [41],
FUNCTIONM (Measurement Breakpoint Function applied identically to the Setpoint and Measurement Inputs of a controller) [42],
FEEDFWD [43], FEEDFWDX (with Feed Forward EXACT) [44],
HILIMIT [45], LOLIMIT [46], HICONSTR [47], LOCONSTR [48], BACKUP [49],
SPLR(Split Range) [50], BLEND [51], CC (Fuel Combustion Control) [52], MOC (Multiple Output Control) [53].

Idiom pcode elements will generally include a Loop Link pointer which links them to their associated Idiom Loop pcode element, an Operation State controlling the detailed behavior of the Idiom, and all of the basic control parameters. In addition to the basic pcode element, there will be, for each Idiom, a separate data structure which defines the information to be placed on the Loop stack in support of the backward path computation. This structure generally contains a pointer back to the main pcode element and any temporary data (not needed permanently, for later sample times) which is computed dynamically, but which must be passed between forward and backward path calculations.

- General Computational pcodes: REXPR (Real Expression) [54], RASSIGN (Real Assignment) [55], DASSIGN (Discrete Assignment) [56], TRTBSP (Truth Table State Prefix) [57], TRTB (Truth Table) [58]. These pcode elements contain information peculiar to the implementation of their particular calculation.
- HOLE pcode element [59], denoting a set of unused bytes in the program.
- Theme Statement pcode elements, for the following Theme Statements:

MARK [60], corresponding to a special debugging pcode as in **MARK POINT_ONE**, which logs the execution of the statement in the run time interpretation.
WAIT [61], which invokes a Wait for a given time (**WAIT 2 MIN**) or event (occurrence of the transition to a given State: **WAIT HI**)[44].
RAMP [62], Main Ramp Theme Statement pcode, with the additional associated:

- Ramp Theme Statement Phrase pcode elements:

RAMP2 [63], RAMP_FOR [64], RAMP_AT_FOR [65], RAMP_TO_IN [66], RAMP_FROM_TO_IN [67].

Main Theme Statement pcode elements will generally contain a Operating state, references to argument parameter variables or expressions, phrase program counter pointers and control clocks. The phrase pcode elements may contain secondary program counters and clocks, but the design intends that this kind of information be centralized in the Main element. In addition they will contain their own references to argument expressions or parameters, including any timer limits being clocked in the main element.

[44] Associated Wait States requiring a preceding Local State Environment Declaration, will keep that declaration active for the duration of the Wait testing.

interpretation.

Double colon State Prefixes, WAIT Statements, and State Driven require some variation of the above implementation because they require that the basic test setup be repeated every sample time until the condition has been met (or, as in the case of the State Driven Activity, indefinitely). One way or other, this will be supported by providing the LSED setup function (redundantly) as part of the associated pcode elements.

Activities, Compound Statements, and Theme Statements

State Prefixed Statements, Null Prefixed Compound Statements (Expressed by the user as several statements on a line separated by semicolons), Activities, and Theme Statements all represent compound multi-statement or multi-phrase forms with broadly unconstrained internal structure. They are implemented in terms of nested pcode elements. Their pattern of execution is supported by variants of a main interpreter, called recursively. These interpreters are each controlled by passed state variables, which allow them to controllably run, repeat, and abort, both under the control of the application program, the system, and the operator.

Meaning and pcode Translation of the Example Statements and Elements

As indicated earlier, the Small System Language is based about the basic SuperVariable scan. The application program is translated to a sequence of pcoded elements. These elements are each represented as a data structure instance with an initial one byte code (equal to a value between 0 and 67), which differentiates between the different applications functions that the pcode element might represent. For variable sized pcode elements, not having a constant length defined in a standard table, an integer pcode element length is included as well. The rest of the pcode element will consist of fields appropriate to the particular coded element.

The following subsections catalog the different kinds of pcode elements, while augmenting the detailed specification of some of the related language elements. The basic elements are the Attribute elements; they are included in the configured sequence in the order of application program definition (and run time execution). This treats the processing of Attributes as they are ordered in the Definitions Tables, from left to right and from top to bottom. The pcode representation incorporates both the Attribute type and value in each pcode element, thus supporting a certain heading redundancy. But, for user clarity, the Back Compiled listing groups consecutive variable definitions with identical heading patterns into tables for which the headings are shared. Correspondingly, their compilation, from consecutive data elements in a Definition entry, consists of recycling the header entries as the Attribute values are themselves compiled, to generate the pcode element type and value data.

The Back Compilation of Definition Tables and their entries requires recognizing any pattern of Attribute types starting with an initial NAME Attribute; continuing, to include Attributes other than NAME Attributes; and terminating just before a new initial NAME Attribute. It also requires counting the number of times that this pattern repeats itself consecutively. Such a pattern defines the shared set of headings for the corresponding repeated sequence of Attributes making up the next Table of Definitions, and the number of separate Definitions included in that Table. The remaining pcode element types in a program are also embedded, as much as possible, at their appropriate point of execution. Back Compilation of these other pcode elements, in turn, must also sort and reorder the elements into the natural listed order for user understanding, as described above.

Catalog of pcode Element Types

Extending earlier discussions, there are several broad classes of pcode element. Each pcode type name will be followed by its pcode code in brackets:

• Attributes: The pcode elements that declare SuperVariable Attributes and contain the associated data, of types:

- *Providing text variable names or descriptions:*

  NAME [1],

- *Associated with a particular type of data Value:*

  VALUE (Real Value) [2], STATE (Discrete) [3], TIME [4], COUNTS[5]

- *Associated with process I/O*[43]*:*

  IN (Real Valued) [6], OUT (Real) [7], DIN (Discrete) [8]; DOUT(Discrete) [9], FILT (including codes for several simple filters) [10],

  CONV(ERSION) (including codes for neutral, square root, square, log, exponential, and several kinds of thermocouple or other conversions) [11],

- *Associated with process Engineering Units and scaling of other representation translations:*

  MIN_MAX (representing the combination of MIN and MAX scaling Attributes) [12],

  RUNITS (representing the UNITS Attribute when used with a Real variable (VALUE Attribute)) [13],

  TUNITS (representing the UNITS Attribute when used with a Time variable (TIME Attribute)) [14],

  STATES (including a States Expression string which defines the user defined States for the most immediately preceding STATE

[43] The Process Input Attributes are ordered in the SuperVariable according to their normal execution scan: IN before FILT or CONV, before VALUE (or DIN before STATE). For ease of use, this same order is applied to Output Attributes are ordered; they are thus out of order for proper execution. The system uses the buffers described earlier to pass data between the successive Attribute processings. For Input I/O these buffers keep track of available Attributes and pass data between earlier and later stage Attributes in the scan. Thus, after data is input by the Input Attribute, the result is stored in the buffer meas_buf. Any filter or conversion Attribute operations can modify this value. When the Value Attribute is encountered this value can then be permanently stored. For Output I/O, code and pointer buffers are included to allow the execution to be deferred, supporting a reversed ordering. Thus as the OUT Attribute is encountered, its I/O address is buffered, and as a FILT or CONV address is encountered, its code is buffered. Then when the VALUE Attribute is encountered, the buffered I/O operations can be carried out. Similar considerations are necessary for the execution of Idiom Loop Statement output.

In this case, the LSED is supported by an Environment Buffer array of multi-bit (envbit) elements, each element corresponding to a distinct State Prefix Condition occurring in one or more of the LSE State Prefixes. The processing of the LSED snapshots the States of the associated variables and sets up a corresponding envbit element. Each State Prefix pcode element includes a subcode distinguishing its State Prefix Expression type, an index to the associated envbit element, a program state element used to control the execution of its conditioned statement or activity, and the nested pcode elements making up that conditioned statement or Activity. Because the State Prefix thus supports the nesting of Compound Statements, a Null Prefix pcode element has also been included in the design, based on the same structure but without any support for condition tests; it supports unconditioned Compound Statements as described later.

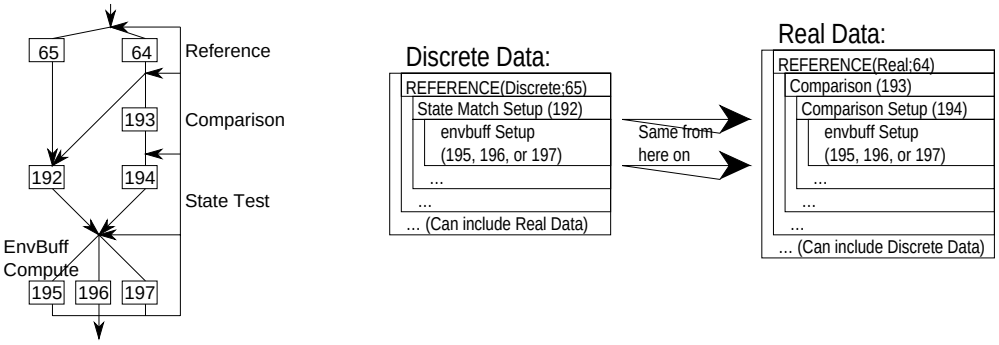
The Environment Buffer adds elements, stack-like, as it encounters new nested LSEDs, deleting those that are no longer in use. Its envbit elements support three kinds of interpretive action:

- Simple ORing action, in which all bits in the element are cleared or set. This is used to representing the collective effect of a number of State tests which collectively set the element if any one of them succeeds.
- Keyword action, in which one bit is set whenever a test fails, and another bit is set if the test succeeds. This is used to represent the collective effect of Keyword expressions when any/some/none of the distinct but identically named States may fail or succeed in matching the test condition.
- ANDed action, where several State names (each possibly corresponding to several identically named States) are to be tested, each bit corresponding to a distinct State name. This is used to represent the collective effect of the ANDed State conditions; a final State Prefix success requires the successful test of one of the tests for each of the State names, as represented by the setting of all bits.

Recalling that an LSED is made as a list of variable references, the LSED pcode element thus consists of a sequence of nested structs designed to carry out the above `envbit` element setup, for each execution. These structs are grouped to support the tests involving each variable taken one at a time. The REFERENCE struct representing each such variable is then followed by one or more structs that set up a particular masked State field representation, and compare the value to the target State value in the variable. Each such struct is followed, in turn, by one or more `envbit` setup structs which indicates the indexed envbit elements which should be set to respond to the result of the test. Real Data REFERENCE structs additionally require one or more comparison setup structs which provide the HI/LO/EQUAL comparison values, each with one or more state setup structs to set up the result of the comparison for the continued testing corresponding to the Discrete case. This is summarized in the table of structs below and the following figure:

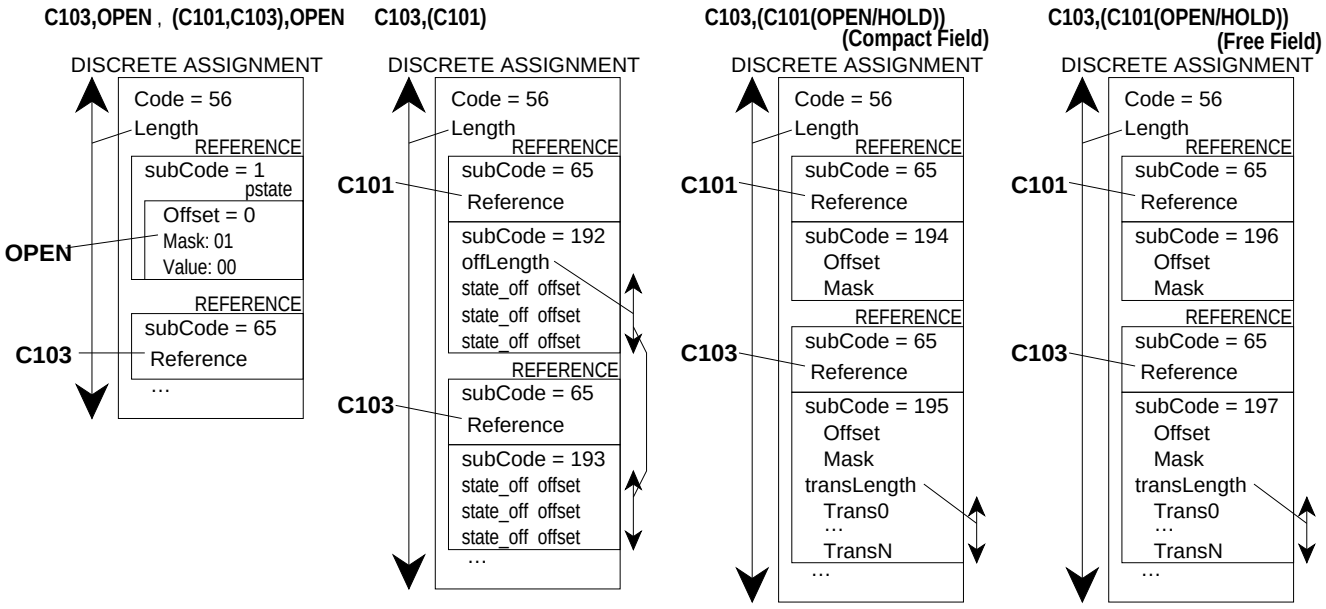
Code	Role	Entry Data	Run Time Function
• 192:	State Match Setup	Partial State Offset/Mask/Value	Compares Masked Value to State; returns Match state.
• 193:	Comparison Execution	Comparison Value or Reference	Executes Real Comparison and returns the HI/LO/EQUAL State, in the 1/2/4 bits respectively.
• 194:	Comparison Match Setup	Match Comparison State	Compares selected Match State Mask to Comparison result; returns Match state. Allows ORed combinations in the test.
• 195:	Simple envbuff Setup	envbuff index	Sets the indexed envbuff element to a full bit pattern, if the Match applies. Applicable to represent: - Single State State Prefixes, - OR State Prefixes (with several States separated by slashes; each such State requires a separate setup structure), - AND State Prefixes if all States apply to only one application variable (which can then be represented in a single Offset/Mask/Value State Match Setup.
• 196:	Keyword envbuff Setup	envbuff index	Sets the 0 bit of the indexed envbuff element if Match applies. Sets the 1 bit otherwise. The actual Keywords (ANY, SOME, ANY_NOT, ALL, NONE or NOT (or ELSE)) and their actions are supported in the associated State Prefix.
• 197:	AND envbuff Setup	Bit Pattern and envbuff index	Sets the bits corresponding to the included bit pattern in the indexed envbuff element if Match applies. Applicable to any other AND State Prefix.

The 64/65/192-197 numbers in the figure represent REFERENCE struct subcodes described earlier, or the table codes for the structs. The figure shows flow chart and data structures summarizing the above constrained structures.



Truth Tables are managed similarly to the LSE combined structure of LSED and State Prefixed Statements or Activities, with the Truth Table pcode element structured as an LSED, and State Prefix like entry elements, except that the Keyword based State Prefix Expressions are disallowed. The Truth Table entries are implemented as variant form State Prefix pcode elements in which each element has a separate `envbuff` element index for each variable in the Truth Table heading list. The interpretation of the Truth Table is analogous to the earlier LSED/State Prefix

and operator structs which can be compiled to support any variation of the statement and translation requirements.



A Discrete Assignment pcode element can consist of a free structured set of the following pieces (subject to the described constraints), consisting of REFERENCE struct and associated operator structs:

For Simple Assignments (Cannot be mixed with General or Mapped Assignment structs.):

- Discrete constant source States, stored (each in a REFERENCE struct, coded 1 above [as a Constant Discrete value]) as an offset (to locate a Discrete value element as offset from its normal reference element), a mask (to locate the intended data field within the value element), and a value (representing the changed value of the field).
- Sink REFERENCE structs (subCoded above as 65 [as a Discrete reference], and distinguished by not being followed directly by any other operator [subCoded 192-197] struct).

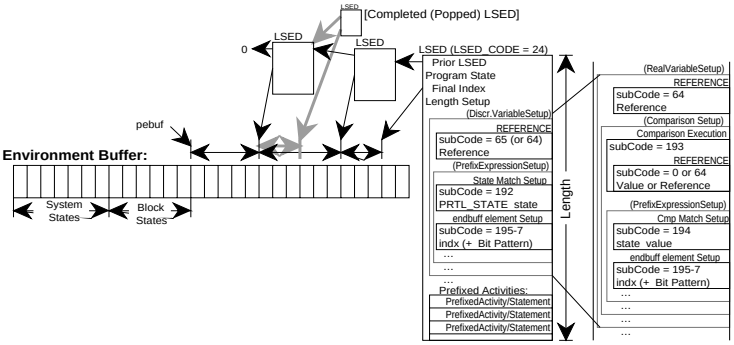
For General Assignments (Cannot be mixed with the Simple or Mapped structs):

- REFERENCE source structs configured as Discrete references (coded 65 above), to be later assigned, followed by a specially subcoded (192 above) list (with included length element) of offsets to represent all the (offset) Discrete values of the associated Variable.
- Sink REFERENCE structs (subCoded above as 65 [as Discrete references]), each directly followed by a struct (coded 193) listing the offsets of the Discrete sink values (equal in number to the source offsets).

For Mapped (Compact or Free)[42] Assignments (Cannot be mixed with the Simple or General structs):

- REFERENCE source structs configured as a Discrete reference, to be later assigned, followed by a specially subcoded (194 [Compact] or 196 [Free] above) Offset and Mask representing a field which may be translated for the assignment.
- Sink REFERENCE structs configured as a Discrete reference, to be later assigned, followed by a specially subcoded (195 [Compact] or 197 [Free] above) Offset and Mask representing the sink field to which the translated data will go. The struct includes a Translation table (with length) which acts as a look up array, to make the translation.

The Local State Environment Declaration and its associated State Prefix are also based on a similar structure of nested structs:



[42] The implementation distinguishes between internal packed field State representations whose mask bits are contiguous (Compact) and those (whose mapping is more time consuming) whose mask bits may be separated (Free). The Compilers distinguishing between these, choosing the most efficient pcoding, in pursuit of efficiency equal to conventional methods.

