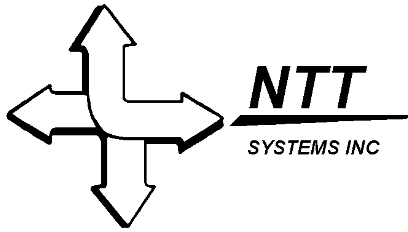


The Husky Hot Runner Nozzle Calculators

Theory of Operations

NTT SYSTEMS INC.,
86 Dunblaine Ave.,
North York, Ontario
CANADA
M3M 2S1
(416)486-3565

http://www.NTT.ca

19981214

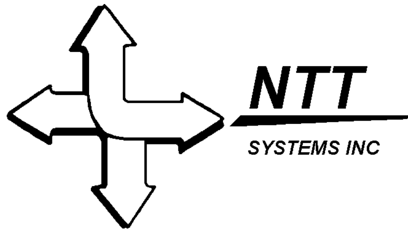


Table of Contents

1. Overview **1**

2. Software Architecture **1**

2.1 File Naming Conventions 2

3. Form Structure and Management **3**

3.1 Dependencies 3

3.2 JavaScript Form Management 4

3.2.1 Function Descriptions 5

3.2.2 Interfaces to Other Modules 7

4. Calculator Structure **8**

4.1 Common Components 8

4.1.1 Engineering Variables 8

4.1.2 Creating Engineering Variables 9

4.1.3 Using Engineering Variables 11

4.1.4 Handling L and BLR 11

4.1.5 Handling NDrops 11

4.1.6 Handling Pitch 12

4.1.7 Handling Temperature 12

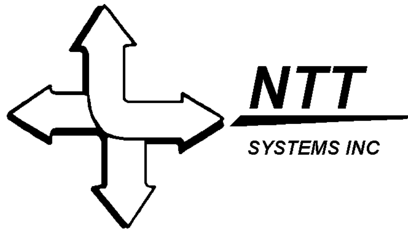
4.2 calccom.js 12

4.3 The Calculator's Structure 14

5. 750 Nozzle Calculators **15**

5.1 Series Logic 15

5.2 Valve Nozzles 15



5.2.1 Valve Nozzle Types 15

5.3 Thermal Nozzles 16

5.3.1 Thermal Nozzle Types 16

5.4 Some Review 16

6. 250 Nozzle Calculators 17

6.1 Expansion and Deflection Modeling 17

6.1.1 Data Structures 18

6.1.2 Modeling Algorithms 18

7. 1250 Nozzle Calculators 19

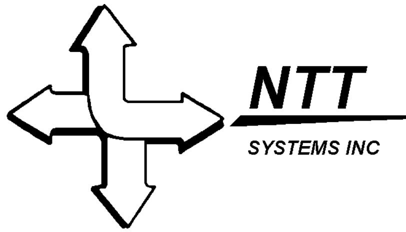
7.1 Series Logic 19

7.2 Valve Nozzles 19

7.2.1 Valve Nozzle Types 20

7.3 Thermal Nozzles 20

7.3.1 Thermal Nozzle Types 20



ReadMe.1st

This manual describes a set of nozzle calculators developed by *NTT SYSTEMS INC.* for HUSKY. They are intended to exist within HTML documents, one for each type of nozzle. The HTML was developed by Passage Prepress Inc.

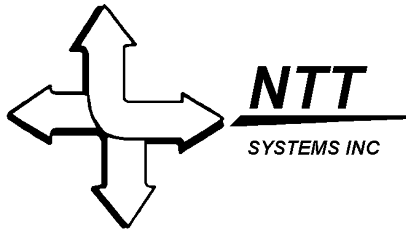
The purpose of the manual is to describe how the system works to those who will modify and enhance it. There are two types of work that may need to be done. The first involves changes to HTML documents or to calculators in areas that impact how data is received from or sent to users. Sections 1, 2, and 3 will be of most interest to people doing these tasks. The second involves making changes to calculator internals or creating new calculators. I'm afraid that people doing this will have to read the entire document in detail.

The implementation makes extensive use of JavaScript's object creation and management features. I have tried to describe how and why we used these features, but this is not intended to be a JavaScript tutorial. I recommend "JavaScript Bible" by Danny Goodman (IDG Books Worldwide, Inc., ISBN 0-7645-3188-3) for this purpose. A note on typographic conventions… We use Courier New for plain JavaScript `functions`, `variables and module names`, and Arial for objects, object attributes and methods.

While we are proud of the design that we created, it does have its warts and we tried to identify these. Hopefully they can be addressed in future releases.

This document is also not intended to be a nozzle design manual. All nozzle design specifications were provided by Greg Cheshire (GCheshir@Husky.on.ca). These are not included, but they are mandatory reading for anyone who wants to modify or understand the calculations. Our thanks to Greg, who answered our dumb questions and validated the final deliverables.

19981214



1. Overview

The HUSKY Design Centre includes a set of hot runner nozzle dimension calculators. The complete calculator system consists of a series of HTML pages, developed by Passage Prepress Inc., that invoke JavaScript routines developed by *NTT SYSTEMS INC*. This document describes the structure and operation of the JavaScript software and the relationship between it and the HTML components.

All technical information relating to the nozzles was assembled and written by Husky's Greg Cheshire. Our code implements Greg's specifications in as straight forward a manner as possible. Reading and understanding his documentation, especially relating to expansion and deflection modeling for 250 nozzles, is a critical prerequisite to understanding the details of the calculators.

Prior to the creation of the CD-ROM, customers were able to determine various critical dimensions by using tables and formulae provided by Husky. These were based on a spreadsheet used internally by hot runner design engineers. The current calculator is more sophisticated than the tables and formulae but is still not as flexible as the internal spreadsheet. This was done intentionally so customers could quickly get the information they need so long as their requirements fell within a well-defined set of constraints. A design goal of the customer calculator was to ensure that no engineering outputs are produced if customer specifications fell outside these limits. The system is also designed so that all outputs are cleared immediately when any input field is modified. This prevents confusion that would arise if a user changed the inputs after a calculation and then printed the page.