State/Field Translations

The translation from States to fields, packed into some defined masked subset of allowed bits[40] in the target integer, is applied as a two part repetition, repeated for each State in the State Expression while scanning the States Expression for that State and mapping it into a mask and value:

- Looking for the named State in each succeeding set of alternative States (and fields), and mapping that name to a value to be ORed into the intended field, by the following steps:
  - Counting the number of alternative States up to finding the intended State.
  - If a match is found, mapping each consecutive (from right to left) bit value (0 or 1) of the resulting count integer, into a consecutive (from right to left) allowed bit, on a bit position by bit position basis, to form the mapped value.
  - On completion of the computation of the field mask below, masking out the field, and ORing the value bits into the field to replace the preceding value.
- Computing a mask defining each field as it is encountered in the States Expression for the particular State search, according to the following steps:
  - Counting the number of alternative States for each set of alternatives (each packed field).
  - Mapping a 1 bit for every bit position in the resulting counts integer, up to the left most set bit position, into the right most uncommitted allowed bit position to create a field mask (defining the field).
  - During the search for each State in the States expression, a temporary mask of dis-allowed bits is developed additionally excluding the bits of each completely searched field from use in the search of succeeding fields (sets of alternative States in the States Expression). The mask is initialized at the beginning of each State name search to include just the defined dis-allowed bits. For each field search, the mask is translated to a mask of allowed bits.
  - If no match to the searched State name is encountered for the particular field, the temporary mask is extended by ORing the field mask (computed in the earlier step) into it.
  - If the searched State name is encountered for the particular field, the bits of the corresponding field mask are permanently added to the set of dis-allowed bits (or removed from the set of allowed bits).[41]

Minor modifications of this basic pattern (and the following one) allow the accommodation of the various alternate field patterns discussed previously. The implementation also keeps track, in a separate State Expression, of any States never encountered in the States Expression (for error notification or search in a separate Attribute element). The similar translation, from a masked subset of field bits and a States Expression declaration to a State Expression, is carried out by the repetition, for each State (and field or set of alternative States) in the States Expression, of the following steps, starting before the first States Expression State name:

- Scan to the next States Expression State name and count it.
- Scan the (temporary) mask of allowed bits, and:
  - Map, from right to left, for every set bit and its corresponding position, the (0 or 1) value of the correspondingly ordered bit from the current count value. The result is the mapped current count value.
  - Keep track of the left most encountered allowed bit position, for which a count bit is set.
- If the total packed value being mapped, masked by the allowed bits up to the left most encountered allowed bit position, matches the above mapped current count value, then the corresponding State value applies; it is added to the output State Expression.
- Mask out all bits, covered by the field, from the allowed bits.

Discrete Computations

The programmed computations include the following:

- Discrete Assignments.
- State Prefixes.
- Local State Environment Declarations, supporting the State Prefixes with a variant to support Truth Tables.
- Truth Table and Truth Table Entries

A basic principal in these computations is that States of different variables are the same, for purposes of control computation, if they have the same State names, whether they correspond to the same packed field value or not. This imposes a need to distinguish between identical packed formats and different ones, and make the appropriate translations as needed. For reasons of speed and efficiency, all this is handled, as much as possible, in the compilation of the original program source text. The pcode implementation, as designed, is capable of complete generality in handling all of these issues, but the above specifications have initially included constraints, to simplify the implementation and application safety.

Each of the above forms corresponds to a single pcode element. But these elements are designed to include substructs which function like the operators later used in the Real Expression pcode element. The Discrete Assignment, and LSED (including the Truth Table variant) each include an open ended set of REFERENCE structs

[40] Currently expressed as a mask of disallowed bits, converted to the temporary masks of disallowed and allowed (described later) bits by inverting each bit value.

[41] The final version of the mask of disallowed bits is returned to the initiating calculation to define the set of bits disallowed from any later State mapping.

ReName Attribute (which in turn contains the character string for the alternate name). The same code element in a ReName Attribute defines the renamed Attribute Type code.

- A `supvar_chain` pointer element, which can be set to point to the next Attribute pcode element, somewhat speeding Zipper Reference and other system searches.

The system also includes an `an_table` (Attribute Name Table) directory to facilitate binary searches of Attribute and Type names and a `cntxt_table` array which codes the remaining Attribute application data and structure. This table is used to control the pcode compilation of Context related reference.

REFERENCE struct.

All referenced constants or variables, or expressions taking their place, in any Small System ICL program are translated into an internal REFERENCE struct. For constants, the value itself is stored in the struct; for variables or expressions, a pointer to the intended value is stored in the struct. The struct is coded for data type, for constant/Reference/Expression, and, if a Reference, for Direct/Context/Idiomatic/External reference type. This code is designed so that all net code values in excess of 191 cannot represent REFERENCE structs. For Real (,Time, or Count) data, this code thus allows REFERENCE structs to be alternated recognizably with operator codes (coded to values greater than 191). For Discrete data, a similar, distinctly coded (>191) struct defines various kinds of Discrete operator subpcode elements. The REFERENCE struct supports pre- and post- compilation formats, allowing the simplified entry of variable and State references before the internal data base is complete enough to establish their final locations.

The REFERENCE struct also includes an opstate displacement code which helps to locate an associated operations state (using the OPSTATE and DSPLT (Displacement) macros). For Real, Time, or Counts data, any REFERENCE struct reference points to the actual referenced data; Back Compilation must recover the Attribute address itself. Discrete data involves some special complications, because the actual data may be located in complex relation to the actual referenced value, even in a different Attribute, or in several Attributes. The actual data is selected by an associated data value offset.

Buffers and Stacks Coordinating Attribute/Idiom Computations

Certain Attributes and Idioms make implicit use of data computed by each other, within the basic interpretive scan. This includes the passing of I/O data to and from the basic Value or State Attribute value element, scaling of a process value, and normal control or control manipulation of a variable. In the Small System this communication is supported by a collection of standard buffers. The existing implementation includes the following buffer variables:

Name	Purpose
• `meas_buf`	Store the most recent Real I/O input Measurement value.
• `value_buf`	Stores a pointer to the (previous) Value Attribute value.
• `out_buf`	Stores a pointer to the Output Attribute value.
• `set_buf`	Stores a pointer to the current Setpoint Attribute value.
• `min_buf`	Stores the current minimum scaled value.
• `max_buf`	Stores the current maximum scaled value.
• `io_code`	Stores the current I/O flag state.
• `pio_code`	Stores a pointer to the I/O state value in the current I/O Attribute.

Idioms generally use unscaled data internally (representing data in terms of percent of range) and scaled data externally, for human interfacing. This allows control data to be indicated and trended in natural, percent of scale, analog or control debugging terms, leaving the engineering units display for more application related uses. The buffering is arranged to support this need. Idioms also frequently are supported by one part of their function which is to be carried out (relative to the control variables and other Idioms) ordered from measurement to valve, and a second part which is to be ordered from valve to measurement. The Idiom Loop pcode element (described later) which initiates each Loop, includes a stack which serves to manage the stacking and dual execution ordering of the associated function calls.

Attribute Renaming

When an Attribute heading is renamed, its first associated Attribute pcode element is directly preceded by a ReName pcode element, containing the alternate name and a `renm_attribute` code indicating the renamed Attribute Type. All affected Attributes (from the same table column) will include an index coding the ReName pcode element as their `renm_attribute` element value.

Discrete Translations and Computations

The design goal is to simultaneously support the naturalness of the named States, defined in the State and States Expressions, and the efficiency of conventional Discrete Computation and packed data. The implementation assumes that all computations will be carried out with respect to a packed data based representation, requiring the actual States and the translation between the packed data and the named States only when communicating with people.

- Associated Theme Statement Footnotes, embedded at their point of execution. These allow the associated computations to be coordinated with relevant events of the Theme Statement.

Theme Statement pcodes are executed by a more specialized version of the recursive run time interpretation described below for Activities. As indicated earlier, the Theme Statement area before Definitions can include simple Activities and other statements which may occur as Footnotes. Those Footnote statements will be executed continuously as if they were normal Footnotes even though they will not be listed in the Footnotes Paragraph, even as all other forms will be executed once according to their basic execution rule.

- Definitions; made up of a continuous list of the pcoded SuperVariable Attributes, listed as ordered from right to left and from top to bottom in the users program listing. Inserted among the definition Attribute pcodes will be associated pcode elements, whose execution is carried out within the scan of the Attribute pcodes:
  - Footnote Statements, Expressions, and Activity Brackets (with or without Prefixes), embedded at their point of reference and execution. These can represent general computing statements, and, infrequently, Theme Statements or Activities (but not Idioms or Loop Statements (or embedded Attributes)). The configurer will generate the pcode from the configuration listings and merge them with the Attribute pcodes in the appropriate position. In Back Compilation, the appropriate data will be sorted and separately displayed. Individual general computing statements are executed continuously in the Attribute Footnote context, but Activities (and Theme Statements) are expressed and executed normally to their natural continuation or termination, as indicated below.
  - Filter/Compensation Idioms, embedded at their point of execution. The configurators and Back Compilers will maintain their separate listed position.
  - Idiom Loop Statements, broken by the compiler into the individual Idiom pcode elements which are then embedded at their point of execution, after the Set Attribute of the associated Real Value Attribute.

  Note that the definition Attributes for consecutive definitions occur consecutively without any other separator. The Back Compiler imposes an almost artificial grouping into definitions based on initial Names and repeated Heading patterns. For run time execution purposes, the several definitions amount to a single string of pcoded elements. The listed tables are artifacts of the original user entry and the later Back Compilation only.
- Procedures, positioned after the main computations where they can draw on the accumulated I/O and control data. They consist of:
  - Statements, with or without Prefixes, which here are executed to completion and terminated upon initial change to the appropriate State.[39]
  - Activities, with or without Prefixes, as described above, executed to their normal continuation or termination. This is the normal position for Activities. An Activity pcode is directly followed by its included, nested statements, statement support expressions, and any recursively included nested Activities. The whole combination is seen as a single pcode element from the outside; its included length structure field is computed to include all of the nested pcode elements. It is executed by making a recursive call to the same run time interpreter which interprets the pcode sequence as a whole, but applied to the sequence of internal nested pcode elements. When first executed, the Activity (or Theme Statement) is initialized. Thereafter, each sample time execution continues its natural progression, until it is terminated. Thereafter the sampled scan will encounter and bypass the Activity, unless it is re-initialized by some means.

Basic Implementation Structures and Strategies

The following discussion summarizes elements from the "Introduction and Data Structuring Practices for the ICL IA Block" document. Its purpose is to provide some bridge between the specification, the design concept, and the more detailed design documents.

Relation between SuperVariable pcode Element Order and Displayed Listing

As indicated before, the complete set of SuperVariable definitions is compiled into a single sequence of pcode elements, which may also have individual interspersed Idiom or Footnote pcode elements, as well. Each element in this sequence has its own (possibly redundant) type code and value element. Thus the compiler must repeatedly apply the type headings to the successive entry elements in each definition group. The Back Compiler must first recognize any repeated pattern of Attributes, which could be converted to a sequence of entries all grouped under the same set of table headings. It must then display a line of these headings, and then the successive lines of Attribute value data (without redundant headings). The Back Compilation will always form the largest possible such group even when the original source tables had been split into a succession of similarly headed groups.

The Back Compiler must also sort the individual Idioms into Loops using the chained information defined later, and the Footnote Statements, all displayed in their appropriate Paragraphs.

Attribute and Zipper Reference Support

The Attribute pcode elements share a basic initial structure, including:

- An initial `code` element, which distinguishes between the distinct Attributes and pcode elements
- An optional `length` element, which defines the actual length of pcode elements of uncertain length.
- An `attribute_chain` pointer element, which allows the element to be linked on a chain of similar Attributes from an `attr_drctry` array entry corresponding to the `code` value.
- A `renm_attribute` code element, whose non-zero value, for any renamed Attribute index, codes the corresponding

[39] The Attribute Footnote statements described earlier may be executed repeatedly. Other Footnote statements execute continuously or to termination depending on the context, and their own internal structure.

An ICL application program, like the one illustrated, is translated into a sequence of pcode structures. While implementation is not usually thought of as part of a language specification, the Small System Language is motivated by implementation opportunities which make the implementation discussion meaningful in a number of respects. The pcode structure is designed to be executed in real time, under an interpreter. This real time execution reflects the real meaning of the program. The pcode has also been designed to allow a reconstruction of the original application by Back Compilation.

Preliminary Definition of pcode Terms

The following pcode related definitions establish a base for discussions, beyond that already introduced in the previous example. Some of these concepts will be redefined in greater detail as the discussion continues:

- pcode and pcode structure. A pcode is an artificial "machine language" designed to represent the basic functions of a higher level language in a form that is easily translated into (from the higher level language) and at the same time easily translated itself for run time execution (or Back Compilation). The corresponding ICL Small System pcode structure is a data structure which contains an initial `opcode` field, assigned a number between 1 and 67, distinguishing the type of the particular pcode structure. Depending on the type, the `opcode` may be followed by a field defining the pcode element length (if the pcode element has variable length), and fields standardized for all structures of similar type. These standard fields will be followed by further fields including operations States, values, and parameters associated with the representation and function of the particular pcode structure type and corresponding function. Since the Small System language is based in the SuperVariable scan, a pcode will, in the simplest case, correspond to a SuperVariable Attribute.
- Attribute pcodes; those structures representing SuperVariable Attributes, including the type and operating data.
- Statement pcodes; those structures which represent a simple statement in the language.
- General Computational Statements; those statements which support a greater language flexibility by expressing standard (mathematical) computations not addressed by the special application related statements. These will normally correspond to one main pcode element.
- Theme Statements; most completely illustrated in the Ramp Statement, but designed to reflect more open ended, complex but standardized sequencing and coordination activities, as reflected in such traditional pieces of equipment as cut cam follower units or drum sequencers. They are designed to accommodate their own Footnote support statements. In pcoded terms, these will be translated to a main pcode element and associated subordinate phrase pcode elements.
- Expression pcodes; those structures designed to support a standardized treatment of computational expressions that can naturally occur within a number of kinds of statement. The expression pcode is designed to be used with some other pcoded statement to carry out that expression’s computation as it is embedded in the programmed statement, and pass the resulting data to the associated statement pcode element. The example Footnotes are all General Computational Statements (e.g. **T104=(T101+T102)/2.0**) translated to Statement pcodes with supporting Expressions (e.g. **(T101+T102)/2.0**) and associated pcode elements.
- Activity pcodes; more generalized pcode structures (than the Theme Statement pcodes, but similar in their usages), designed to express the Activity language form with its embedded statements and Activities, in a recursive embedded pcode structure.
- Footnote; a general computing statement or Activity set aside from its position of program execution and referenced from that location by a numbered superscript (which may include an asterisk or dagger).
- Filter/Compensations Footnote; like a general Footnote except that it applies only to Footnoted Attributes and expresses the application of dynamic or nonlinear compensation to some variable. Its action is declared by single word Filter/Compensations Idioms.

Pcode Paragraph Arrangement

This subsection outlines the basic arrangement of pcoded elements in a compiled application program. As indicated earlier, the program starts with a Header Paragraph containing a Name, and a list of user defined program States. Thus the first three pcodes in the translated pcode sequence will be standardized and fixed as a special case, to consist of:

- A Name Attribute pcode, containing the name of the Block or Small System controller.
- A State Attribute pcode, containing the current program global State value, divided in three State value categories:
  - Standard (built in) operations States (e.g. UNDEFINED/DEFINED/DEAD) associated with the Attribute, included in the opstate field, built in but only defined for this initial overall program/block State Attribute.
  - Special case operations States, defining the system program State (RUN/SUSPEND/ABORT), included in the opstate field, built in but only defined for this first occurrence of a State Attribute,
  - User Defined States, included in the usstate field, whose interpretation is defined by the STATES Attribute below.
- A States Attribute pcode, containing a States Expression which defines the user defined States (their names and relation to each other) associated with the above User Defined usstate States.

The configured order of the remaining Paragraph pcodes generally follows their scan execution order, which is distinct from the order given in the program listing (Definitions, Footnotes, Filters/Compensations, Loops, Theme Statements, and Procedures). The intended translated pcode order is:

- Theme Statement pcode elements, positioned at the start of the sequence where they can set the setpoints of the following control loops. Each Theme Statement will be translated into a single initial key pcode element and any number of subordinate Phrase pcode elements associated with the particular type of Theme Statement. These pcode elements also include (with the special Theme Statement pcodes) associated general pcode elements:
  - Preceding expression pcodes, used to express statement parameters, when computed from component variables.

5. This Bracketed Local State Environment Declaration represents a special shorthand combining declaration and Prefix State, and testing whether the flow setpoint is less than the prescribed 6000 barrels per hour (BPH). The **END** statement diamonds are not in this case numbered; this indicates a normal termination of the Activities and procedure.
6. This is the normal Local State Environment Declaration Expression, on its own line as an independent statement, with included items (**M200**, **P200**, **V200**) separated by semicolons, and separated from the following line and State Prefixed statement by a semicolon. All related consecutive State Prefix statements testing the States of the Declaration, are separated by semicolons, whether occurring on the same line or not.
7. This is the corresponding normal State Prefixed statement. In this case, if any of the associated variables is associated with a Loop or Idiom in manual, the whole procedure is aborted.
8. This the normal Real assignment, applicable to Reals, and to Count and Time variables.
9. This statement sets the **CONTROL_BLEND** State, allowing the execution of associated State Prefixed Loop Statements.
10. This is a nested single statement Continuous Activity (indicated by the thick continuous Activity Bracket), indicating a continuously executed shorthand comparison Local State Environment Declaration and State Prefixed statement. The test is carried out continuously and repeatedly until it succeeds. In this case, the Activity causes the suspension of further processing until the measured outlet flow exceeds the targeted 2000 BPH value. The **END** command and paired asterisks indicate a normal termination of an internally nested Activity.
11. This statement sets the **CONTROL_BOOSTER_PUMP_MOTOR** State, allowing the execution of an associated State Prefixed Loop Statement.
12. Two more Discrete assignments combined on a line separated by a semicolon. At this point they advance the count[38], and restart the timing.
13. This statement sets the **CONTROL_RAMP_PRESSURE** State, allowing the execution of an associated State Prefixed Loop Statements.
14. The following Continuous Activity tightens the integral setting if the loop is too slow to respond. However the **CONTROL_RAMP_PRESSURE** State controls an Idiom Loop Statement which includes a Hi Rate Limit Idiom. The change in setting is meaningful only after the Hi Rate Limit will naturally have ramped the loop to its final position. This Wait statement puts off the loop tightening until this is no longer an issue.
15. The temporary variable **ABSERR** in the Continuous Activity is continuously computed as the absolute value of the difference between **P200** and its setpoint, the absolute error in **P200**. The present statement sets a **LAG_FILTER** State, permitting the execution of an associated State Prefixed Activity, which filters the computed **ABSERR**.
16. The filtered **ABSERR**, stored back in **ABSERR**, can be tested. If the absolute error exceeds 3 PSIG the P200 Integral setting is changed to **INT2** from the earlier **INT1**, and the Continuous Activity is terminated. The enclosing Parallel Activity, with its initial Loop statement, continues executing, continuing the control of the ramped pressure.
17. This line includes the Shutdown State Prefix associated with the final nested Sequential Activity.