The system is developed using object-oriented techniques. This provides very tight interfaces and lays a firm foundation for enhancement. In particular, it could be extended to provide the functionality of the internal spreadsheet. By doing this, both customers and engineers could have calculators customized from the same code base and neither would have to rely on platform specific tools such as Excel.

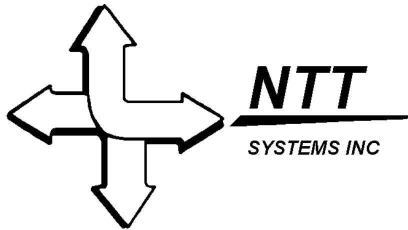
2. Software Architecture

The JavaScript code is modular and comprises about 2,200 lines in 28 modules. One of these handles all interactions between the HTML document and its calculator[1], and another contains logic that is common to all calculators. The remaining modules contain code unique to the related nozzle.

A given calculator belongs to a *series*, *group*, and *type*. For example, a 750 VG nozzle is in the *750* series, the *valve* group, and the *VG* type. There are corresponding JavaScript modules for each of these levels. That is, there is one module that implements things common to all 750 nozzles, one that handles things common to all 750 Valve modules (*i.e.,* VG and VX), and one that handles things specific to 750 VG modules (both L-Dimension and BL-Dimension).

---

[1] That isn't quite true. There are a few places where the HTML code calls a routine below the top module, and a few where the calculator knows about what kind of HTML controls are being used. These are worth fixing, but aren't much worse than embarrassing. We'll try to point out the offenders.



The effect of this is that each document consists of the HTML needed to create the interface and references to five JavaScript modules. The first two of these are the same in all cases, the third is specific to the nozzle *series*, the fourth to the *group* for that series, and the fifth to the *type* for that group. When a document is loaded, the last three modules work together to "construct" a calculator specific to the nozzle displayed in the interface.

2.1 File Naming Conventions

The user interface was developed by Passage, and the calculators were developed by *NTT*. In order to reduce the need for communication and to minimize the possibility of error, a number of conventions were established. One related to naming JavaScript modules.

The two modules that are always required are `formcom.js` and `calccom.js`.

The module required for all nozzles in a given series is named *`Series`*`calc.js`.

The module required for a group of nozzles in a series is named *`SeriesGroup`*`.js`

The module required for a specific nozzle is named *`SeriesType`*`c.js`.[2]

`Series` is "s1" for 250's, "s2" for 750's, and "s3" for 1250's.

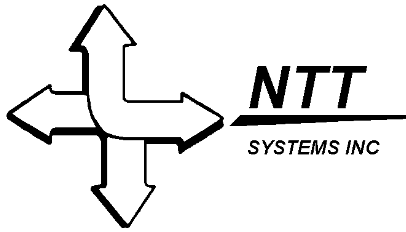
`Group` is "vgvx" for valve nozzles and "other" for thermal nozzles for 750's and 1250's. There is only one group for 250's and this is denoted as "all".

`Type` is nozzle specific and we used the normal Husky designation with hyphens (if any) removed.

The following table shows examples of how a nozzle name is used to determine the set of modules needed to build the corresponding calculator:

Husky Nozzle Name	Series Module	Group Module	Type Module
250 HT-X	`s1calc.js`	`s1all.js`	`s1htxc.js`
250 HT-T	`s1calc.js`	`s1all.js`	`s1httc.js`
750 TS	`s2calc.js`	`s2other.js`	`s2tsc.js`
750 VG	`s2calc.js`	`s2vgvx.js`	`s2vgc.js`
1250 VX	`s3calc.js`	`s3vgvx.js`	`s3vxc.js`
1250 HT-DC	`s3calc.js`	`s3other.js`	`s3htdcc.js`

[2] Note the "c" following the *type* designation. This is a nuisance, but it got in early and we had too much done to change it. Sorry.



3. Form Structure and Management

3.1 Dependencies

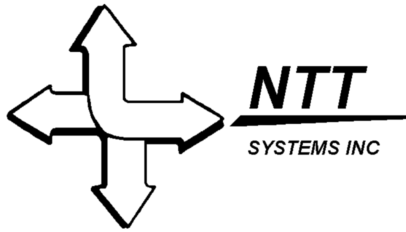
If you are not interested in how the HTML and JavaScript work together to handle the user interface, skip to Section 4.

The JavaScript and HTML were designed to have as narrow an interface as possible. The dependencies are described in the following sections. **Note that all names are case sensitive.**

From our perspective the document has an input section, an output section and some layers that contain error graphics. In addition to these, there are a number of other graphics and decoration that are of no concern to the code. All input controls must reside in a single form. The output fields may be scattered in any number of other layers. The key to the interaction between the document and the calculator is a small set of naming conventions. The primary one is that controls that are to contain calculator inputs and outputs must have exactly the same names as the corresponding engineering variables.

The JavaScript code assumes that the HTML will conform to the following rules:

1. The form's <BODY…> tag must contain the call "onLoad= initForm()". This allows the code to perform its initialization logic.
2. All user input controls must be either <INPUT TYPE= "text"…> or <SELECT-ONE …> and must be located in a form named "inForm" in a layer named "inFormLayer".
3. All other controls (limit boxes and outputs) must be <INPUT TYPE="text" …>. Each may reside in its own form or there may be multiple controls in multiple forms. The names of the forms are irrelevant. Each of these forms must reside in its own layer. The names of the layers are irrelevant, except that none can be named "inFormLayer".
4. The names of the input and output controls must match exactly the names of the engineering variables. The same name may be used for both input and output controls. This could be done to copy an input to the output area. However, the name may be used only once for an input control and only once for an output control.
5. Limit controls (eg., upper and lower limits for the L dimension) must be named "lb_*variableName*" and "ub_*variableName*". For example, the boxes that will hold the lower and upper limits for the L dimension are "lb_L" and "ub_L".
6. All input controls of TYPE="text" must contain the text "onBlur="setCalcVal (this.name, this.value, this)" onFocus="clearOut()"". The onBlur handler transfers user input to the calculator. The onFocus handler is used to clear all calculator outputs the first time the user touches an input control following a calculation. As a result, the user can not produce a printout with a new set of inputs and calculator output from a previous configuration.
7. All SELECT controls used for input must contain the text "onChange="setCalcVal (this.name, this.options[selectedIndex].text, this)"".



8. The HTML mechanism used to trigger a calculation must call the JavaScript routine "doCalculation()".
9. Error graphics may be associated with input controls; for example, to indicate that a user input is out of range. In this case, the graphic must be contained in a layer named *variableName*Error, eg., LError. The graphic must be hidden 1000 pixles to the left of the location it should be displayed at if an error occurs. The calculator will shift it into view as required and back out of view when the next calculation is performed.
10. There must be a frame named hiddenFrame that contains a form named hiddenForm that contains a text box named calcUnit. hiddenFrame must be a sibling of the frame that contains the nozzle calculator. The text box, calcUnit, must contain the text "imperial," "metric," or "husky." The case of the text is not significant.
11. There must be a TEXT box in some output layer named revNum. This will be used to display the software revision number.[3]
12. The location of the nozzle graphic should be indicated with an anchor named #output. The browser window is moved to this location after a successful calculation.

3.2 JavaScript Form Management

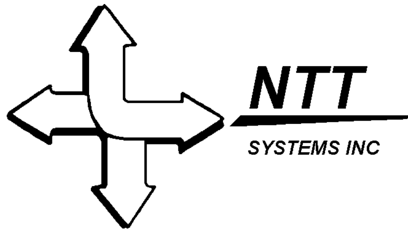
`formcom.js` contains a set of routines that act as a buffer between the HTML document and the calculator itself. They are responsible for moving data between the calculator and the document and for starting calculations once validation has been performed. Section 3.1 defines what `formcom` routines assume about the document. They also have some knowledge of the structure of the calculator. They know that its name is nozcalc and that they must call `initTypeCalc` during initialization. They know how to enumerate all of the engineering variables and how to initiate a calculation.

`initForm` is called when the document has been fully loaded by the browser. Its job is to find all the controls on the form and separate them into groups, one for inputs, one for calculator outputs and one for upper and lower bounds fields. Once this has been done, all of these controls can be accessed without regard to the browser and document models. It then initializes the calculator itself, clears the inputs and outputs and initializes the limit fields. The calculator is now ready to be used.

As the user enters data, the calculator may decide that a new value entered by a user in one field effects other display fields. Currently this only happens with pitch and temperature. In these cases, the calculator uses routines in `formcom.js` to signal the change. These routines are then responsible for taking the appropriate display actions. As a result, the calculator is able to know nothing about the structure and mechanics of the display system.

When the user finally hits the "calculate" button, the `doCalculation` routine interacts with the calculator to validate inputs. Any error is reflected to the user and the process stops. If validation

[3] Each module contains a revision number variable. The "software revision number" is a concatenation of the revision numbers of each of the five JavaScript modules contained in the HTML document.



succeeds, the calculator is called to do its job. When it completes, the form logic takes over again and fills output fields with the results of the calculation.

3.2.1 Function Descriptions

All JavaScript code that interacts with the form is localized in the module `formcom.js`. It contains the following routines:

3.2.1.1 Initialization

`function initForm ()`

The calcUnit text is located and a check is made to ensure that the units are valid (husky units are °F and mm). The browser is checked to ensure that it is at lease Netscape 4.x or IE 4.x. `addFormControls` is called to locate and save the references for the input controls and `addLayerControls` is called to locate and save the references for the output controls. `initTypeCalc` is called to initialize the calculator. Note that `formcom.js` assumes that this function is defined in some other module, which it is. See Section 4. `doOutput` is called to clear all of the input and output fields and `setLimits` is called to fill in the upper and lower bounds boxes in the interface.

`function addFormControls (theForm, theList)`

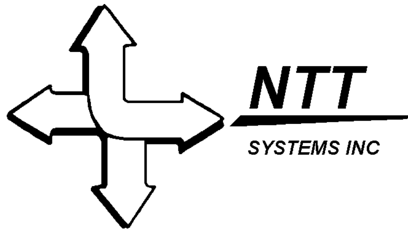
`theForm` is searched for TEXT and SELECT-ONE controls. References to these are saved in `theList`. An exception is made for limit boxes (TEXT boxes with names beginning with ub_ or lb_). References to these are saved in the `limitControls` list. The result is that `theList` will contain a list of all references to input controls and `limitControls` will contain all references to limit display boxes.

`function addLayerControls (theDocument, theList)`

This routine is functionally similar to `addFormControls`, above, except that it finds TEXT and SELECT-ONE controls in all layers other than `inFormLayer` (i.e., the layer that contains the form that contains the user input controls). It uses browser dependent logic to handle document model differences between Netscape and IE. Once these two routines have been called, all controls are in the `inputControls`, `outputControls`, or `limitControls` lists and can be accessed and operated on directly, with no further regard to browser differences.

`function findElement (controlSet, elementName)`

Elements of JavaScript arrays can be stored and retrieved by name or by index number. The above routines created the lists by object name and the arrays can be considered to be hash tables. This routine simply extracts the object stored in `controlSet` using `elementName` as a key. Using an unknown key will result in `NULL` being returned. This routine is a simple wrapper on some JavaScript syntax and we did it this way in case there was a reason to change the underlying data structure.