Other Theme Statements

Theme Statements have been introduced through the illustrative RAMP Theme Statement as most useful in the Small System language as the coordinating framework for small applications not bound together by Large System ICL Operations. In addition to the RAMP Statement, used for representing a basic production time based production profile in small batch units such as dye becks and canning retorts, the design envisions several other Theme Statements: WAIT (for simple delay timing and event waiting), BLEND (for coordinating Batch Blends), SEQUENCING/TIMING (to take the place of drum sequencers and timers), STREAM (to coordinate conveyor belt production), and PROFILE (to coordinate the operational control of related loops).

WAIT Statement

Unlike other Theme Statements, the WAIT Statement can be used anywhere a normal computing statement can be used in a sequenced list of statements, to suspend execution for some time or condition. The form consists of the WAIT keyword followed by a time value (constant followed by time units or Time variable) or by a State Prefix:

WAIT 2 MIN or **WAIT HOT** or **WAIT ALL HOT**

A WAIT can be viewed as a conditional statement defining the condition under which the following statements will be executed. Under this interpretation (particularly when there are following statements on the same line [with separating semicolons]), the WAIT statement has much the same effect as a double colon State Prefix. Thus:

WAIT ALL HOT; C103, OPEN is the same as **ALL AUTO:: C103, OPEN**.

Pcode Paragraph Arrangement

[37] The Message statement assumes a single console target position. Ideally a more sophisticated strategy would be pursued in which the user could define several distinct display locations. These would be selected with a special Prefix: ON CONSOLE: "Display Message".

[38] By one; the COUNT variable is restarted elsewhere to 0, and compared to its setpoint of 1; It becomes high of that value in the sense of the earlier State Prefix if it exceeds that value [=2].

can also be used with an included number to indicate the distinction between a normal and an abnormal exit condition. State Prefixed statements immediately following the Activity can be conditioned on the corresponding numbered States. In this case, the zero State in a Prefix will correspond to a normal exit.

The example below comes from a Gas Blend Booster Pump application. It assumes continuous control of blended flows (the **CONTROL_BLEND** Procedure) each flow supported by its individual pump, but with the total blended flow supported by a booster pump. The point of the application program shown is to activate the booster pump only when the anticipated or measured total flow exceeds certain limits (F200>2000 or F200.SET>6000, below). The Booster Pump can only have two starts within any hour. The example makes apparent lower level Procedure calls to **CONTROL_BLEND**, **CONTROL_BOOSTER_PUMP_MOTOR**, **CONTROL_RAMP_PRESSURE**, **LAG_FILTER** Procedures. Within the limitations of the Small System Language this is implemented using Block States defined by the States Expression **_/CONTROL_BLEND, _/CONTROL_BOOSTER_PUMP_MOTOR, _/CONTROL_RAMP_PRESSURE, _/LAG_FILTER, _/START/SHUTDOWN,** and associated State Prefixed control Idiom Loop Statements, all States initially Nulled.

```
                                                          Numbered Notes
START; COUNT, RESTART; TIME, RESET                                    1
START:                                                                2
  [TIME] TIME/LATE :      COUNT, RESTART; TIME, RESET                 3
◊ [COUNT] HIGH :          "Terminated; Too Frequent Starts"; END◊     4
◊ [F200.SET < 6000] :     "Finished; Booster Not Needed"; END◊        5
  [M200; P200; V200];                                                 6
◊   MANUAL/LOOP_MAN :"Terminated; Loops in Manual"; END◊             7
  V200 = 5 %                                                          8
  CONTROL_BLEND                                                       9
  * [F200 > 2000] :       END*                                       10
  CONTROL_BOOSTER_PUMP_MOTOR                                         11
  COUNT, ADVANCE;     TIME, RESTART                                  12
  M200, ON
    CONTROL_RAMP_PRESSURE                                            13
    P200. INT = INT1
      WAIT 5 MIN                                                     14
      ABSERR = ABS(P200.SET- P200)
      LAG_FILTER                                                     15
    * [ABSERR > 3] :    P200. INT = INT2; END*                       16
SHUTDOWN:                                                            17
  V200 = 100 %
  M200, OFF

                    Legend
  Sequential      Parallel
  Looping         Continuous
                  State Driven
```

The Noted statements are explained below.

1. The control program consists of a State Driven Activity. Such Activities include State conditioned Statements or Activities (preceded by State Prefixes as explained previously), and simple unconditional statements or Activities (without such Prefixes). In the State Driven Activity, the State Prefixed statements or Activities are executed whenever the system enters the corresponding State, and aborted whenever the system enters some other State. All other statements are executed once when the Activity is first called, if no State (i.e. the Null State) applies. The first statement in the line noted is an abbreviated form of State Assignment. It sets the Block to the indicated Start State. This will be the normal initial statement form in a State Driven Activity: a single word statement defining the initial State of the Operation during the processing of that Activity.
   The State Prefixes in different statements in a State Driven Activity, are intended to be chosen from a set of mutually exclusive States, each used in just one Prefix. However, the system will not discipline this constraint. Each State Prefixed statement or Activity is activated or aborted on its own terms. This line also includes two other statements which initialize the States of a Counts and Time variable.
2. This is the State Prefix for the following (Sequential) Activity. Given the State Driven Activity initialized in the Start State, the Prefix indicates that its associated Activity is to be executed as long as the Operation remains in that State.
3. This is the initial statement in a nested Sequential Activity. It shows the shorthand use of a Local State Environment Declaration. Its listed object, the **TIME** variable, has the internal States defining the state of timing of the variable.[35] If one of those States matches the Prefix (The expression **TIME/LATE** calls for a match of either **TIME** or **LATE** States), the remaining statement is executed. In this case, there are two following statements to be executed, separated by semicolons, re-initializing the Counts and Time variable States.
4. This line is similar to the last except for the single named State in the Prefix and the executing statements: The pair of numbered diamonds, associated with the **END** statement, indicates an aborting termination of both nested Activities. The Activity terminates if two starts have already occurred in the last hour.[36] Before the **END** statement a message is sent to the operator as a text string in quotes.[37]

[35] These timing States are set based on the comparison to a one hour Setpoint.

[36] The counting States of the **COUNT** variable are set based on the comparison to a one counts Setpoint.

colon corresponds to the State Prefix colon punctuation. The entries take a similar form to the State Prefixed Statement[31] except that each Prefix contains several separate State Condition Expressions, one for each of the Declaration header variables, listed in the same order as those variables, and each containing only States relevant to the particular variable. As with the State Prefixes, these State Condition Expressions may contain separating slashes or commas but not both. The State Condition Expressions are separated by spaces or tabs; more than one space or tab sets off an empty (don't care) State Condition Expression, meaning that the corresponding variable can be in any state without making the entry unacceptable for execution.

Activities[32]; Semicolon Separated Statements

ICL can group statements into executing lists in two ways: by use of semicolons, and in Activities (both illustrated earlier). Statements separated by semicolons can be grouped on the same line or on consecutive lines.[33] 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.[34]

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. If any of these line executions have suspended statements included, their continuation will proceed independently of related lines, as in the case of Parallel Activities. The underlying purpose of the semicolon is to group statements subject to some related data environment. Usually this environment is related to the set of States that affect their conditional execution.

As indicated earlier, Activities are Bracketed groups of statements (as illustrated below with legend), principally applicable to the Procedures Paragraph. There are five kinds of Activities, distinguished in their execution order:

- Sequential, executed sequentially in real time, as a whole, where actions are carried out at particular times, or for particular intervals of time, with waits for particular events before other actions are to be taken up.
- Looping, where an activity, sequenced in real time, as a whole, is repeated after each completion, some number of times, or for some duration (or "forever").
- Continuous, where each member of a set of calculations or controls is executed, over and over, independently of the others, once each sample time, simulating the behavior of the traditional analog or relay control system.
- In parallel, where several sequenced activities are all carried out at the same time independently, each only once to completion and subject to its own internal timing and event tracking, no one blocking or affecting the other.
- Discrete State Driven, where, at any time, a single one of several independent activities is chosen for execution, depending on the discrete state of some variable, operator control, or process element. The conditioning States will normally be system States, but a Local State Environment Declaration may be used to select some other set of States to condition the Activity. If no such State is active, the State Driven Activity is considered to be in the NULL State; in this case, the non State Prefixed statements or Activities will be executed (once initially or on the initiation of this condition).

As indicated earlier, Activities are initiated, on configuration entry, by a left parenthesis occurring unclosed on the line preceding the start of the Activity, directly followed by a carriage return (e.g.: **(**© or **START:(**©), and terminated by a matching final unopened right parenthesis with single ©. The choice of Activity Type is controlled by using repeated ©'s to cycle through the displayed Activity Type, before continuing with the next statement.

The example below also shows the use of **END** statements with their marking asterisks or diamonds. **END** statements cause the immediate termination of an Activity, without regard to the normal termination condition of the Activity Type. The paired asterisks or diamonds enclosing an **END** statement include the Brackets for all Activities that are to be terminated at that time; they control the level of termination for the statement. On configuration entry, multiple ©'s after the **END** statement are used to control the position of the left hand asterisk of diamond. For example, **END**© causes the innermost Activity to be terminated, **END**©© causes the innermost two Activities to be terminated, etc. For each of these uses of multiple ©'s, the choices cycle back to the original case once all alternatives have been exhausted.

The use of asterisks and diamonds is similar except that diamonds are used for emphasis whenever the outermost activity is terminated. The processing of multiple ©'s on entry automatically includes the choice of symbol. Diamonds

[31] The colon punctuation takes a similar role here.

[32] Note that an Activity can have a preceding Prefix just as a Statement.

[33] 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.

[34] 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.

State Condition Usage (Simultaneous States) **AUTO, REMOTE: C103, OPEN** A set of mutual conditioned States, which, if each is seen in any one of the Block (or Local State Environment Declaration variable) States, will cause the execution of the rest of the statement.

Keyword Usage **ALL AUTO: C103, OPEN** A State (as named), preceded by a Keyword, which defines some combination of the States of the Block (or Local State Environment Declaration variable) which, if seen, will cause the execution of the rest of the statement. The relevant Keywords are:

- **ANY**; same as Single State.
- **ANY_NOT**; The rest of the statement executes, if at least one of the relevant States in the Block or set of variables is not active.
- **SOME**; The rest of the statement executes, if one or more but not all of the relevant States in the Block or set of variables applies.
- **ALL**; The rest of the statement executes, if all of the relevant States in the Block or set of variables apply.
- **NONE**; The rest of the statement executes, if none of the relevant States in the Block or set of variables apply.
- **ELSE**; The rest of the statement executes, if all previous consecutive semicoloned State Prefix conditioned statements failed and the rest of the Prefix (which may include any of the above forms including the Keywords) succeeds.

If a State Prefix is to be affected by a Local State Environment Declaration on the preceding line, the Declaration must be terminated by a semicolon. If several Prefixes, on consecutive lines, are to be affected by the same Declaration, they must each (except the last) be terminated (before their normal carriage return) by semicolons.

The normal State Prefix (with or without a preceding LSED) is executed once, continuing if it succeeds and directly terminating if it fails. There is one other simple conditional situations needing support: a set of statements which are to be executed whenever the indicated condition finally occurs. This variation is indicated by replacing the single colon by a double colon: **ALL AUTO:: C103, OPEN**. Closely related to this kind of expression is a statement reflecting a need to wait for some condition. In ICL this is stated in a WAIT command as described later. Finally there is a set of alternative computations which are to be alternatively switched on and off execution depending on which States are currently applicable. This situation is described by the State Driven Activity described below. All of these forms involve related implementation issues.

Shorthand Declaration/Prefix Combinations

A Local State Environment Declaration can be combined with a single State Prefix on a single line, without semicolon, to carry out a single independent test:

[TIME] TIME/LATE: COUNT, RESTART; TIME, RESET

A modified Declaration can be used combining any number of legitimate **< > <= >=** comparison operators, each operator between two variable names, and representing the simultaneous occurrence of each of the comparison conditions:

[F100.SET > 6000]: V100, OFF	, which is equivalent to:
[F100.SET, 6000]; HI: V100,OFF	, or
[2000 > F100.SET > 6000]: V100, OFF	, which is equivalent to:
[F100.SET < 2000]: [F100.SET > 6000]: V100,OFF	, or to:
[F100.SET, 2000]; LO: [F100.SET, 6000]; HI: V100,OFF	.

[Compilation of any of these shorthand combinations produce the pcoded equivalent of its longhand equivalent. Back Compilation of any combination of Declaration and Prefix is assumed to develop the shortest and simplest form, using the shorthand forms if they result in single statements, and the standard form if the result is a single leveled set of multiple prefixed statements.]

Truth Tables

The Local State Environment Declaration can be combined with Discrete computations in another more general way, defining a Truth Table. If the Declaration is preceded by the Keyword **TT**, it defines the heading for a Truth Table. Following lines separated from each other by semicolons must take the form of Truth Table entries:

```
TT[ C101; C102; C103 ] : ACTION;
    OFF   OFF   OFF    : L100, OFF;
    OFF   ON    ON     : L100, ON; L101, ON;
    ON          ON     : F103.SET=50
```

The general ICL Truth Table differs from the conventional Truth Table only in allowing "Don't Cares" (Blank States) in the Table and arbitrary ordering of the entries. It is interpreted to call for the first (top most) entry whose States are satisfied, and can be interpreted as calling for no action when no entry is satisfied. For the Small System Language, the result of the Table is not expressed as a result variable but as an action statement, carried out whenever the particular entry is the top most entry whose included State Condition Expression column entries are each consistent with the current State of the corresponding Declaration variables. The Action Keyword heading and colon punctuation are unnecessary on configuration input; they can be created automatically in Back Compilation. The

being tested are States of the Block taken as a whole. But State Prefixes can also be based on States of individual variables. In this case these States must be included in an appropriate Local State Environment (LSE). When the States to be tested in a State Prefix do not correspond to States of the Block as a whole, the variables or other ICL elements whose States are to be tested are declared in a Local State Environment Declaration (LSED). This consists of a pair of square brackets containing the variable or element names separated by semicolons:

```
[M200; P200; V200];
ON,READY:    "MOTOR ON";
HI/BLOWN:    "PRESSURE TOO HIGH";
CLOSED:      "VALVE CLOSED"
```

The Declaration is usually terminated by a semicolon and all following Statements grouped with it (separated from it and from each other by semicolons) constitute the LSE, i.e. the set of statements for which the expanded environment of States may be tested. The effect of the Declaration is to treat the States of the included variables or elements as if they were added to the States of the Block, for the purpose of any testing. When the Declaration is executed, it records each current State value relevant (needed to support the associated combinatorial conditional test computations) to the statements operating within its declared Environment. When the last LSE statement is completed, the LSE is no longer active; the State values, applicable for any following State Prefix testing, becomes again the set in use before the LSE was declared.

If the variables in a Declaration are Real, they can each be compared to some limit, each comparison returning one of three states (**LO**, **HI**, **EQUAL**), as the case may be. The notation separates the variable from its following limit (which can be any Real valued expression) with a comma:

```
[FST, 500; FSP, (FSP1)];
```

The variable being compared is always the one before the comma; it must be a variable. The value after the comma may be a variable, constant, or Real valued expression. Ideally any variable, used as a comparison limit, will be expressed in parentheses, but this is not required by the system from configuration entry text. However, the Back Compilation will always supply surrounding parentheses about any variable or expression occurring after the comma:[30]

```
[FST, (FST1); FSP, (FSP1)];
```

In the simplest case (as above), the States returned by the LSED are added to the Block States and States of nesting LSEs. However if only the States of the LSED are desired, a variant form restarts the nesting of LSEs to include only the declared variables. The break in nesting is indicated by a '**|**' symbol as below.

```
|[M200; P200; V200];
```

State, Null, and Scoping Prefixes

Prefixes are expressions appended before a statement or Activity separated from it by a colon, which qualify its action. State Prefixes define State conditions under which the statement or Activity is to be run. They also apply to multiple statements defined on the same line separated by semicolons. In this case the statements are run in sequence if run at all. Null Prefixes are pcode artifacts, not part of the language, but representing pcode elements grouping any set of multiple statements listed on a line and separated by semicolons, to control their normal sequenced execution. Scoping Prefixes are used to express any statement which is to be executed in some other Block, as in:

```
IN ICL_BLOCK_2: C103, ON
```

State Prefixes

The basic State Prefix consists of a colon preceded by a State Prefix Expression. This Expression is either a State Condition Expression or one of the Keywords below followed by a State name:

Form	Example	Explanation
State Condition Usage (Single State)	**AUTO: C103, OPEN**	A single conditioned State (as named: **AUTO**), which, if seen as any one of the Block (or Local State Environment Declaration variable) States, will cause the execution of the rest of the statement.
State Condition Usage (Alternative States)	**AUTO/REMOTE: C103, OPEN**	A set of alternative conditioned States, any one or more of which, if seen a Block (or Local State Environment Declaration variable) States, will cause the execution of the rest of the statement.

[30] The general rule is that a variable used as an unchanged secondary data value source in a computation, will most clearly be indicated as such when surrounded by parentheses (making it an expression, but subordinating it visually as well). This is required in Discrete Assignments, but not in Real Assignments, where the conventional usage overrides such a constraint. The underlying assumption is that the variable to the left of the comma in an assignment (the sink) or in a comparison is the essential subject of that assignment or comparison; any source value or comparison limit must be visually de-emphasized in some way (in this case by hiding it in parentheses). In more advanced ICL usage, parentheses around a variable further indicate that the value of the enclosed variable is not in any way changed by the associated computation (distinguishing call by value from call by name).

single source State.

Simple Assignment; Listed Variables: **(C101,C103), OPEN** Several sink variables being independently set to the same source value. The independent variable (or Block, or other ICL element) names are enclosed in parenthesis and separated by commas.

Simple Assignment; Composite State: **C103, (OPEN, LOCKED)** The sink variables being set to a Composite source State with several named fields. The independent field State names are enclosed in parenthesis and separated by commas. If such an ANDed expression contains contradictory States, the compiler should flag the situation but apply those States which occur last in the expression. Also allows Listed Variables:

(C101,C103), (OPEN, LOCKED)

General Assignment: **C103, (C101)** The source State value is derived from an existing variable (or other ICL element). In this case the name of the source variable (or element) is surrounded with parentheses, representing the value of the variable (as distinct from the variable as a whole), but designed to de-emphasize it visually, and subordinate its role. The General Assignment introduces the complication that the distinct variables may have different State structures, because they have different associated Attributes and Idioms (with different operating States), or different States Expression declarations. For example, **C101** may have the States Expression declaration: **ON/OFF,FREEZE/RELEASE**, whereas **C103** may have **ON/OFF/HOLD,RELEASE/FREEZE**. The initial design is constrained to allow only variables with identical State definitions, including identical associated Attributes and Idioms when these have operating States. (I.e. if **C101** has the associated States Expression **ON/OFF,FREEZE/RELEASE** then so does **C103**, etc.) The form can also include Listed Variables.

Mapped Assignment: **C103, (C101(OPEN/HELD))** Same as the General Assignment except that the indicated States (corresponding to the States Expression enclosed in parentheses [**(OPEN/HELD)**]) from the source variable are mapped into corresponding sink variable States. This is the better form to use because the affected States are made explicit. The initial design restricts this form of the Assignment, to apply only when the States Expression States occur in a single field of each of the involved variables. However, the resulting States can then be mapped arbitrarily. The form also allows Listed Variables.

Self Assignment: **STOPPED** A single word statement, the word being a Block State name, causes the Block to be set to that State. In the Small System language, there is no Task Call capability. But the Block State names can correspond to intended tasks. The Self Assignment then can be made to have the appearance and function of a Call, if a Procedures Page Activity is set up to run in the appropriate State Prefixed statement.

For Simple Assignments (without source variables), the assignment causes a search (at compile time) for a Discrete Context in each sink variable, which includes the particular source State. This requires searching of all consecutive Attributes and Idioms for such a State, until a new variable definition encountered. A search failure constitutes a compile time error. For General Assignments, the source variable and the sink variable must have the same Attribute Type pattern and the same associated Idioms. For restricted assignments from variables, each variable must support all of the States of the restricting States Expression. This is the more useful assignment from variable form because there is less likelihood of an unpredictable setting. For this reason, all assignments from variables will be Back Compiled to show the collective States Expression even though the original compiled form did not include it.[29] If there is a need to assign States from source variables not matching the sink variable, this must be carried out using State Prefixed simple assignments, or Truth Tables, to make the explicit translation.

Local State Environment Declarations (LSED)

The earlier illustrations of conditional execution with State Prefixes (to be explained below) assume that the States

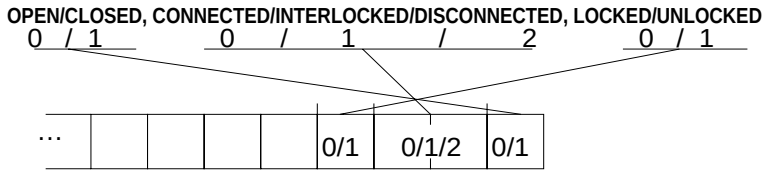
[29] For General Assignments the Back Compiled form may include multiple fields. Multiple field States Expression (Mapped) Assignments will allow compilation (recompilation) as long as the States Expressions represent identical States patterns for each of the affected variables.

Blank State name (_)). The names in a group are separated by slashes. The groups are separated by commas:

_/EARLY/LATE/TIME,BOOKED/UNBOOKED,DEFINED/OVERFLOW/UNDEFINED,FREEZE/UNFREEZE,HOLD/RESET/RESTART/START

It will also be useful to define State Condition Expressions as lists of (one or more) State names separated by either slashes or commas, but not by both. This Expression is used to express alternative States or combinations of States which must apply to satisfy certain conditional expressions (such as State Prefixes).

Normally, States Expression will be reordered on Compilation or Back Compilation (as shown) so that each State is ordered alphabetically in its group, and each resulting group is ordered alphabetically in the Expression as a whole. In this simplest form, the corresponding fields are defined from left to right in the Expression and packed from right to left in the corresponding internal integer form. The first field starts with the 1 bit, and then the 2 bit and then the 4 bit, using as many lowest order bits as necessary to contain the number of independent state codes defined by the left most States Expression group (**FREEZE/UNFREEZE** requires just one bit). Within that field the left most State name applies to the 0 field value, the next left most State name applies to the 1 field value, and so on through consecutive integers until all named States in the group have been coded. The field is restricted to however many bits are required to code all named States.



The next left most group is similarly assigned to define the field width and coding for the next right most (lowest most) field in the integer representation, and so on. In this way the Expression codes the representation, and any needed translation between the coded State names and that representation (or any associated field devices). Several variants are included:

- Two adjacent slashes (**AUTO//MANUAL**) indicate an additional State with no assigned name. This is distinct from the Blank State (The underscore (_) as in: **AUTO/_/MANUAL**).
- Two adjacent slashes with intervening dot (**AUTO/./MANUAL**)[26] indicate that the next named State is to start as a 1 in the next free bit in the integer, with zeros in the lower order bits of the field for the initial State value.
- A hexadecimal number followed by a colon preceding a group in the Expression sets aside the corresponding field mask; the group is to be coded packed into that defined field. Thereafter any further fields are defined to include the right most possible bits excluding all fields that have already been defined, even though these may involve some bits that are not consecutive. However, if the field mask for a new field is explicitly defined to overlap an existing field, this is allowed. This usage overrides the alphabetical reordering for its group. A colon by itself, before the group, also overrides the Compilation/Back Compilation reordering.

As described above, each field in the the packed field representation structure of a State variable is independent of any other. But in the practical use of States, some States only become meaningful if the system or variable is already in some other state. The following notation could be used to represent that kind of situation, with remaining fields being defined conditioned on the State of already specified States. With this extended notation, a parenthesized Conditional Declaration is added following a comma, consisting of a State Condition Expression, a colon and a States Expression:

UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/OVERFLOW, (FREEZE/BOOKED: LOCKED/UNLOCKED) or

UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/OVERFLOW, (FREEZE/BOOKED: LOCKED/UNLOCKED), (UNBOOKED: SIMPLE/COMPLEX)

This total Expression means that the additional field **LOCKED/UNLOCKED** will be defined, but only if the **FREEZE** or **BOOKED** States apply. Either Expression within the parentheses may be as complex as desired, if consistent with the above rules. There may be multiple sets of conditional States, either defined independently in comma separated parenthetical expressions, as above, or in consecutive State Prefixes separated from each other by semicolons as below:

UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/OVERFLOW, (FREEZE/BOOKED: LOCKED/UNLOCKED; ELSE: SIMPLE/COMPLEX)

Note that the corresponding data field definitions may be allowed to overlap if the associated conditional State Condition Expressions apply exclusive of each other; the ELSE related notation above is particularly appropriate for this usage.

Discrete Assignments

As illustrated earlier, the Discrete Assignment has its own assignment operator (a comma) and form. In general this consists of a left hand sink[27] variable, a comma, and a right hand source[28] state or State Expression. The form is designed to visually emphasize the sink variable and subordinate the source; the sink is intended to be viewed as the subject of the statement and center of attention. There are a number of variants:

Variant Type	Example	Explanation
Simple Assignment:	**C103, OPEN**	A single sink variable (or other named ICL element) being set to a

[26] 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.

[27] The variable being set or changed.

[28] The variable providing the data.

Expression (before the first dot) must match the Name of the Block (or Box).

In the above example, all references from outside must start as:

ICL_BLOCK. … .

Thereafter, the SuperVariable and Reference Expression Names are scanned in parallel, from left to right, starting with the first unmatched Reference Expression Name and the first Attribute.

The rest of the process can be summarized in the following numbered steps, in order:

1. If the next Attribute is a Name Attribute, its Name (value) is matched to the current Reference Expression Name. If there is a match, the Reference Expression scan is advanced to the next Name and the SuperVariable scan is advanced to the next Attribute. This step is then repeated on the new pair of elements.

   In the above example, a reference to the first variable must start as:

   T101. … .
2. If the Attribute is not a Name Attribute, its Type Name is matched to the current Reference Expression Name. If there is a match, the Reference Expression scan is advanced to the next Name and the SuperVariable scan is advanced to the next Attribute. The process then is restarted at the first step, for the new pair of elements.

   In the above example, a reference to the HI Attribute of the first variable continues as:

   T101.HI. … .
3. If the scan encounters a (Filter/Compensations or Control) Idiom in the Attribute scan, and the Reference Expression Name matches one of the Idiom associated Parameters, then this Parameter defines a match.

   In the above example, a reference to the first Loop Statement REGULATE Idiom PB parameter becomes:

   T101.PB .
4. If no match is encountered with the current SuperVariable Attribute (or Idiom), the SuperVariable scan is advanced to the next Attribute, and the process restarted at the first step.
5. If there are no further Reference Expression Names, then the Attribute scan advances to the first Attribute including an internal data field appropriate to the Context of the reference. If the Context is Discrete (as in a Discrete Assignment) the Attribute advance continues until an Attribute or Idiom is found whose system or user defined State (defined in a following States Attribute) matches the requirement of the original reference Context.

   In the first Footnote, the **T104**, **T101**, *and* **T102** *references are effectively altered to:*

   T104.VALUE. … , **T101.VALUE.** … , and **T102.VALUE.** … , respectively.
6. If there are no further Reference Expression Names, the last Attribute was a Name Attribute, the Context condition has been satisfied, the referenced variable is under Idiom control, and the referencing statement will set the variable, then the Attribute will be advanced to the next corresponding Set Attribute.

   In the RAMP Statement and its second Footnote, the **T101** *reference is further altered to become effectively:*

   T101.SET .
7. If there are no more Attributes, but more Reference Expression Names, and one of the above steps has not been satisfied, then a reference error has occurred. This may have one of the following causes:
   - At compile time, the Reference Expression is in error. The compiler will notify the user.
   - At compile time, the referenced object has not been defined, or an external Referenced object is not visible to the compiler. The user can override the notification. If the error remains at the end of the session, the notification is repeated.
   - At run time, any internal reference remaining in error will cause an operator notification before executing. The notification must be explicitly overridden.
   - At run time, any external reference remaining in error, without an internal **FAIL:** or **FAIL::** State Prefixed statement set to run immediately if the reference is encountered, will cause an operator notification before executing. The notification must then be explicitly overridden.
   - At run time, any external reference encountered running in error (previously overridden), without an internal **FAIL:** or **FAIL::** State Prefixed statement set to run immediately if the reference is encountered, will cause an operator notification before executing. The application will continue to run unless then acted on by the operator.

The final result of all of this scanning and matching is a match of the Reference Expression to the desired Attribute data value.[25]

Specification of Remaining Language Elements

The preceding discussion alludes to Activities, States, and State Prefixes or other conditional computations. This section details these as well as certain more elaborate (fanout) Idioms.

Discrete Data, States, and State/States Expressions

A Discrete data value is represented in the language by a list of appropriate State names separated by commas, and internally as an integer made up of appropriate packed fields. The list is referred to as a State Expression. The structure of the data value and the mapping between packed field and the corresponding State Expression is defined in a States Expression. This defines the packing of one or more fields into an internal State value integer representation. It consists of one or more groups, each of at least two State names (which may include the Null or

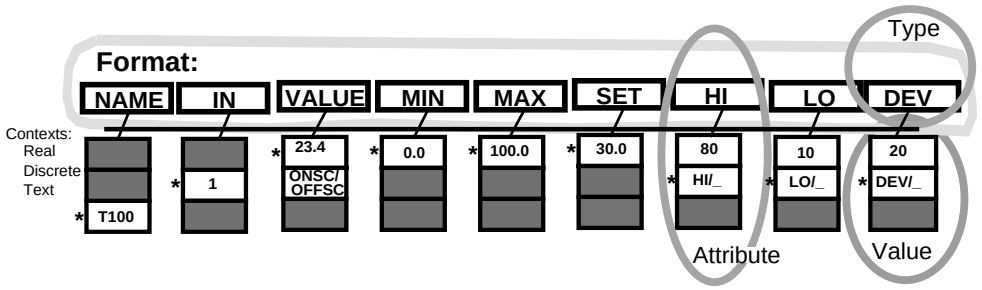
[25] There are two easily implemented extensions to the Reference Expression, useful in general external references and desired by several users: If the dot in an expression is replaced by a double dot (as in **P100.PB..DIR**) this means that all Attributes or Parameters occurring, between that matched before the **..**, and that matched after, are to be returned. Similarly if the dot is replaced by a comma, then each item Attribute matched before a comma in the expression is returned. Since these references are likely to be used for external higher level access, all such references will be assumed to represent a Context which calls for all associated Contexts or Parameters of each referenced Attribute or Idiom.

External Access; the SuperVariable and Zipper Reference

The Small System design focused on the SuperVariable for two reasons:

- The small system naturally emphasized extensive process variable I/O and processing.
- It seemed that the most natural functional interface to a small system was through its variables.

With these two facts, a simplified design based on hanging all computing and control function within the natural variable/I/O scan arose naturally. In this section we address the consequences of this concept as it affects reference to data inside the application, or external to it, handled by the Object Manager. The figure below, highlights the character of the SuperVariable as seen from the outside; it consists of a sequence of typed data elements called Attributes (which in general correspond to the Attribute pcode elements discussed later). Conceptually, the Attribute is characterized by type and value. For the Large Language, the sequence of Types, making up a pattern and defining a definition structure, is called a Format; the Small Language constrains the Type code to be part of the Attribute and does not make significant use of the Format concept.

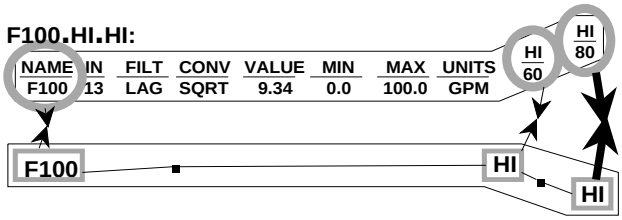


From the users point of view, the value data (of any Attribute) may include data of one or more (depending on the Attribute type) of three data types called Contexts: Numerical (which can be Real, Time based, or Counting), Discrete, or Text. Any reference can require a given Context, depending on the matching Context of the referencing computation. In addition, the language statement forms (Idioms, Assignments, Theme Statements) differentiate between the reading of an actual measured value and the setting of an intended setpoint for a measured value. This differentiation is called Idiomatic reference. The reference notation for the language, and for all external references takes these issues into account.

The notation takes the usual dotted structure form (e.g. **F100.HI** would reference the high alarm Attribute of the **F100** variable). This raises two other considerations related to the interplay of reference with the unique SuperVariable structure: First, where ever possible, the basic data of a variable should be referenced by the variable name (without requiring the Attribute Type name). Thus, in English, we can refer to the value of **T100** as **23.4**; we don't need to talk of it as **T100.VALUE**. But in a different context, we may refer to **T100** as being On Scan (**ONSC**), without needing to refer to an operating State Attribute Type name. The Context concept takes care of this automatically, accessing the first appropriate Attribute Context after the Name Attribute. [But if this concept is confusing **T100.VALUE** will also work.]

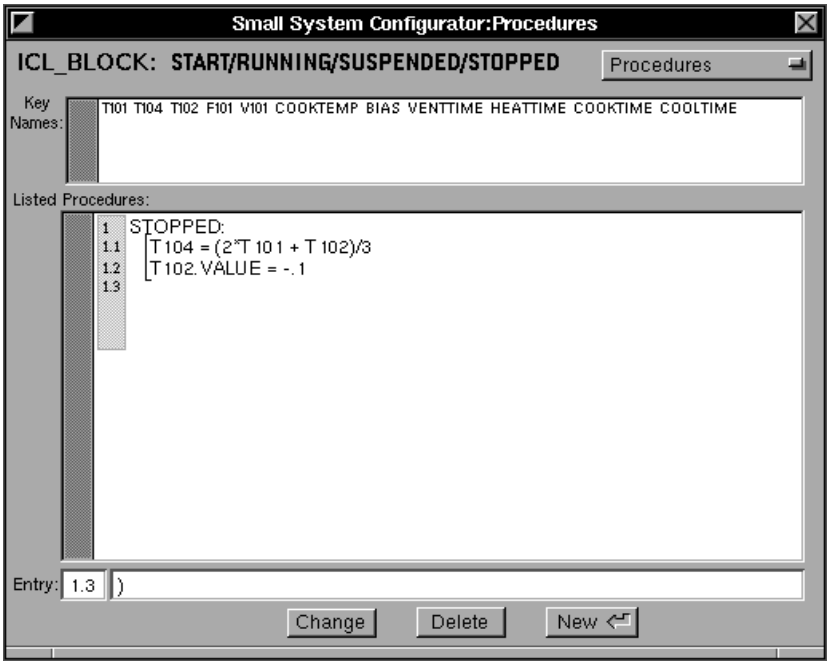
Second, the reference structure should be able to accommodate multiple Attributes of a given type, such as multiple Hi alarms. The notation accommodates this problem by allowing repeated Attribute names to represent corresponding repeated Attributes within the SuperVariable, as illustrated below. The SuperVariable also allows particular Attribute headings to be renamed by the user to allow that particular column of Attributes to be called by a particular user selected name even as it functions normally. For example, if the second HI Attribute heading, corresponding to the second Hi alarm in the definition below, were entered as **HH(HI)** (meaning an Attribute type of **HI** renamed as **HH**), then the reference **F100.HH** would be equivalent to **F100.HI.HI**.

The overall reference structure can be viewed as a matching process between the desired reference in the notation, and the Attributes as ordered in the SuperVariable. Because of the appearance of this process, illustrated below, we will refer to the reference form as Zipper Reference.