3.2.1.2 Control and Logic

`function doCalculation (doJump)`

This module is called as the result of user interaction with some control on the form (typically a button that says "Calculate"). `DoOutput` is called to clear all output fields and `hideError` is called to slide any error graphics that may be visible out of the visible space. If `validateInputs` indicates that all is well, the calculator's calculate method is called. `doOutput` is used to display the results. If `doJump` is not specified, or is specified and is TRUE, the browser is told to move the display area to an anchor named #output. This effectively slides the window from the input to the output portion of the document. The revision field is refreshed in all cases.

`function validateInputs ()`

This routine extracts the validationList array from the calculator (see Section 4.) and calls each object in the list at its validate method. If any object indicates a failure, `showError` is called, and the focus is placed on the input control that contains the invalid input. Remember, the name of the input control is the same as the name of the calculator variable so it is easy to move from one to the other.

`function isBLCalculator(blIndicatorControl)`

A given calculator can handle BL and ML/L calculations. This routine is called to see if the HTML document that contains the code is for a BL orML/L calculation. The routine checks to see if there is a control with the name passed as the `blIndicator` parameter. If so, it returns TRUE. The assumption is that no one form would contain both an L and BLR input control.

`function isMetricUser ()`

Returns TRUE if `calcUnit` is "metric," otherwise FALSE.

`function isImperialUser ()`

Returns TRUE if `calcUnit` is "imperial," otherwise FALSE

`function isHuskyUser ()`

Returns TRUE if `calcUnit` is "husky," otherwise FALSE

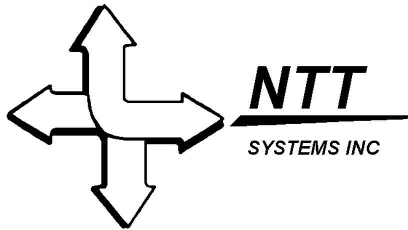
3.2.1.3 Display

`function showError (inputName)`

This routine attempts to locate a layer named *inputName*`Error`. If one can be found, it becomes the `errorObject`, and is shifted 1000 pixels to the right. If Passage set things up right, an error graphic will be moved to the neighbourood of the input box that contains the erroneous value. If a suitably named layer can not be found, `inputName` is used as part of the error message text of a standard alert box. This routine contains browser dependent code.

`function hideError ()`

If there is a current `errorObject`, it is shifted out of view, 1000 pixels to the left.



`function clearOut()`
The onFocus event handler of all input controls should call this routine. If a calculation has just been performed `doOutput` will be called to clear all output boxes. This ensures that the user will never see output that does not correspond to the contents of the input fields.

`function doOutput (controlList, setVal)`
If `setVal` is FALSE, all of the controls in `controlList` are cleared. This can be used for both input and output fields. If `setVal` is TRUE, it is assumed that we want to transfer calculator values to output boxes. In this case, control names are used as calculator variable names and values are retrieved by calling the `getCalcVal` routine in `calccom`.js. They are then set in the output fields.

`function setLimits ()`
This routine is called to set or refresh the upper and lower limit boxes associated with some of the calculator inputs. It does special processing the first time it is called as follows. Variable names are extracted from the calculator and a check is made to see if controls named lb_*variableName*, and/or ub_*variableName* exits in the `limitControls` list. If so, the variable and its associated limit controls are remembered in `limitObjects`. For all invocations, the `limitObjects` list is enumerated and for each entry the variable's lower bound and upper bound values are transferred to the controls and displayed to the user.

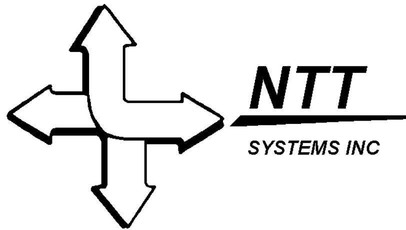
`function pitchLimitsChanged ()`
This is a notification routine called by the calculator when it changes the values of its pitch limits. These limits have initial values, but they may change as the user enters specific values for X, Y, and Z. In this case, the calculator will change the upper and lower bounds in the pitch variables and call this routine. It in turn simply calls `setLimits`.

`function deltaTChanged ()`
This is analagous to the above routine, but for the ΔT variable. ΔT is changed implicitly when the user specifies or changes plate or manifold temperature settings.

3.2.2 Interfaces to Other Modules

The following routines use variables, functions and methods in the calculator modules: `initForm`, `doCalculation`, `validateInputs`, `setLimits`, and `doOutput`. In addition, these routines know nozcalc, the name of the calculator object itself.

Functions and methods in other modules use the following routines: `deltaTChanged`, `pitchLimitsChanged`, `isHuskyUser`, `isImperialUser`, `isMetricUser`, and `isBLCalculator`. These routines let the calculator get information that is stored in the HTML document and modify the output without having to understand anything about how the interface is structured.



4. Calculator Structure

The remainder of this document will be primarily of interest to those who want to create or modify nozzle calculators.

All calculators are built from the same design rules. If you've seen one, you've pretty much seen them all. We relied very heavily of JavaScript's implementation of object oriented technology. Each calculator is constructed as an object and each engineering variable is a separate object. The rational behind this is discussed in Sections 4.1.1 and 4.3.

`calccom.js` provides utility functions that are used by all calculators. A specific calculator is built from three additional modules:

1. *series*calc.js (eg., s2calc.js) contains the calculator object and common code for a particular series of calculators (eg., 750's).
2. *seriesGroup*`.js` (eg., s2vgvx.js) contains common code for a group of nozzles within a series (eg., 750 VALVE nozzles).
3. *SeriesType*`c.js` (eg., s2vgc.js) contains code specific to a specific type of nozzles within a series (eg., 750 valve (VG) nozzles).

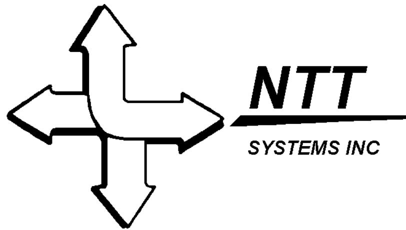
Each of these contains an initialized variable at the start that indicates what nozzle they belong to. This could have been used at construction time to confirm that the modules in a document are compatible, eg., that they are all part of a 750 valve gate nozzle. This was not done, but it would be a good enhancement. Currently, it is possible to create an HTML document containing mismatched parts that might run and produce a result.

Each module contains a function that is used as part of the construction process. `InitTypeCalc` is called once the browser has loaded the document. It calls `initGroupCalc` and then does whatever special initialization it requires. `InitGroupCalc` calls `initSeriesCalc` and then does its own initialization. InitSeriesCalc calls `initCommon` (`calccom.js`) then does its initialization and returns. This is a standard construction mechanism that models the idea of base and derived classes.

4.1 Common Components

4.1.1 Engineering Variables

Calculator variables (also called engineering variables) are encapsulated as objects. This was initially done to handle the problem of precision. The difficulty was that not only did different variables have to be displayed with different precision, but also a given variable had different precision in metric and imperial measure. Carrying the value around in a package (i.e., object) with its precision seemed a good idea. We expanded this to include methods for conversion, validation, and special input handling. Note



that the calculator uses "Husky units" internally. This means that unit conversion is required for all non-Husky users since °F must be converted to °C for metric users and mm must be converted to in. for imperial users.

An engineering variable object contains the following attributes and methods:

name	The text representation of the name, eg., "L," "P," "DeltaT"
value	The current value associated with the variable
precision	The number of decimal digits to round to on input and output
inConv	A method that converts from user units to Husky units
outConv	A method that converts from Husky units to user units
fuzz	A value to allow for the impact of precision differences between user and Husky units.
lowerBound	The value of the lower bound of allowable user input. May be NULL.
upperBound	The value of the upper bound of allowable user input. May be NULL.
doSpecial	A method to be called for special handling of user input. Normally set to a function that simply returns.
validate	A method to be called to validate the current value of the variable. The default function validates for a value between lower and upper bounds (if any).
outOfBoundsError	Always set to `callHuskyError`.

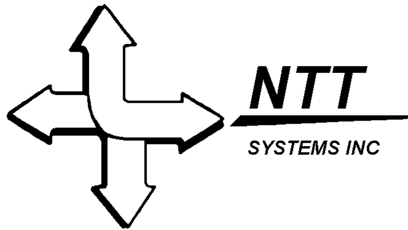
4.1.2 Creating Engineering Variables

An engineering variable is created by a call to `varObject`. The syntax is:

```
obj = varObject (varName, varType, initialValue, mPrec, iPrec, hPrec)
```

where …

varName	A string containing the name of the variable.
varType	"Internal" – for variables that will not appear as user input or output. "Distance" – for linear variables "Temperature" – for temperature variables "DeltaT" – for temperature differences "Unitless" – for all others. Equivalent to "Internal"
initialValue	Initial value. May be NULL. Not validated.
mPrec	Precision if user is metric.
iPrec	precision if user is imperial.
hPrec	Precision if user is Husky.



"Internal" variables ignore precision parameters.

All calculators use the same base set of variables, so we made a separate constructor for each[4]. They take the form:

```
function newvarnameVar (varName, initialValue)
```

eg., `myVar = newLManVar ("Lman")`

The `newLManVar` constructor in turn calls newObj as follows:

```
var = new varObject (varName, "Distance", initialValue, 1, 2, 1)
```

This approach ensures that all calculators use and display manifold length information (for example) consistently. Perhaps more important is that we use these constructors to customize the objects that they create. The complete code for the `newLManVar` constructor is as follows:

```
function newLManVar (varName, initialValue) {
    return new varObject (varName, "Distance", initialValue, 1, 2, 1)
}
```

Its only "value added" is that it ensures that the variable is created with the proper conversion routines and precision. Creating a variable that handles ΔT is slightly more complex since we want to set the upper and lower bounds at construction time.

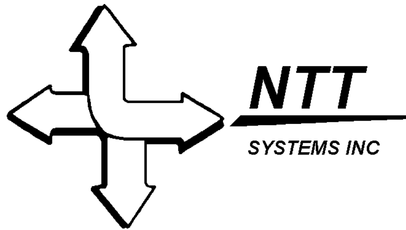
```
function newDeltaTVar (varName, initialValue) {
    var obj = new varObject (varName, "DeltaT", initialValue, 0, 0, 0)
    obj.lowerBound = 200
    obj.upperBound = 600
    return obj
}
```

Creating a variable that implements all of the rules for handling pitch is more complex as can be seen in the following:

```
function newPitchVar (varName, initialValue) {
    var obj = new varObject (varName, "Distance", initialValue, 1,2,1)
    obj.lbExpression = null
    obj.ubExpression = null
    obj.doSpecial = specialPitchHandler
```

[4] We may have overused this a little when we created individual constructors for many of the internal and unitless variables. These could have been created directly in the calculator, for example:

```
P = new varObject ("P", "Unitless", 6, 6, 6)
```



```
    obj.validate = pitchValidation
    return obj
}
```

In this case we override the default methods for doSpecial and validate that were set up in varObject. This is because it has different input and validation rules. What is unusual is that pitch variables have two methods that no other engineering variable has; lbExpression and ubExpression. These will be discussed later in Section 4.1.6. The point to note here is that the structure of JavaScript objects can be modified after they are created, and here we use that feature to give pitch variables extra capabilities. In a sense, pitch variables are derived from engineering variables.

4.1.3 Using Engineering Variables

The user sets the value of an engineering variable by entering information in an HTML input field. By convention (see Section 3.1), the browser will invoke the `setCalcVal` function when the user leaves the field. This function converts the string to a value and stores it in the associated variable. The process is described in detail in Section 4.2.

The string representation of an engineering variable is retrieved via the `getCalcVal` function, also described in Section 4.2.