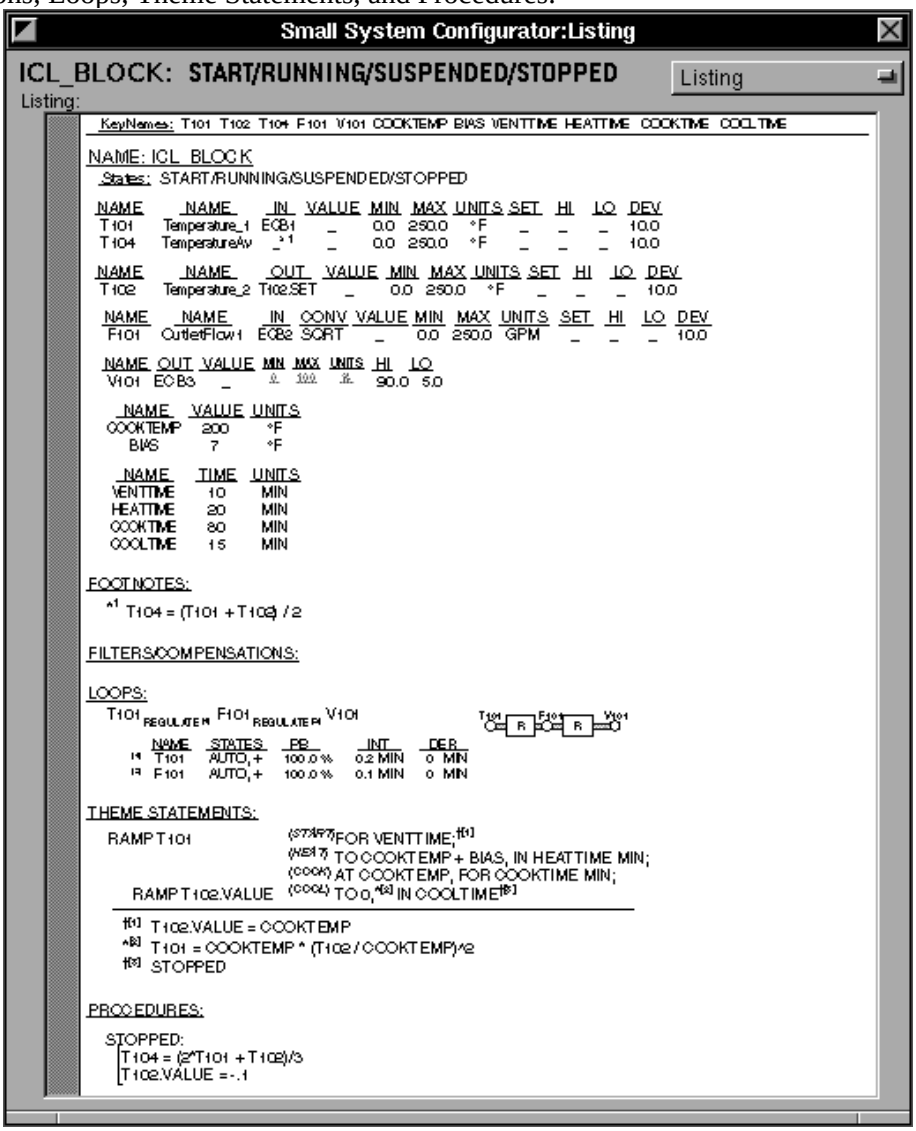
The Zipper Reference process can be summarized starting with the initial step, executed only once:

- If the reference is made from outside the ICL Block (or Box; through the Object Manager), the first Name in the Reference

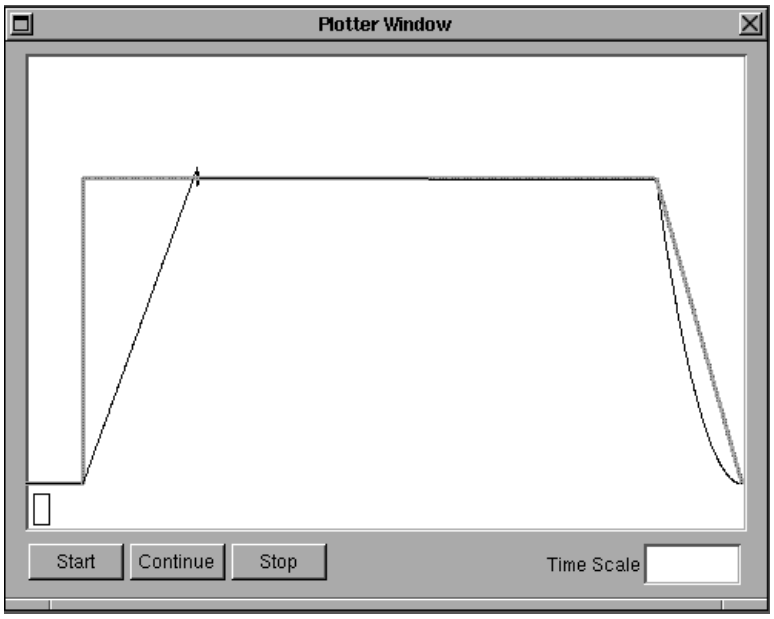


The Overall Example Listing

The Paragraph menu includes a single overall Listing display combining all Paragraphs: Definitions, Footnotes, Filters/ Compensations, Loops, Theme Statements, and Procedures:



Footnote (in the middle of the Phrase as with Footnote 2) is executed repeatedly/continuously for the duration of the Phrase. The secondary Ramp Phrase RAMP T102 switches the profiling action to apply to T102 instead or T101. The final result carries out the specified profile, with the Footnotes coordinating those actions not directly profiled by the Phrases. The figure shows the resulting control:



The final form of the Ramp Statement for the example includes one other feature. The statement form allows the definition of user named States for each Phrase involving continued computation. This allows operator display and access to the running Phrases. The States are defined by assigning preceding State names to each such Phrase (italicized, in parentheses, and superscripted):

```
RAMP T101:          (START) FOR VENTTIME;†[1]
                    (HEAT)  TO COOKTEMP+BIAS, IN HEATTIME;
                    (COOK)  AT COOKTEMP, FOR COOKTIME;
          RAMP T102:(COOL)  TO 0*[2], IN COOLTIME†[3]
```

---

†[1] T102 = COOKTEMP

*[2] T101 = COOKTEMP * (T102 / COOKTEMP)^2

†[3] STOPPED

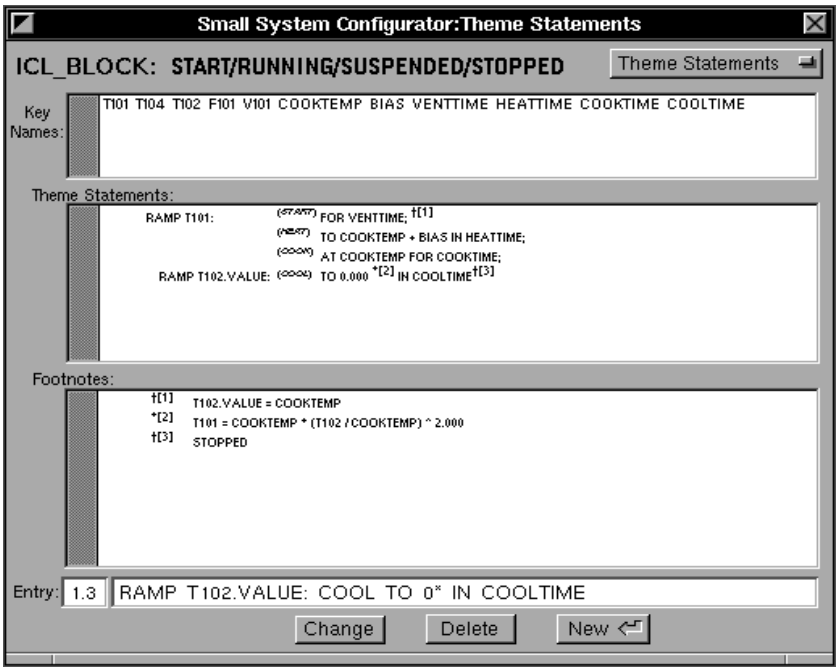
The Theme Statement Paragraph can in fact contain any computational Statement or Activity, even though it is principally designed to support the Theme Statements. Theme Statement Paragraph statements are executed in the scan before the variable Definition Attributes.

Procedures Paragraph

The Procedures Paragraph is intended to contain more Complex or higher level Activities (introduced above and detailed later), not reflecting a standardized application related structure. It can contain any kind of computational statement or Activity. These will be executed at the end of the scan after all other statements and Attributes. Normally these will be Activities preceded by a State Prefix (detailed later) whose State is one of the Block States (entered in the Header Display). This has the effect of causing the Activity to run whenever the State is entered[24]. Since the State can be entered as the result of a statement consisting of the single State name, these State Prefixed Procedures Paragraph Activities appear to act like conventional Procedures. Hence the name. More generally, any other (unPrefixed) Procedures Paragraph statement or Activity executes once whenever the Block as a whole is activated. The figure shows a single Sequential Activity which is executed once to completion, whenever the **STOPPED** State is entered.

The figure shows the final termination of the Activity. The initial line in the Procedure definition would have been entered as **START:(**© [including an unmatched left parenthesis and a carriage return; see earlier footnote on © indicating a carriage return]. This would initialize the Sequence Activity Bracket in the listing. Additional ©'s would switch the type of Activity Bracket to Parallel, Looping, Continuous, and State Driven. This alternation of Bracket types (also labeled in letters) would stop with the entry of the next statement (**T104 = (T101 + T102)/2**). The final line, consisting only of a)© [whose) would match the initial (in the **START:(**© line], terminates the Activity.

[24] The Activity will be aborted whenever the State ceases to apply, if it is still running at that time.



The Statement consists of an initial Ramp Phrase, which defines the initial profile variable, followed by later Phrases which further define the profile being applied to that variable:

```
RAMP T101:      FOR VENTTIME;
                TO COOKTEMP+BIAS,  IN HEATTIME;
                HOLD AT COOKTEMP,  FOR COOKTIME;
                TO 0,  IN COOLTIME
```

Semicolons terminate lines between related phrases.[22] As the statement is executed, the single variable T101 is first held constant for VENTTIME (minutes), then ramped to COOKTEMP+BIAS, in HEATTIME minutes, then held at COOKTEMP for COOKTIME minutes, and cooled to 0 (degrees) in COOLTIME minutes. This defines the basic profile shape for T101 specified earlier. However T102 needs also to be ramped. The Statement can be modified adding Footnotes and a secondary Ramp Phrase. As above, he lines of multi-line Footnotes are separated by semicolons. The final Footnote in the example [3] sets the system State to STOPPED after the statement is completed. It represents an abbreviated Discrete Assignment.

In ICL, the Discrete Assignment is distinguished from the Real Assignment by its use of a comma operator rather than an equal sign operator (**C103,OPEN** rather than **F100.SET = 23**). This reflects the intended natural English usage and facilitates the use of the ICL Context concept.[23] The language permits the simpler form when used to set the Block state, leaving off the Block name and comma and simply applying the single word State name, as done here. The particular STOPPED Assignment relates to the following Procedures Paragraph.

```
RAMP T101:      FOR VENTTIME;†[1]
                TO COOKTEMP+BIAS,  IN HEATTIME;
                HOLD AT COOKTEMP,  FOR COOKTIME;
RAMP T102:      TO 0*[2],  IN COOLTIME†[3]
```

---

†[1] T102 = COOKTEMP

*[2] T101 = COOKTEMP * (T102 / COOKTEMP)^2

†[3] STOPPED

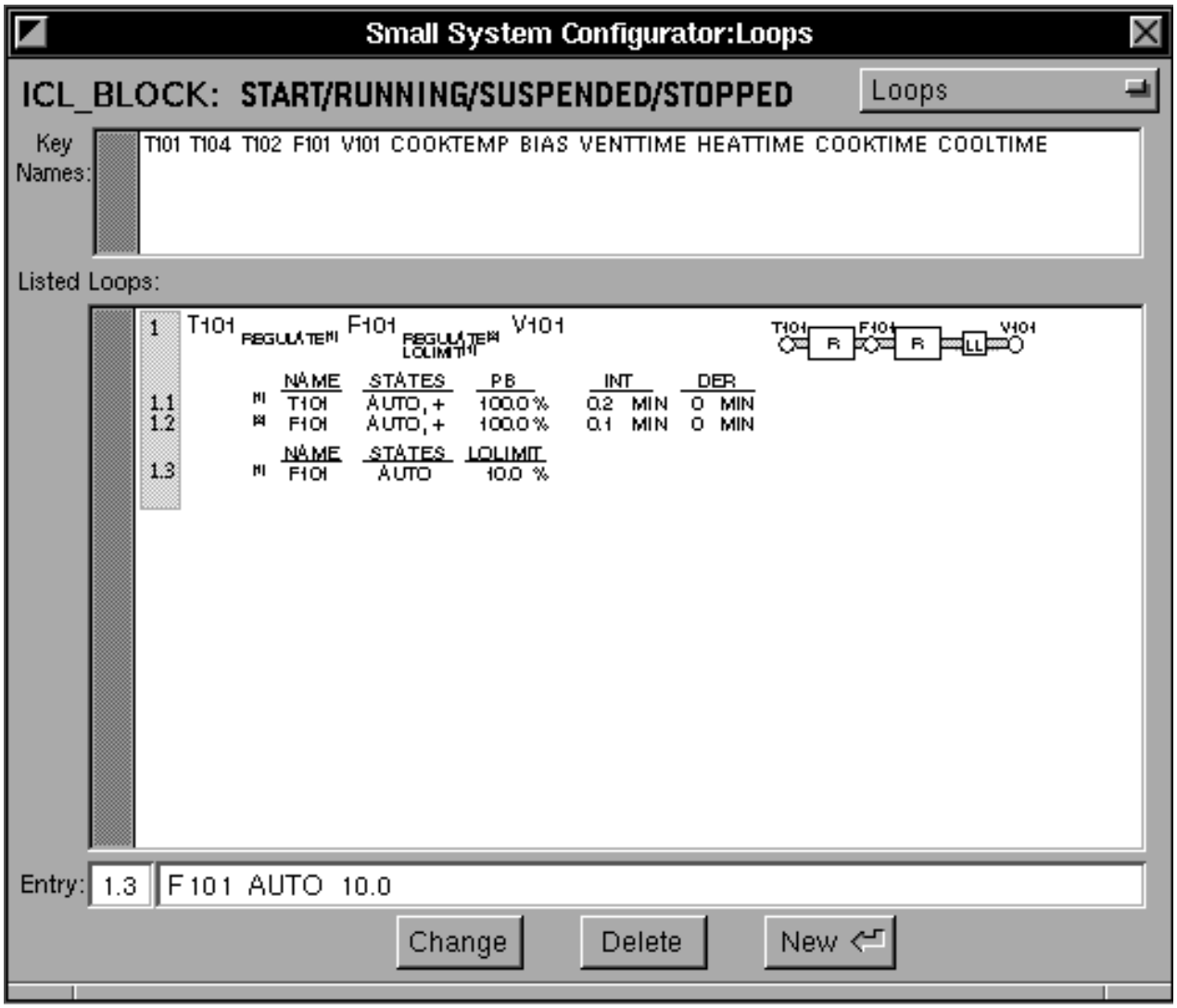
A Phrase ending Footnote (as with Footnote 1) is executed once when the Phrase has been completed. An embedded

[22] Or more generally, lines between related statements in a program. The semicolon has a special ICL role. When used to separate statements listed on a single line, these statements are executed sequentially in real time (suspending in the middle of the line when encountering any incomplete statement). This is the only way that multiple statements can occur on a line. When used to terminate a line, they express the relationship between the line (and its included phrases and statements) with the following line (and its included phrases and statements).

Lines will always be terminated by a carriage return (in addition to any semicolon). Because the carriage return itself has, later defined, special functions, it will be indicated by the © character in the later discussion.

Note that the RAMP phrases are terminated by colons; where possible, failure to include a Theme Statement colon or semicolon delimiter will be ignored on input and generated by the Back Compiler.

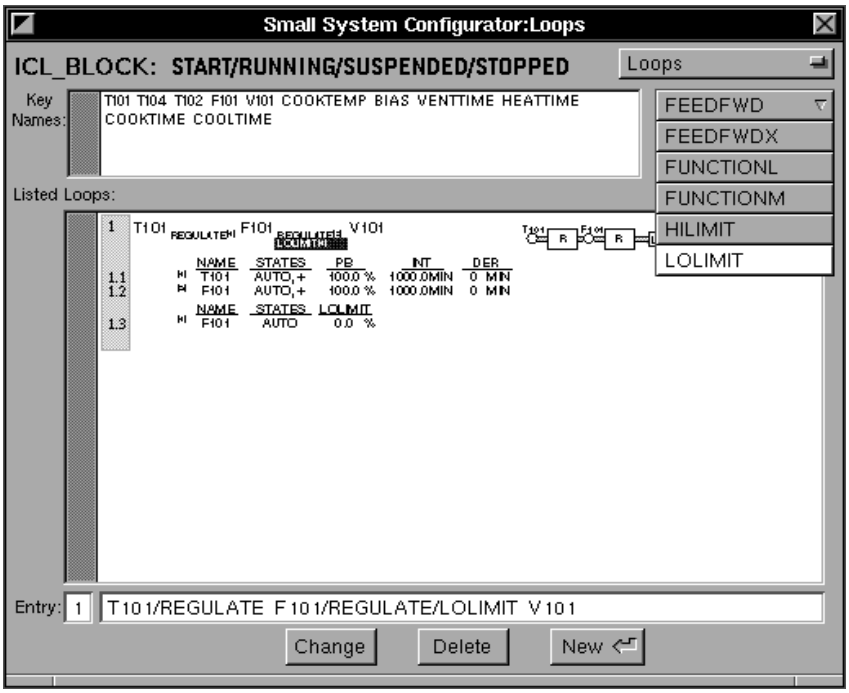
[23] As will be seen later, the distinction also permits the simple setting of operating and control States associated with a process variable, directly in terms of the State and variable name, without reference to some obscure Discrete Parameter. For example: **F100=23** and **F100,AUTO** allow the independent control of the Real and State data for the variable without reference to any other name construct. At the same time the use of a natural separate operator symbol disciplines the distinct usage and ensures that the State names will never be confused with variable names.



An Idiom Loop Statement may include a preceding State Prefix (detailed later), making its execution conditional on the associated State. In such a case, the Loop is active whenever the State applies and inactive otherwise. The transition to active and inactive conditions will take place bumplessly. If several Loops involve the same variables but different State Prefixes, a bumpless transition will occur between those Loops as the corresponding State changes occurs.