The value of an engineering variable is set and retrieved using standard JavaScript syntax, i.e., *objectReference*.value = 27, or someVar = *objectReference*.value Section 4.3 describes the structure of the calculator object and how to access the engineering variables that it contains.

4.1.4 Handling L and BLR

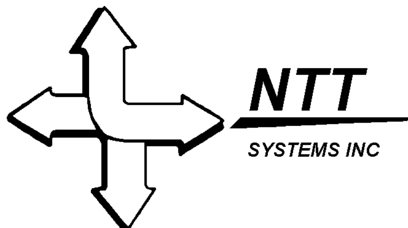
The value that the user enters for L determines the value of LNoz. The L variable's doSpecial handler performs this mapping. It does this by using the LRangeMapper that is created by each calculator. It is an instance of a mapper (described in Section 4.2). A similar approach is used for BLR (BL Range), except that in this case, the input control is a drop down box and the LNoz value is extracted from the input control itself. It was placed there during calculator initialization by a call to `initSelect`[5].

Note that there is no special handling required for L in 1250 nozzles. The 1250 calculator cancels special processing by setting *LVariable*.doSpecial to defaultDoSpecial.

4.1.5 Handling NDrops

NDrops, manifold length (LMan) and Pitch are closely related. The calculator initializes the NDrops selector in the form via the `initSelect` function. It passes the function an array specifying pairs of values that define the mapping from *numberOfDrops* to LMan. The array is created somewhere in the calculator (usually at the *series* level). An example follows:

[5] This is a place where the calculator knows something about the structure of the document. This is not necessary and a replacement routine should be written in `formcom`.




```
var NDropsSelections = new Array (2,40., 4,40., 6,60., 8,60.)
```

When the user selects a NDrops value, specialNDropsHandler is called. This routine determines whether Y and Z pitch values are allowed in addition to X, and sets and displays limits for each. This is done using the functions `calcPitchLimits` and `PitchLimitsChanged`. Finally, the corresponding LMan value is determined and stored.

4.1.6 Handling Pitch

NDrops determines whether Y and Z pitch values are allowed in addition to X. Once this has been done, upper and lower bounds values have to be evaluated. An array of tripplets defines the relationship between drops, lower bound and upper bound. The arrays are usually defined at the *group* level of the calculator. An example follows:

```
var XPitchLimits = new Array (2,133.3,900, 4,82.6,900, 6,59.0,450,  8,59.0,300)
var YPitchLimits = new Array (            4,82.6,900, 6,133.3,900, 8,133.3,900)
var ZPitchLimits = new Array (                                     8,82.6,"900-2*pitchX")
```

Note that no values are defined for Y for the two drop configuration, and Z may only be specified for an eight drop plate.

Note the expression `"900-2*pitchX"` in the `ZPitchLimits` array. All pitch limits are reevaluated at run-time whenever NDrops, X, Y, or Z is changed. Expressions like 82.6 simply evaluate to themselves; we know what the result will be when we create the source code for the calculator. This is not true for Z, which varies based on the current value of `pitchX`. The naming convention is critical. The pitch limits arrays must be named as above. The current, user specified, values can be referenced in the expressions as `pitchX`, `pitchY`, and `pitchZ`.

The limit expressions can contain any valid JavaScript statement. The most complex one currently is in the 250 calculator: `"900-2*((pitchX>pitchY)?pitchX:pitchY)"`. It sets Z upper bounds based on the larger of the current X and Y values.

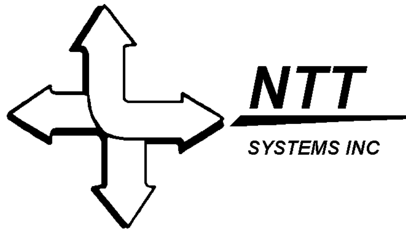
Each time the user enters or changes a Pitch value, specialPitchHandler is called to reevaluate and update all pitch limits.

4.1.7 Handling Temperature

Temperature requires special attention. The user specifies manifold and plate temperatures (TMan and TP) but the dependent variable, DeltaT has the associated bounds. Each time TMan or TP is changed, specialTManTPHandler is called. It sets the DeltaT value and calls `deltaTChanged` in `formcom` so that the display is updated.

4.2 calccom.js

This section identifies and describes the common calculator routines. `calccom` contains several constructors with names of the form `new`*VariableName*`Var`. The purpose and structure of these routines



was discussed above in Section 4.1.2 and they are not included here. Similarly, special input and validation handlers are not included.

`function formatNumber (expr, decPlaces)`
Returns the string representation of the `expr` rounded to `decPlaces`. Will handle positive and negative fixed and floating point expressions.

`function F2C (temp)`
Fahrenheit to centigrade conversion. Returns NULL for NULL or non-numeric inputs.

`function C2F (temp)`
Centigrade to Fahrenheit conversion. Returns NULL for NULL or non-numeric inputs.

`function deltaF2C (temp)`
Fahrenheit to centigrade conversion. Returns NULL for NULL or non-numeric inputs.

`function deltaC2F (temp)`
Centigrade to Fahrenheit conversion. Returns NULL for NULL or non-numeric inputs.

`function mm2i (dist)`
Millimeter to inch conversion. Based on 2.54

`function i2mm (dist)`
Inch to millimeter conversion. Based on 2.54

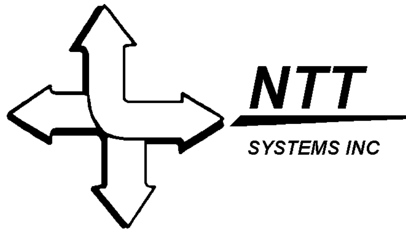
`function x2x (val)`
This is the identity conversion. That is, it simply returns its input value. For metric users, it is applied to all "Distance" variables, for Imperial users, it is applied to all "Temperature" and "DeltaT" variables, and for Husky users, it is applied to all variables.

`function defaultDoSpecial ()`
Default handler. Does nothing.