Theme Statements Paragraph

Theme Statements have their most relevant usage in the Small System ICL, as the coordinating framework for small applications not bound together by Large System ICL Operations. ICL contains special Theme Statements for representing naturally structured sequencing actions. In the example, the RAMP Theme Statement is used to impose the heating, cooking, and cooling profile. Its final configuration form is illustrated in this figure:



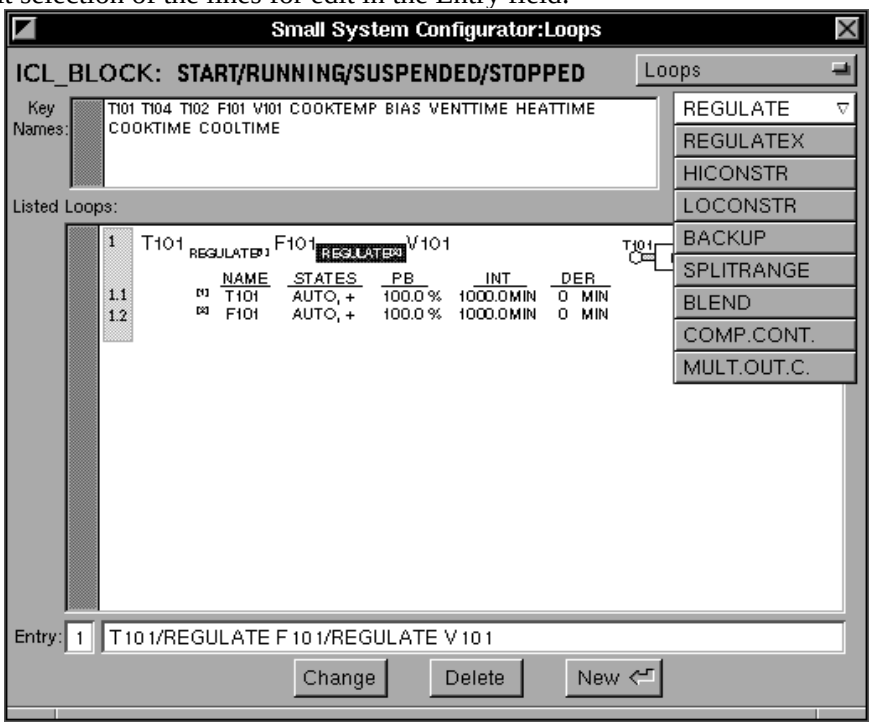
The figure shows the separate application of a Secondary LOLIMIT Idiom to the F101 REGULATE Primary Idiom, creating extensions to the illustrating graphic and a new Parameter Table. The menu listed Secondary Idioms are:

- FEEDFWD: Feed Forward, whose subscript expression always includes a feedforward variable, as in FEEDFWD FF1.
- FEEDFWDX: Feed Forward with EXACT, also configured with its feed forward variable listed.
- FUNCTIONL: Function Idiom in Loop, which allows a break point function to be applied on the output of the Primary Idiom.
- FUNCTIONM: Function Idiom on Measurement which allows a break point function to be applied to the Idiom input variable being applied separately to both Setpoint and Measurement.
- HILIMIT: Hi Limit, applied as a high limit on the output of the Primary Idiom (including the effects of intermediate Secondary Idioms).
- LOLIMIT: Lo Limit, applied as a low limit on the Primary Idiom output, as above.
- HRLIMIT: Hi Rate Limit, applied as a high limit on the rate of the output of the Primary Idiom.
- LRLIMIT: Lo Rate Limit, applied as a low limit on the rate of the output of the Primary Idiom.

The Parameter Table values in the above figures represent defaults. Once the Loop has been completed, the Tables can be edited to provide preferred values:

T101/REGULATE F101/REGULATE V101.

The automatically displayed square brackets number and reference special Footnotes. These contain the tuning setting of the corresponding controllers, illustrated later. The Loop Statement is also illustrated automatically by the system by an iconic diagram (also shown later). The sequence of figures below illustrates the consecutive steps in configuring such a Statement. The first figure shows the above configured Statement with menu, and associated Parameter Footnote Table. As Idioms are added, the associated iconic graphic develops and the corresponding square bracket references and Parameter Table Footnotes are created with default values. The line numbers are added automatically to permit selection of the lines for edit in the Entry field:



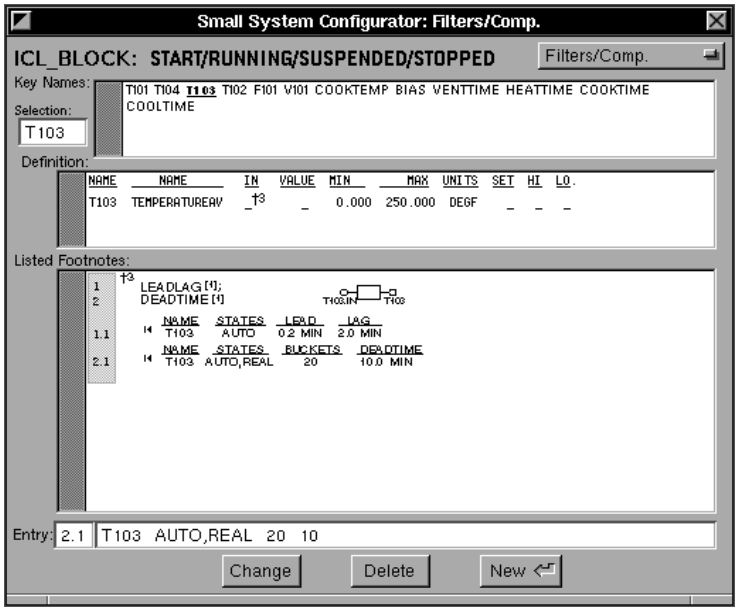
The Idioms shown in the menu are:

- Primary Idioms: REGULATE (PID Control) and REGULATEX (PID Control with EXACT). Primary Idioms always occur as the top Idiom in an Idiom subscript.
- Constraint Idioms: HICONSTR (PID/Selector Hi Constraint Control) and LOCONSTR (Lo Constraint). Constraint Idioms always occur as the top Idiom in an Idiom subscript.
- Fanout Idioms:
  - BACKUP: Defining an alternative PID Control path when a main loop is bounded or constrained.
  - SPLITRANGE: Split Range Control, based on a special generalized multi-path external feedback fanout algorithm, permitting, when desired separate tunings in the split loops.
  - BLEND: Paced Blend Control.
  - COMB.CONT.: Fuel/Air Ratioing Combustion Control.
  - MULT.OUT.C. or MOC: Multiple Output Control.

Fanout Idioms occur either as the top Idiom in an Idiom subscript, or immediately below a Primary Idiom.

The Loop Statement also allows for Secondary Idioms, conditioned on Primary Idioms. These constrain or improve the execution of their associated Primary Idiom without altering its basic intent:

various controlling Loop Statements. This will be the order visible in the Key Names. If a Loop Statement is configured to list its variables out of this order, an error message will so indicate. the compiler may then be instructed to apply a minimum reordering to the variable definitions, to remove the inconsistency.



This document will not generally spell out the parameterization of the Compensators and Idioms. The illustrations will include the intended parameterizations for those Idioms covered. The design goal in ICL is to restrict the parameterization of all functions to the minimum inherent set, and rely on the flexibility and ease of use of the language, to allow the user to create his own combined functions. This relieves the user from the need to learn large numbers of arcane parameters and functions, but allows him to achieve any needed control action easily.[18]

Loops Paragraph

A Control Idiom is a special function representing some basic control intent and its implementation. Control Loops are listed in the Loops Paragraph as Idiom Loop Statements, combining variables and associated Idiom names. The Loop Statement has several advantages, over traditional control Blocks, derived from its inherent nature as a higher level description of the intended integrated behavior of the included Idioms:

- The collective behavior of the Loop is immediately visible to the user.
- It becomes possible to automatically configure operator displays that take into account the intended collective relationship and behavior of the control functions.
- As implemented, the execution of the underlying control algorithms can be carried out in optimal order, separating those parts of the algorithm which should be carried out in order from primary measurement to valve, from those which should be carried out in order from valve to primary measurement.
- As expressed and implemented with the ordered execution, the passage of data between variables and algorithms is implicit, requiring neither expressed connection nor internal pointer connections.
- Control and alarm Setpoints are a standard part of the variable and its declaration rather than a part of the control function, separately declared for each of several control and alarm blocks. Further, within the Idiom and Loop Statement expression and implementation, the distinction between a variable and its associated Setpoint is implicit.[19]

The Loop Statement consists of the sequence of cascaded variables, ordered from primary (controlled variable) to secondary (to tertiary etc.) to the final valve. Each such variable, which acts as controlled variable to some control intention and function, has the name of the corresponding Idiom subscripted (there may be several arranged vertically: generally an initial Primary Idiom followed by a Secondary Idiom as described below) after it. In the example, the Loop Statement below specifies a cascaded control of T101 to F101 to V101. The Idiom, REGULATE, represents the normal regulatory function of a PID controller:

$$T101_{REGULATE[1]}\ F101\ _{REGULATE[2]}\ V101$$

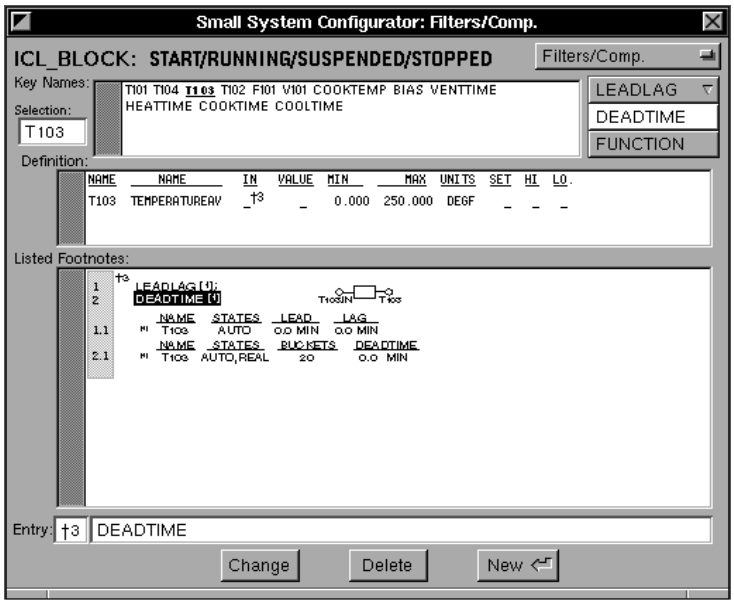
The Statement words can be entered optionally with keyboard or menu; the above statement can be either entered by alternation of T101 F101 V101 keyboard entry and menu selections, or by keying the Statement directly as (each Idiom preceded by the forward slash[20]):[21]

[18] While not specified here, some well thought out macroing mechanism for letting the user define standard combinations of SuperVariable Attributes and Idioms may be useful. The Back Compiled result might also label the use of such macros, but it should, in any case, spell out the components as if they had been entered without the macro.

[19] This same implicit connection between measurements and associated setpoints is carried throughout the language: In general, unless explicitly programmed to the contrary, references to a measured variable apply to the associated setpoint whenever the underlying function would set the value, and apply to the value Attribute whenever the function reads the value. For example RAMP T100, will apply a ramping value to the T100 setpoint.

[20] In this case the slash is chosen to represent the successive upper and lower text elements, by analogy with the way that 1/2 can be used to represent $\frac{1}{2}$.

[21] As indicated earlier, the variables in the SuperVariable must be ordered in the same order (Primary to Secondary to … to Valve) as they will occur in the

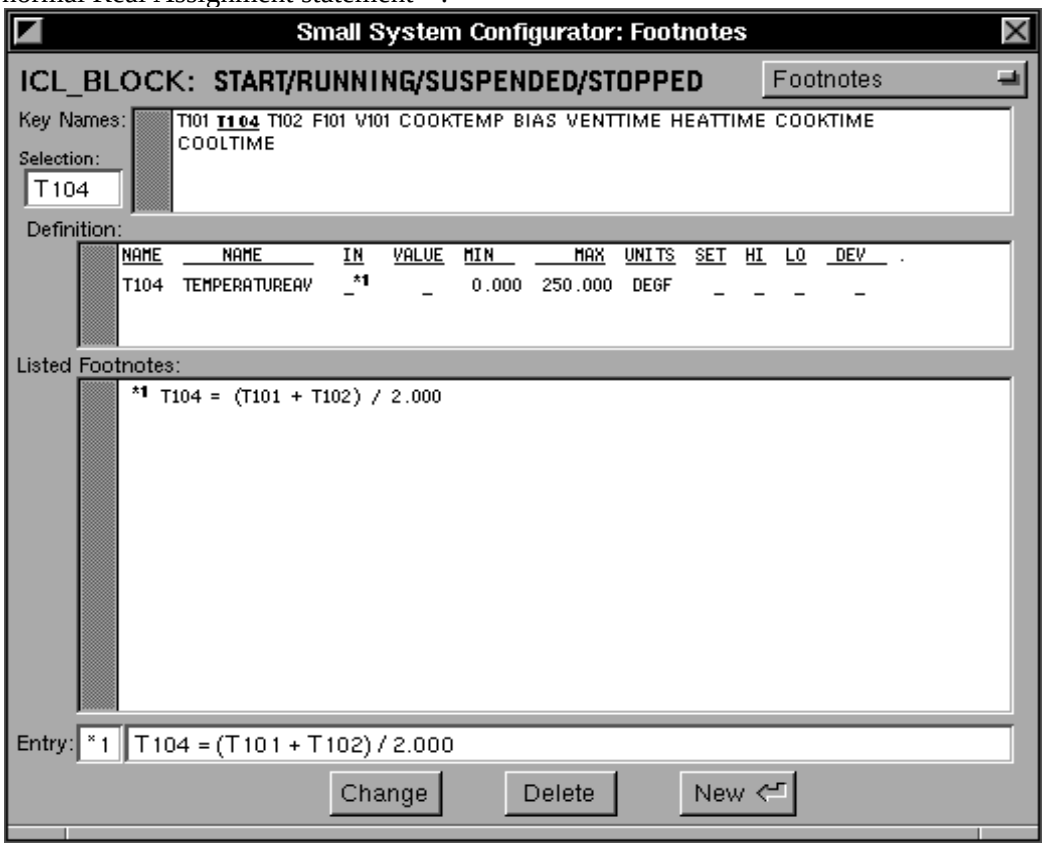


The current thinking includes several kinds of Filter/Compensations Idioms like those shown in the pulldown menu:

- LeadLag Filters with user set Lead and Lag time constants (both defaulted to 0.0). (And possibly Gain and Bias as well.) The operating States are AUTO/_/MAN, indicating: a fully operational dynamics filtering, an inactive pass through no dynamics kind of operation, or an inactive function which passes no data at all, respectively.
- (Lag Filters might be specified, explicitly, implemented as a LeadLag Filter with zero Lead time (and non-zero Lag time). The Lead time would be included in the parameter table where it could be set to a non-zero value, thus preserving user control over the changing type. The Back Compilation would chose Lag or LeadLag Idiom name based on the current Lead and Lag time value.)
- Totalizer, with Integrating Time (and possibly Gain, in any case both defaulted to 1.0; Gain and Integrating Time are redundant computationally, while expressing different user perspectives), and Integrating Time Units (a code with the same value range as the Time Units Attribute code). The operating States are INTEGRATING/HOLD/RESTART/RESET, indicating: an active integration, a held value, integration restarted from zero, or integration reset inactive.
- DeadTime Filters, using the existing IA algorithm with user specified number of Buckets and DeadTime. If the number of buckets matches the DeadTime divided by the Sample time, the DeadTime acts like a normal bucket brigade DeadTime; a real data shift register. Otherwise, the special variable DeadTime, bucket averaging algorithm form is invoked. The operating States are AUTO/_/MAN/REINIT,REAL/DISCRETE/ALL, the AUTO/_/MAN States interpreted as above. The REINIT State reinitializes all buchets to the input values. The REAL State indicates that only the Real value of the variable participates in the bucket brigade/shift register action. The DISCRETE State indicates that only the State value of the variable participates in the bucket brigade/shift register action. The ALL State indicates that all of the data for the variable participates in the bucket brigade/shift register action.
- Function Compensation, represents a ten segment breakpoint function compensator. This compensates the input data only, distinct from the breakpoint Function Idioms operating in the Idiom Loop Statements below.

Like the Loop Idioms below, the Filter/Compensations Idioms will generate associated Parameter Tables, with initially defaulted values, as shown above. The entries will be grouped by Idiom Type under a common set of Type Headings, and automatically numbered within each Type. The line editing can then be applied to alter the associated values, as in the figure below:

Definition. If more than one Footnote applies to the Definition, each is listed with its number; the number of the desired Footnote must be selected (here, by entry into the small selection field). The Footnotes entry (initially blank) can then be entered or edited in the main Entry field. The illustrated Footnote corresponds to the earlier asterisk in the T104 definition, and represents a continuous computation of T104 as the average of T101 and T102 temperatures, expressed as a normal Real Assignment statement[15]:



Footnotes can contain multiple statements (or Activities) separately listed, each on its own line (or lines). All but the last such statement will be terminated by a semicolon.

Filters/Compensations Paragraph

Filter and Compensation Footnotes (not shown in the example) are edited similarly to the computational Footnotes, based on specially named Filter/Compensation Idioms.[16] They are marked by daggers (automatically generated after the Footnote is fully defined) instead of asterisks to distinguish the kind of Footnote reference; the two kinds of Footnote are also separately numbered. For input Footnotes, each such Footnote takes as an input, either an IN or VALUE Attribute associated with its Footnote asterisk, and outputs to the following VALUE Attribute. For output Footnotes, the output is to the OUT Attribute associated with the Footnote asterisk, and the input is taken from the following VALUE Attribute.[17] The Filter/Compensation Idiom in such a Footnote executes a continuous (dynamic) compensation between the input and output. These Footnotes can be cascaded to create a more complex filter or compensation, as shown. All but the last Idiom (statement) in the set will be terminated by a semicolon. Footnotes can in fact include mixed (computing and Filter/ Compensation) forms; the appropriate Paragraph is defined by the first statement or Language form in the Footnote. But the two kinds of Footnotes will act independent of each other. The Filter/Compensation Idioms can be entered in through the Keyboard as with the above computational Footnotes. Or they can be entered by menu selection as shown. The figure below shows a combined LeadLag DeadTime compensation, with optional menu entry. Filters/Compensations Idioms will be illustrated automatically by the system with an iconic diagram emphasizing the input/output Connections.