`function defaultValidation ()`
The routine returns a failure indicator for NULL and non-numeric inputs. If bounds are NULL, no further checking is done and a success indicator is `returned`. Otherwise, the variable's value is checked against the bounds, stretched by fuzz.

`function varObject (varName, varType, initialValue, mPrec, iPrec, hPrec)`
This is the base constructor for all engineering variables. It creates the object and initializes the attributes and methods. `mPrec, iPrec,` and `hPrec` specify precision for metric, Imperial and Husky users and the appropriate value is used based on the value returned for isHuskyUser, isMetricUser and isImperialUser. Conversion routines are selected based on `varType`.

`function setCalcVal (theVar, expr, feedbackBox)`
This routine is called directly from the HTML document when the user moves focus out of an input box. It should be moved to `formcom`. The routine converts the input to binary and then back to a string (using `formatNumber`). This rounds the value to the precision specified for the variable. For example, if the user specifies a TP of 255.784 degrees, a value of 256 will be used, since precision for TP is zero digits. The rounded value is then converted to Husky units by calling the variables inConv method.



Finally, the (possibly) rounded value is refreshed on the document. This ensures that the user sees the exact value that is being sent to the calculator.

`function getCalcVal (theVar)`

This routine returns the string representation of the variable in user units with the precision established when the variable was created. If no precision is specified, as is the case for "Internal" variables, no rounding and truncation is performed.

`function doRangeMapping (theProbe)`

This function is set as the map method of a mapper object. See following.

`function mapper (pairList, associatedNozVar)`

This is the constructor for mapper objects. These are used to map an input value against a series of ranges. Once the enclosing range is found, the associated value is returned. If the second parameter is non-null, it is assumed to be an engineering variable and its lower and upper bounds are set to the extremes of the `pairList`. See the discussion of L ranges in Section 5.2.1.

`function doExactMapping (theProbe)`

This function is set as the map method of a matcher object. See following.

`function matcher (pairList)`

This is the constructor for matcher objects. matchers are similar to mappers except that an input the map method must exactly match one of the values in the `pairList`.

`function initSelect (varObj, selObj, selList)`

This function initializes the designated HTML SELECT-ONE object. It creates one OPTION object for each entry in the `selList`. Users can then select one of these from the drop down list.

`function initCommon ()`

This initialization function simply adds the module's version number to the version string and builds the `calcDescription` string (eg., "750_Valve_VX").

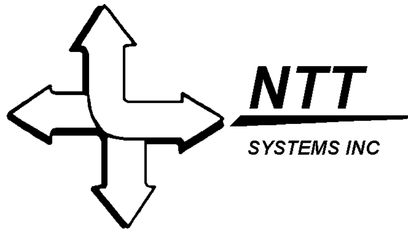
4.3 The Calculator's Structure

A calculator is created as a JavaScript object. All calculator objects have the name nozCalc. Outside users of the object (ie., functions in `formcom` and `calccom`) expect that all calculators will provide the following methods and attributes:

1. An array (parms) containing all of the engineering variables used by the calculator. It can be indexed by variable name. Eg., nozcalc.parms["DeltaT"].

2. A method named calculate. This method can be called with no parameters to calculate all of the needed internal and output variables.

3. An array (validationList) that contains a list of input variables that must be successfully validated before calculate can be invoked.



In addition, the calculator must build a mapper named LRangeMapper to handle the mapping of L dimension to nozzle length if L dimension is used (as opposed to BL dimension). Once it meets these requirements, the calculator designer can add whatever characteristics are needed.

5. 750 Nozzle Calculators

We'll look at the 750's first. These were the first ones that we built and they are more "regular" than the others are.

5.1 Series Logic

Initialization is performed in `initSeriesCalc`. It simply constructs the calculator and stores it in nozCalc. The NDrops selector is initialized based on the array `NDropsSelections`, defined locally here and used for all 750's. Finally a mapper is created if this is an L-dimension calculator, or alternately, the BLR selector is initialized.

The 750 calculator constructor (`S2Calc`) creates the following engineering variables, all of which could be displayed on a form: A, B, C, BL, BLR, F, FPrime, L, LMan, LNoz, NDrops, P, X, Y, Z, SL, LTip, TMan, and TP. The validation list is set to one of the two arrays `lValidationList` or `blValidationList` depending on the value of `isBLCalculator`.

Additional methods are created to perform calculations for the various internal and output engineering variables. These are invoked in the required order in the `calcNozzle` function that is used as the calculate method, called from `formcom`. Note that none of the functions for these additional methods are defined in `s2calc`. Their implementation is different for different groups and types.

5.2 Valve Nozzles

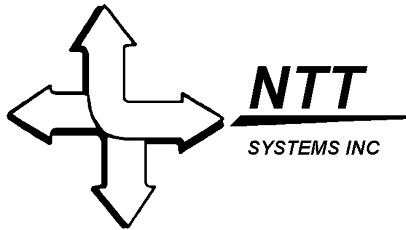
`S2vgvx`.js is the 750 valve *group* module. The initialization routine sets the value of A to 60mm. The module contains functions for calculating B, BL, C, F, FPrime, and P. Both 750 valve nozzles share these functions. The pitch limit arrays, `XPitchLimits`, `YPitchLimits`, and `ZPitchLimits` are also created here and shared by both valve nozzles.

5.2.1 Valve Nozzle Types

There are two 750 valve nozzles and the final module in the calculator is either `s2vgc.js` or `s2vxc.js`. These modules contain the function for performing the SL calculation and the arrays used to set up L range mapping and BL range selections (BLR).

The L range array for the 750-VG nozzle is as follows:

```
var LRanges = new Array (35,null, 45,70, 55,80, 65,90, 75,100, 85,110, 95,120,
105,130, 115,140, 125,150)
```



It is interpreted as follows by the mapper. The array consists of pairs of values, the first is an upper limit on the L dimension and the second is the corresponding LNoz mapping. The first pair always has NULL as the mapped value. The semantics are, "If the L value is 35 or less, there is no mapping; that is, L dimensions less than or equal to 35 are invalid." L dimension values greater than 125 will not be mapped and when the mapper is created (in `initSeriesCalc`), the bounds for L will be set to 35 and 125. Using the above L range array, if the user enters a value greater than 55 but less than or equal to 65, the mapper would return 90 as the LNoz value.