[15] The Real Assignment represents the basic means for carrying out general purpose Real data computations, as defined later. Its form generally follows the assignment form taken in conventional computer languages, except it is restricted to using Real operators and functions. It (with its Discrete counterpart) also allows a single value to be assigned to a number of listed (in parenthesis) variables: **(D, E, F) = 23**. Its detailed specification will be given later.

[16] These filters and compensations must be parameterized or tuned to the process for control, unlike the simpler FILT Attribute smoothing filters or CONV Attributes.

[17] This requires a reversal of processing of input data (from the Value Attribute), Idiom, and actual output, from the normal SuperVariable scan order.

NAME	DOUT[14]	STATE	STATES	SET
C101	ECB3	OFF,UNLOCKED	ON/OFF,LOCKED/UNLOCKED	ON.

VALUE and I/O Attributes (and other ICL objects) have associated system defined States, controlling their operation and characterizing their current operating State. For current purposes, the State names in the Table below will be more or less self-evident. More detailed discussion is seen in the Large System ICL Specification.

Attribute Name	System Defined States	Special Comments
VALUE	**UNFREEZE/FREEZE,UNBOOKED/BOOKED,UNDEFINED/DEFINED/DEAD**	
	I(1)/O(2)/Scaling(4)/EndLoop(8)/Conv(16,32,48)	Special scan control code; with bit numbers
STATE	**UNFREEZE/FREEZE,UNBOOKED/BOOKED,UNDEFINED/DEFINED/DEAD**	Normal Case
"	**CONFIGURE/SETUP/SIMULATE/OPERATE, START/RUN/SUSPEND/MANUAL/CONTINUE/NEXT_STEP/AUTO/ABORT/END, ACTIVE/INACTIVE, _/INITIALIZE**	Special System State Attribute Case Additional States for the initial System (i.e.Block) STATE Attribute.
TIME	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/OVERFLOW, TIMING/HOLD/RESTART/RESET,_/EARLY/TIME/LATE**	For normal relative timing; Units: **SEC/MIN/HOURS**
TIME	**UNFREEZE/FREEZE, UNBOOKED/BOOKED,UNDEFINED/DEFINED/OVERFLOW, SET/RESET,_/EARLY/TIME/LATE**	For absolute dating/ timing; Units: **DATE**.
COUNTS	**UNFREEZE/FREEZE, UNBOOKED/BOOKED,UNDEFINED/DEFINED/OVERFLOW, COUNTING/HOLD/RESTART/RESET/ADVANCE, _/UNDER/DONE/OVER**	
IN	**OFFSCAN/ONSCAN,BAD/GOOD**	
OUT	**OFFSCAN/ONSCAN,BAD/GOOD**	
DIN	**OFFSCAN/ONSCAN, BAD/GOOD**	
DOUT	**OFFSCAN/ONSCAN, BAD/GOOD**	
FILT	**0(UnDefined)/1(.1 SEC)/2(1.0 SEC)**	Special State coded Filter choice.
CONV	**UNDEFINED/SQRT/SQR/LN/EXP/…**	Special State coded Conversion choice.
TUNITS†	**SEC/MIN/HRS/DATE**	Special State coded Time Units choice.
RSET*	**UNFREEZE/FREEZE, UNBOOKED/BOOKED,UNDEFINED/DEFINED/DEAD**	**UNDEFINED** applies to initial definition or assignment; **DEAD** applies then to later loss of control.
SSET*	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD, _/ALARM**	
TSET*	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD**	
CSET*	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD**	
HI	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD, _/HI, OFFSCAN/ONSCAN**	
LO	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD, _/LO, OFFSCAN/ONSCAN**	
DEV	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD, _/DEV, OFFSCAN/ONSCAN**	
ACCUM	**UNFREEZE/FREEZE, UNBOOKED/BOOKED, UNDEFINED/DEFINED/DEAD, ACCUMULATING/HOLD/RESTART/RESET, _/LO/DONE/HI**	Provides a built in Integrator/Accumulator function.

† A variant UNITS Attribute, applicable to TIME variables, here distinguished by the TUNITS name of its pcode element.

* These represent variant forms of SET Attributes for Real, State, Time, and Counts valued variable. The Attribute names are all the same (SET); the above names represent pcode element names. The system distinguishes between them when compiled.

Other control related ICL Attributes or objects have their own defined operational States.

It will be noticed that the above listing, lists Definitions Paragraph variables so that the primary controlled variables are defined before secondary variables, before valve or output variables. This is a desirable practice putting the most critical controlled variables first. It is also an essential ordering for the Small System language SuperVariable based implementation, associated with the way that the Idiom Loop Statements are interpreted, as explained later. If the variables encountered in a Loop Statement have been configured out of order, the compiler can be instructed to make the minimum sufficient reordering of the definitions, automatically.

Footnotes Paragraph

The Footnotes Paragraph allows individual computational Footnotes to be defined for each of the previously asterisked Footnote references included in the Definitions Paragraph. Each Footnote is defined as a Statement or Bracketed Activity (not illustrated in this example). In the full program listing, each is listed in numerical order with an initial asterisk and Footnote number. The Footnote to be edited is selected by selecting (in the selection field, or by Mouse) the appropriate Key Name. This in turn causes the display of the appropriate Definition with Headings and

[14] Corresponding to Real process I/O Attributes IN/OUT are the Discrete counterparts: DIN/DOUT.

figure shows the T104 Key Name selected as the currently edited variable:

NAME	NAME	IN	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
T101	Temperature_1	ECB1	_	0.0	250.0	°F	_	_	_	10.0
T104	TemperatureAv	_*[1]	_	0.0	250.0	°F	_	_	_	10.0

In the above definition entries, the VALUE, SET, HI, and LO Attributes are initially entered as blank (undefined). Some of these Attributes are entered later by operator action, others (VALUE and SET) by I/O or control computation. The values entered in for the HI, LO, and DEV alarm Attributes would correspond to the Real valued alarm limits. For both variables the scaling range is 0.0 to 250.0. MIN and MAX determine the scaling of not only the variable Value Attribute, but of the associated SET, HI, LO, and DEV Attributes as well. The IN Attribute associated with T104 is left undefined, but with an asterisk (numbered by the system) corresponding to a Footnote defined later.

One can continue similarly with the configuration of the remaining variable definitions in the example. Corresponding to the IN I/O input Attribute is the OUT output Attribute for defining output I/O channels associated with Real valued variables. In addition to the IN and OUT I/O Attributes, processing Real data, there are the DIN and DOUT digital input and output Attributes which return digital or state data, represented as an integer value but interpreted as packed state data interpreted according to the associated STATES Attribute, The example specification and Listing requires that the valve variable V101 to be manipulated (i.e. output). The definition is given below:

NAME	OUT	VALUE	MIN	MAX	UNITS	HI	LO
V101	ECB3	_	0.0	100.0	%	90.0	50.0

The specification and Listing for T102 calls for the outputting to an IA Block. One way of making such a connection from an ICl Block makes a special use of the OUT Attribute. In this case the output represents a connection to the SET parameter in the IA T102 Block. The definition is given below:

NAME	NAME	OUT	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
T102	Temperature_2	T102.SET	_	0.0	250.0	°F	_	_	_	10.0

CONV Attribute conversions represent standard functions including thermocouple conversions, not illustrated here. The definition of the secondary controlled variable F101, below, illustrates the use of the CONV conversion Attribute to apply a square root conversion to the result of an input before storing as a VALUE Attribute. The language supports a separate FILT Attribute, for expressing data smoothing filters. It operates similarly to the CONV Attribute in the SuperVariable definition.

NAME	NAME	IN	CONV	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
F101	OutletFlow1	ECB2	SQRT	_	0.0	250.0	GPM	_	_	_	10.0

Internal variables will generally not require I/O Attributes or scaling:

NAME	VALUE	UNITS
COOKTEMP	_	%

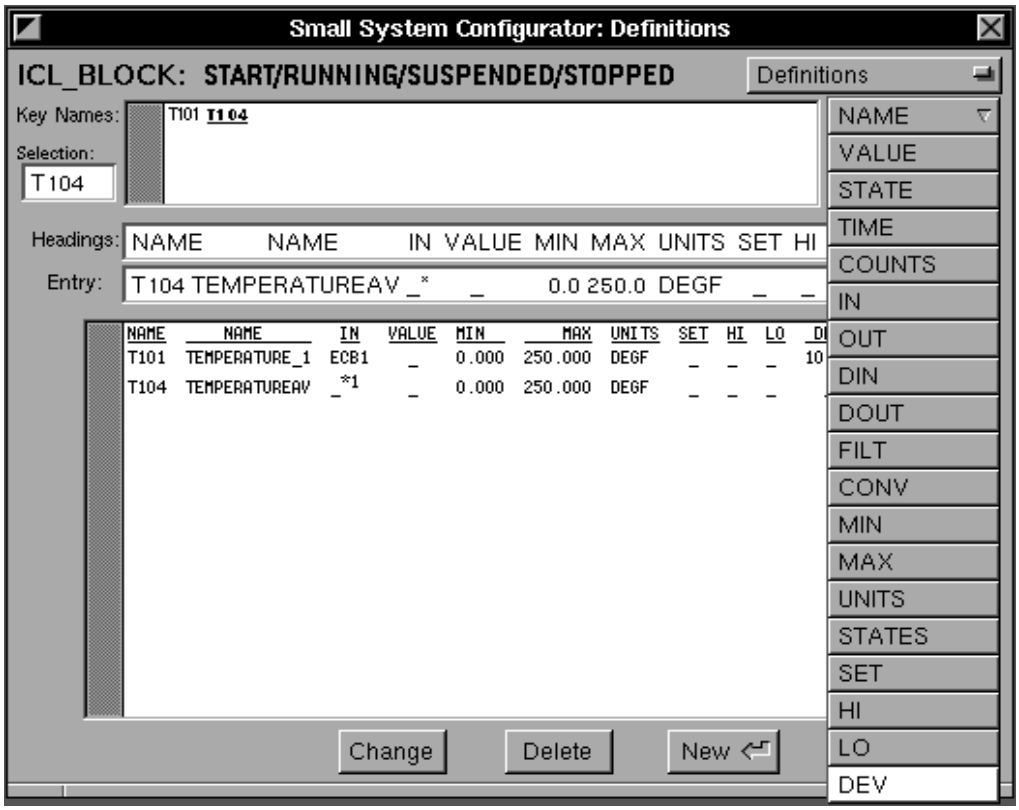
NAME	TIME	UNITS
VENTTIME	10	MIN

The illustrated lower Definition line contains a TIME Attribute, instead of a (Real) VALUE Attribute. ICL supports several kinds of numerical Attributes with their own support. In this case, the Time data is fixed to act as a parameter. But, depending on its associated operational states, the Time data may be automatically changed and used as a clock. The associated Units Attribute, in this case, indicates the actual intended time scale (restricted to SEC/MIN/HOURS/DATE [i.e. absolute date]). In addition to the TIME and VALUE value Attributes the language supports STATE Attributes (representing State or digital values [corresponding to the DIN/DOUT I/O Attributes] and COUNTS Attributes (represented as an integer for counting events or product elements). Like the Time data, the Counts data can be active, used by the system to count and control the program.[12]

ICL digital or State data (as in the STATE Attribute) is represented in packed field form. A particular value is structured against a States Expression (Declaration) containing State names and comma and slash operators (e.g.: **UNFREEZE/FREEZE,UNBOOKED/BOOKED,UNDEFINED/DEFINED/OVERFLOW,START/HOLD/RESTART/RESET,_/EARLY/TIME/LATE**). In these expressions, State names separated by slashes represent alternative State values, and these slashed sub-expressions separated by commas represent separate packed values.[13] In those cases (e.g. with STATE Attributes) where a value has user defined States, the following STATES Attribute will be configured to contain the appropriate States Expression. The STATES Attribute is the digital counterpart of the UNITS and MIN/MAX Attributes. It defines the names of the States and their translation from the I/O internal format to the user display format. The example below shows the Definitions Page usage.

[12] In either case, if the variable is active (in the TIMING or COUNTING State), the current value is compared to a SET Attribute. The result of that comparison is returned as an EARLY/TIME/LATE or UNDER/DONE/OVER State.

[13] The usage is motivated by same normal English usage, which we use in defining alternative states, and sets of mutually exclusive alternative states: AUTO/MANUAL, LOCAL/REMOTE. The Slash has four uses in the ICL (the last two described later): to represent normal division, to represent alternative States as above, to express subscripts in keyboard entered Idiom Loops, and to represent Attributes whose value is to be the same for all Definition entries in a Definition Table.



A new Definition Table is created by first entering the headings into the Headings Text field above, and then entering the desired definitions consecutively in the Entry field. Each definition is terminated by a carriage return (and each Attribute in the definition by a tab or space). The initial definitions of the included example (entered in the above Headings text field) are for the main controlled variable T101 and T104. Their headings are:

NAME NAME IN VALUE MIN MAX UNITS SET HI LO DEV

These definitions each include **two** Name Attributes, a short hand (Key) Name (T101 and T104), and a more definitive Name (Temperature_1 and TemperatureAv). When so defined, the names are completely synonymous, but multiple Names must be grouped together at the beginning of the definition. The IN Attribute defines an associated input I/O channel (ECB1 defined for T101 below) for Real valued data. The VALUE Attribute represents the real value of the associated variable. The data as generated by the hardware channel has a standardized scaling (between 0.0 and 1.0 for ICL) unrelated to the application. It must be rescaled (whether in ICL or IA) to the range of values appropriate for engineering control calculations and numerical operator displays.[8] The Min and Max Attributes correspond to the minimum and maximum scaled range values. Real (analog) valued variables will normally have a corresponding engineering units for operator display, in this case, °F (degrees Fahrenheit). The Units Attribute provides such units. If control is to be applied to the variable, a SET Attribute is needed.[9] HI, LO, and DEV Attributes represent the corresponding alarm limits, when used.

The advantage of user defined Attributes is three-fold:[10] the user tailors his definition precisely to his need; any named Attribute behaves identically in any definition, whatever the remaining definition structure; and he doesn't need to learn someone else's Block formats, with parameter meanings that he doesn't use. The learning of Attribute usage is straightforward. But it can be simplified in two ways: several simple standard formats can be supplied (like the IA I/O Blocks, but without all of their complexity and elaboration. And the Attributes can be selected from a menu, as expanded in the above figure.[11]

Once the headings are provided, the associated variables can be defined in entries under those headings. The above

Names to select a particular, already defined variable for edit, or pre or post insertion of a new variable. If direct Mouse selection of the desired Key Name (or even of the actual Table entry) were directly possible this would be much better.

[8] At the same time preserving the equivalent of 0–100% scaled range for I/O, indicator display, and Idiomatic control.

[9] This usage is unique to setpoints in ICL: the other Attributes will normally have fixed parameter counterparts in the IA process I/O blocks. But IA Setpoints will normally be associated with control blocks. The ICL setpoint is an Attribute of its associated variable, shared between any associated alarm and Idiom calculations.

[10] Apart from the greater efficiency of programs running free of large amounts of redundant parameters and support computations.

[11] There are two special usages associated with Attribute Headings. Such a Heading can occur with an immediately following slash and value: **VALUE/10.0**. This usage means that all of the Attributes in the table are to have the same specified value. In the small system, the Attributes are all duplicated and this usage will only be a short hand for representing the shared usage; it will not (probably?) affect Back Compilation. An Attribute Heading can also be renamed by the user in a particular table by using the user entered Name with the standard Name in parenthesis, as discussed later under Zipper Reference: **HH(HI)**.

The remaining program function is separated into six distinct configuration Paragraph forms, each with its own configuration display:

- Definitions, defining the process variables and any general process I/O, alarm, or control target characteristics. These variable definitions correspond to the normal IA AIN, AOUT, CIN, COUT Blocks, but with a flexibility allowing the user to define his own (SuperVariable) heading formats, associating any number of variables, configured and listed in a single table. The SuperVariable is configured out of Attributes, each corresponding to a name, I/O channel, value, or other basic part of a definition. All control and alarm setpoints are defined, as SET Attributes, in a corresponding variable definition.
- Footnotes, which permit arbitrary computations to be inserted within the variable definitions, and carried out in the corresponding order within the variable processing scan. Each Footnote is configured (and numbered automatically) to correspond to a Footnote reference, already established within the Definitions Paragraph as a terminating asterisk to one of the SuperVariable Attributes.
- Filters/Compensations, similar to the Footnotes Paragraph in allowing the insertion of special computations but supporting dynamic filters and standard compensations. The example does not illustrate these.
- Loops, which permit the configuration of simple and cascaded control loops in terms of the Idioms, which can also include Feedforwards, Constraint Controls, and a number of other usages.
- Theme Statements, which express certain idealized forms of sequencing and coordination, taking the place of Profile or Cut Cam Controllers, Drum Sequencers, etc.
- Procedures, which allow the implementation of more general forms of support or sequencing computation, including the expression of Sequenced, Parallel, Looping, Continuous, or State Driven Activities.[5] The above example shows one example Procedure.

The following sections develop each of the Paragraph forms, discussing them in greater detail to the extent that they are included in the examples. The remainder of the document addresses the specification and implementation of the Language in greater detail. The configuration displays illustrated below envision data being entered in, either from a keyboard or from a menu. Sometimes either can be used as shown.

Definitions Paragraph

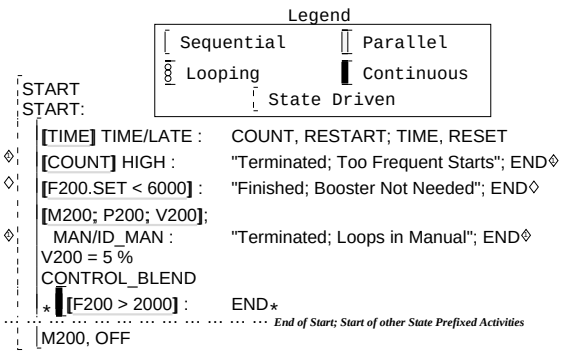
Conventional (IA) systems define process variables in fixed format Blocks, each Block variable definition requiring a full page configuration display. ICL allows grouped display of multiple similar process variables in a table whose user selected headings include all of the Attributes needed by that group, and only those Attributes.[6]

The Headings can mix any combination of individual or repeated Attributes, as needed, generally ordered as follows: Name Attributes, I/O Attributes (IN/OUT/DIN/DOUT), I/O support Attributes (Filters and Conversions), Value Attributes, Scaling or display support Attributes, and Setpoints and Alarm Attributes.

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

The configuration of process variable definitions includes two phases: the configuration of a set of headings; and the configuration of the individual definition entries for the included variable definitions[7]:

[5] As discussed above, illustrated in the example, and summarized in the adjacent Legend, Activities consist of one or more statements (including nested Activities, themselves), listed with a lefthand open bracket as shown, which defines their intended order of execution (and, in the case of Looping, Continuous, and State Driven Activities, conditions of repetition). Except for these repeating Activities, Activities terminate when all their included statements and Activities have terminated. In addition, Activities can include an END command which defines an alternative termination condition, at its point of programming, and expresses (by the position of an associated asterisk or diamond icon) the number of nested Activities affected. Activities are discussed more thoroughly in the Large System ICL Specification.



[6] The ICL Process variable definition format, with associated values, is referred to as a SuperVariable, consisting of a free form list of Attributes each Attribute consisting of its heading and value as shown. In the tables, similar SuperVariable definitions are listed together under a common set of headings. The following discussion shows the separate configuration of table headings and entries. The integration of Attribute type and value, and the Back Compilation of these elements to reconstruct the tables will be discussed later.

[7] Note that the example display separates the text entry fields (for Headers and Definitions) from the output listing display, to accommodate the assumed inability of GUI builders to support Text and Graphics, Entry and Display all simultaneously. When desired, the display can use the Selection field for the Key

Key Names: T101 T102 T104 F101 V101 COOKTEMP BIAS VENTTIME HEATTIME COOKTIME COOLTIME

NAME: ICL_BLOCK

States: START/RUNNING/SUSPENDED/STOPPED

NAME	NAME	IN	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
T101	Temperature_1	ECB1	_	0.0	250.0	°F	_	_	_	10.0
T104	TemperatureAv	_*1	_	0.0	250.0	°F	_	_	_	10.0

NAME	NAME	OUT	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
T102	Temperature_2	T102.SET	_	0.0	250.0	°F	_	_	_	10.0

NAME	NAME	IN	CONV	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
F101	OutletFlow1	ECB2	SQRT	_	0.0	250.0	GPM	_	_	_	10.0

NAME	OUT	VALUE	MIN	MAX	UNITS	HI	LO
V101	ECB3	_	0	100	%	90.0	5.0

NAME	VALUE	UNITS
COOKTEMP	200	°F
BIAS	7	°F

NAME	TIME	UNITS
VENTTIME	10	MIN
HEATTIME	20	MIN
COOKTIME	80	MIN
COOLTIME	15	MIN

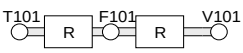
FOOTNOTES:

*1 T104 = (T101 + T102) / 2

FILTERS/COMPENSATIONS:

LOOPS:

T101 REGULATE[1] F101 REGULATE[2] V101



	NAME	STATES	PB	INT	DER
[1]	T101	AUTO, +	100.0 %	0.2 MIN	0 MIN
[2]	F101	AUTO, +	100.0 %	0.1 MIN	0 MIN

THEME STATEMENTS:

```
RAMP T101          (START) FOR VENTTIME;†[1]
                   (HEAT)  TO COOKTEMP + BIAS, IN HEATTIME MIN;
                   (COOK)  AT COOKTEMP, FOR COOKTIME MIN;
  RAMP T102.VALUE  (COOL)  TO 0,*[2] IN COOLTIME†[3]
```

†[1] T102.VALUE = COOKTEMP

*[2] T101 = COOKTEMP * (T102 / COOKTEMP)^2

†[3] STOPPED

PROCEDURES:

```
STOPPED:
  T104 = (2*T101 + T102)/3
  T102.VALUE = -.1
```

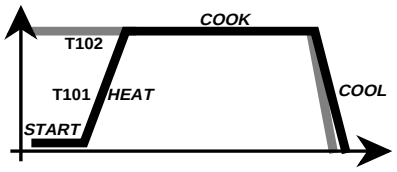
The Small System ICL Paragraphs and Configuration Displays

A Small System Language ICL program is initially configured, as shown, on a Header Display, with a Name (in this case, ICL_BLOCK) and a list of user defined States, as shown in the final listing. These States represent user defined operating modes for the entire program/block. If the program is to be set to a particular State, the corresponding State name can be Mouse selected, and shaded.

(such as a SuperVariable or Theme Statement) in such a way that their statement does not obscure the structured clarity of the higher level computation; also the implementation basis of inserting control computations within the ICL SuperVariable implementation. Footnotes look like their text counterpart, but express active computations.

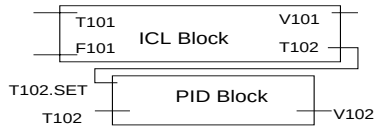
- Idiom; the basic high level ICL method for expressing dynamic compensation and control in terms of the intended control affect in the application: Regulation and Constraint operation.
- Idiom Loop; the basic ICL method for describing controls as loops. It takes the from of an algebraic statement form mixing variable names and Idiom operators, which defines the degree of freedom allocation for controlling some primary variable in terms of secondary (tertiary, etc.) controls and constraint overrides.
- Theme Statement; a high level ICL statement for expressing a particular kind pattern of application related sequencing or coordination in a structured but somewhat open ended manner.
- State; the basic ICL representation form for Discrete data in terms of groups of (English) named values corresponding to the natural processing states and represented internally in packed fields. The use of the State concept allows more natural usage in many ways but requires the development of new computational forms to support it.
- State Prefix; the basic ICL mechanism for making a statement (or Activity) conditional on some Discrete data test. It consists of a restricted (for clarity) State expression which expresses the test, followed a colon, followed by the statement (or Activity).
- Activity; a grouping of statements (or nested Activities), set off by a left hand Bracket icon, whose statements are executed in some standardized order, as indicated by the choice of Bracket shape.

An Example



An ICL IA Block example application assumes the following requirements:

- A cooking vessel is to be heated and cooled, in terms of a temperature T101, to follow the illustrated specified profile, after an initial venting period.
- A related temperature T102 is coordinated to follow its own profile similar to the T101 profile. In fact the final cooling segment of the profile programs the T102 variable with the main T101 being computed to track it.
- T101 is to be controlled by a conventional Temperature/Flow control cascade, within the main ICL control Block, with T101 the primary controlled variable, a secondary controlled Flow variable (F101), and a final manipulated valve (V101).
- T102 is to be separately controlled by an external IA PID Block.



- The application requires the separate computation of the average temperature (of T101 and T102), T104.
- The values defining the profile temperature setpoints and timings are to be configured as user modifiable parameters (COOKTEMP, BIAS, VENTTIME, HEATTIME, COOKTIME, and COOLTIME).
- For the purposes of illustrating the Procedures Paragraph altered values are applied to T104 and T102 when the application is reset to the STOPPED state.

The resulting program is listed below:

NAME	**NEXTSTATE**	**NONE**	**ON**	**OUT**	**OVER**	**OVERFLOW**	**OVERRIDE**
RAMP	**Ratio**	**REGULATE(X,E)**	**Remainder**	**RESET**	**RESTART**	**Result**	**RETRY**
SEC	**SEQUENCE**	**SET**	**SCALE**	**SOME**	**SPLR**	**START**	**STATE**
STATES	**STREAM**	**TIME**	**TO**	**TRUTH_TABLE**	**UNBOOK**	**UNDEFINED**	**UNDER**
UNDERFLOW	**UNFREEZE**	**UNITS**	**VALUE**	**WAIT**	_		

Concepts

Activity	**Attribute**	**Booking**	***Context***	**Footnote**
Freezing	**Idiom**	**List**	**Local**	**State Environment**
Loop Statement	**Named Statements**	**Model/Modeling**	**Prefix: State, Scoping, Naming, Override**	
Recipe/Recipe Call	**Sequential Function Chart**	**State**	**SuperVariable**	**Theme Statement**
User Renaming				

The document will describe each language Paragraph and Statement form, and its translation, illustrating possible configuration displays without imposing a display or GUI specification.[2] The reason for this is that the Small System Language is intended to support several different application product perspectives and GUI platforms, each of which may address the actual configuration differently:

- Single Loop Controller. In its simplest form this would support a single statement display; a more elaborate display (or remote configurator) might support the other configuration attitudes described below. Each separate Paragraph form would be supported by menu selections in the spirit of the 760 controllers, whose choices would allow one to select the successive symbols and operators in the associated statement form. In this framework, each Paragraph would be designed to allow the user an easy memory model in lieu of an actual display; only individual statement and entry lines would actually be displayed.
- Local Equipment Controller. The associated configuration displays are described in the separate SPRITE project Local Equipment Controller ICL Subset Specification documents. In this case, a main configuration menu allows for the selection of the necessary Paragraph configuration displays. Each display supports its own detailed menu selections, as well as the actual Statement or Entry keyboard entry rules corresponding to the Statement or Entry discussions below.
- IA ICL Block. This represents the initial focus of the current effort. The explicit goal is a language block in the spirit of the sequence block, but of a more general control capability. The configuration displays would be based on the Local Equipment Controller discussions. The larger overall program pcode arrangement described below is focussed on this product and based on the LEC design, independent of the configuration displays.

The current document thus focuses on the relation between the actual Statement and Entry forms and their translated pcode. However there are two elements common to any Paragraph display:

- Each Paragraph is labeled by an appropriate Heading, in the listing or configuration display.[3]
- Each display or page in a Listing or configuration Paragraph includes a list of the Key Names. This is a list of user defined Names, which includes one (the first one if there are several) for each user defined variable. Where more than one Name applies to a given variable, they must be grouped together.[4] A Key Name will be generally chosen as the shortest Name for its variable. The Key Name list is thus a reminder for the user, of the spelling of his chosen Names as he programs. The Key Name list also constitutes a menu expressing variable definition choices (indicated below in bold face underlined) as the basis for editing.

Preliminary Definition of Basic Small System ICL Concepts

This section introduces several basic ICL concepts as related to the small system design:

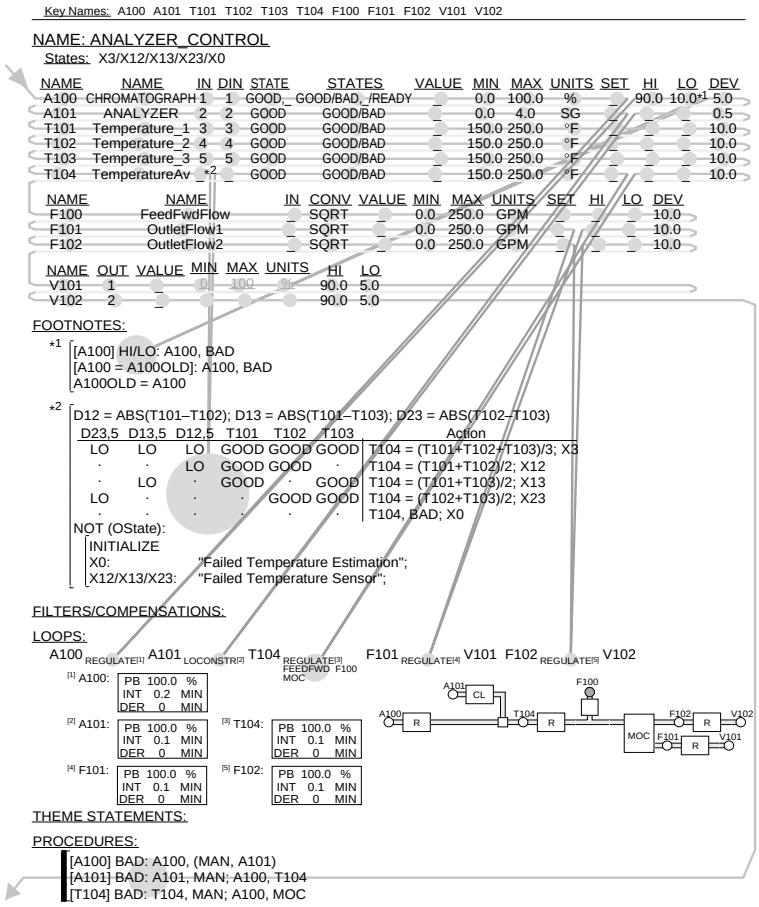
- SuperVariable; the basic free form structure of ICL variables, taken also as the implementation basis of the Small System language. An ICL variable is defined as an open ended list of concatenated Attributes, these Attributes embodying the complete (I/O, scaling, alarming, etc.) functionality of the variable. In the Small System design, the variable definitions are also concatenated into one large SuperVariable which contains all of the Small System control functionality as well.
- Footnote; a basic ICL mechanism for declaring arbitrary support computations for some higher level structured computation

[2] The information and statement forms required within each Paragraph is, of course, specified. The figures below show the information needed in each Paragraph configuration display, however dressed up. There are some assumptions about the capabilities and limitations to current GUI configuring tools, which define the design expectations embodied in the examples:
- The GUI builder supports: Windows and included Buttons, Popup (with persistent selection) and Pulldown (with temporary selection) menus, and Text entry fields.
- It supports Display fields which permit the mixing of Text and Graphics. These fields are assumed distinct from data entry fields because the mixing of text, graphics and text entry is not generally well supported by current Display Objects, unless specially designed. Ideally a single Text Object should cover all. A helpful compromise would support Mouse selection from Text/Graphic Display fields.
- The Text Display fields should support the output in Display PostScript. One Sun available system which does all of this is (will be?) OpenStep.
- For maximum programming ease of use, the end result should permit program entry entirely by the use of normal (not Control) keys. The Mouse should be required strictly for editing and menu selection. In particular, normal programming should minimize the switching back and forth between Keyboard and Mouse. [An exception would be the use of a Mouse menu instead of entering special program function names on a keyboard, when this would facilitate their learning and recall.]

[3] With the exception of the Definitions Paragraph in the listing.

[4] As detailed later, the occurrence of an initially isolated Key Name Attribute in the sequence of SuperVariable pcoded Attributes is the sole internal indication of the start of a new variable. The Back Compilation of definitions depends on this and on the ability to recognized repeating Heading patterns to regenerate the Definition tables. Most other aspects of the Back Compilation are similarly derived from structure implied by such features in the pattern of pcoded elements, which is otherwise executed as a continuous amorphous sequence.

implemented in a generally continuous and parallel executed pcode, executed each sample time without any separate operating system or under a single operating system call.



- Field Device/Single Loop ICL, intended as the basis for small field devices such as single loop controllers. This version is implemented about the same SuperVariable pcode scan as the Small System ICL. It could include a simpler listing format compatible with a simpler (single line) display.

This note describes the Small System Language, generally suggesting the intended structure, the individual language statements and line formats, and their intended pcode translations.[1] The general framework is introduced in terms of one main example and several subordinate ones. The SuperVariable is first introduced with the language examples and then in terms of the Zipper Reference strategy. More detailed elements of the language not covered in the example are then separately developed. The note summarizes key implementation strategies as a matter of illuminating the essential meaning of the associated language elements, and bridging to deeper design documents.

The section on meaning and translation further discusses implementation while completing the detailed specification of some language elements. The final section summarizes ICL symbol and naming rules. Except for the high level Paragraph (vs. Operations and Pages) structure, most of the detailed syntax and semantics will be derived from the Large System Language, compatible with the "Large System ICL; Rough Specification" document. In particularly detailed definitions of the following elements may be found there:

Common ICL Keywords and Symbols

(	**)**	**..**	**:**	**::**	**;**	**<>**	**ABORT**
ACTIVE	**ADVANCE**	**ALL**	**ANY**	**ANY_NOT**	**AT**	**BACKUP**	**BLEND**
BOOK	**CC**	**CONTINUE**	**CONTINUED**	**CONV**	**COUNTS**	**DATE**	**DeadTime**
DEV	**DIN**	**DONE**	**DOUT**	**EARLY**	**ELSE**	**END**	**EXECUTE**
FAIL	**FEEDFWD**	**FILT**	**FOR**	**FREEZE**	**HI**	**HICONSTR**	**HIGH**
HILIMIT	**HOLD**	**HOURS**	**IN**	**NACTIVE**	**INITIAL**	**LATE**	**LeadLag**
LO	**LOCONSTR**	**LOLIMIT**	**LOW**	**MAX**	**MESSAGE**	**MIN**	**MOC**

[1] The listing is defined in terms of Paragraphs, each made up of a simple heading and successive statements and lines. But the program is entered in terms of separate displays for each Paragraph. The resulting program is then translated to (already implemented) pcodes, described in greater detail in other documents. It is usually considered inappropriate to combine a language (syntax) spec with implementation discussion. But the simplifications of the Small Language are derived from the implementation opportunities. Because these are reflected in the semantics of the Small Language, the combination of the two perspectives sheds light on both. The implementation discussion also sets a background for parts of the design which are not otherwise documented. These discussions are not considered an alternative to the detailed design documents.

Small System ICL; Rough Language and Implementation Spec

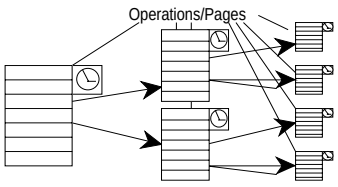
This document develops a rough specification for the Small System ICL, based on the SuperVariable concept as implemented in a continuous running pcode. The discussion addresses both language syntax and the elements of the intended implementation on the grounds that the two are highly related within the Small System concept; implementation is thus central to the defined meaning of the language.

Contents

Introduction

ICL is intended not only as a general purpose control language, but as a basis for control languages of several different application scales, and for distributed control systems made of separate functions operating separately on these different scales. This objective is supported by three different intended versions of ICL:

- Large System ICL, intended as a general purpose, single program, single language, control configuration vehicle for CP scale controllers. This version is programmed and structured in terms of a hierarchy of Operations, SubOperations, and Task Calls, and in terms of a division of functions into standard format, standard function Pages. The large system language is implemented conventionally in terms of a procedural hierarchy of Operations. Each Operation is generally implemented in a sequentially pcoded representation of its different Pages, each run individually, when activated, on executive or operating system queues.



- Small System ICL, intended as the basis for small system controllers (the Local Equipment Controller) or as the basis for CP style language blocks (referred to as an ICL IA Block). For this version, the single controller or block program is expressed as a single level listing, divided into standard format, standard function Paragraphs (analogous to the Pages). The Small System Language is implemented about the process I/O scan expressed in terms of a low level pcoded representation (referred to as a SuperVariable scan). The figure below illustrates the relation between the resulting Language as listed and as implemented. In this case, the entire program is