The definition of the BL range selection is more straightforward. The 750-VG BL range array is:

```
var blRangeSelections = new Array (65,70, 75,80, 85,90, 95,100, 105,110, 115,120,
125,130, 135,140, 145,150)
```

This array will be used to initialize the BLR selector for BL dimension calculators. The HTML selector will be initialized with options for 65, 75, 85, … 145. Each option will have a corresponding value, specifically 70, 80, 90, 100, … 150. If the user selects 75, LNoz will be set to 80.

5.3 Thermal Nozzles

`S2other`.js is the 750 thermal group module. The initialization routine sets the value of A to 40mm. The module contains functions for calculating B, BL, C, F, FPrime, and P. All 750 thermal nozzles share these functions. The pitch limit arrays, `XPitchLimits`, `YPitchLimits`, and `ZPitchLimits` are also created here and shared by all thermal nozzles. `s2vgvx.js` contains the functions and parameters common to all 750 valve nozzles. `s2other.js` does the same for the 750 thermal group.

5.3.1 Thermal Nozzle Types

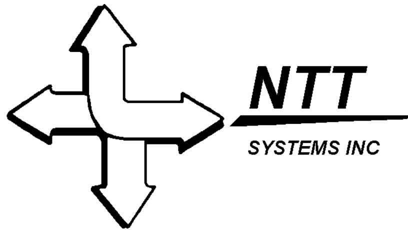
There are five different types of 750 nozzles: EG, HT, MP, SE, and TS. All differ only in parameters used to calculate SL, and in the mappings and selections for L range and BL range. Not surprisingly, once the first one was coded, the others were created by copying and modifying the original.

5.4 Some Review

Now that we have described a complete calculator, there are a few things to note.

The series level module is the place where we provide all of the variables, objects and methods that the higher level modules (`calccom` and `formcom`) expect to find.

We can place lower level implementation details, such as parameters and calculation logic wherever it makes sense. In this case, the main calculation routine could be in the series level module because all nozzles produce the same internal and output values and the order of calculation is the same in all cases. However, there were no calculations that were the same for all 750's. During the design phase, we noted that all 750 valve calculations were the same except for SL, so we implemented these in the valve group module. We could do the same for pitch limits. This meant that the remaining valve information, ie., the



SL calculation and the L and BL range information had to be implemented in the individual valve type modules.

It happens that the same design information applies to the thermal group. All 750 thermal nozzles share the same calculations for everything except SL, so these were implemented in the thermal group module. Note that they differ in detail from the valve group formulae. Again, as with the valve nozzles, the remaining functions and information had to be defined individually for each thermal nozzle in its type module.

It would not have hurt us if things were not so neat. For example, assume that each 750 thermal nozzle had its own P calculation, but all valve nozzles still had a common one. We would leave the valve group module as is, with its `pCalc` function but we would delete the `pCalc` function from the thermal group module. We would then go to each of the five thermal type modules and write a customized `pCalc` in each.

We build a calculator by selecting the appropriate *series*, *group*, and *type* modules. In general, we may not know *where* the `pCalc` function is located, but we know that there will be one, and that is all that matters.

6. 250 Nozzle Calculators

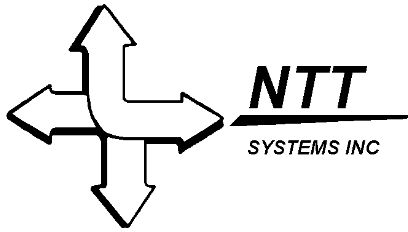
The calculator for this series contains a large amount of code that models the expansion characteristics of various materials and components. All nozzles in the series are in a single group.

Initialization follows the same pattern as the 750's except that there is no allowance for BL calculations. The following engineering variables are created: A, B, C, DeltaT, DSleeve, F, H, L, LMan, LNoz, LSeat, LSleeve, LTip, NDrops, P, SurfDist, TMan, TP, X, Y, Z, BExp, CExp, FExp, MExp, NBExp, and TExp. The calculation routine, `calcNozzle`, first calculates the expansion values and then the outputs.

The modeling functions are defined at the series level since they are independent of any particular nozzle. All of the dimension calculations are located at the group level. The individual modules at the type level are used for type specific initialization of LTip, SurfDist and the `LRanges` array.

6.1 Expansion and Deflection Modeling

The expansion calculations, BExp, CExp, FExp, MExp, NBExp, and TExp, rely on modeling logic in `slcalc`. These routines in turn rely on data structures that parameterize material and component characteristics. We will deal with these first.



6.1.1 Data Structures

Material parameters are defined in a matcher name matChars. In this case, the pairs passed to the constructor are a material name and an array of characteristics (Exp and TCond). Information is defined for materials designated as IT, BECU, H13, WR, and TITANIUM.

Three multi-segment components are evaluated in the models. These are the back-up insulator, the centre insulator, and the nozzle flange. Their net expansion values are designated as BExp, CExp, and FExp. The components are parameterized in the arrays `bExpParms`, `cExpParms`, and `fExpParms`, structured as follows.

Each array has a material designator as its first element, eg., "H13". `cExpParms` has as its second element, the *load* on the component. Each successive element, in all cases, is another array that parameterizes one segment of the insulator or flange. These must be ordered in a way that matches the physical component since the algorithms assume that adjacency has meaning for heat transfer. The secondary arrays each contain four elements, outside diameter, inside diameter, length, and a geometric factor used in calculating heat resistance (`HResis`). Note that the material designator must exactly match one of the material names in matChars.

6.1.2 Modeling Algorithms

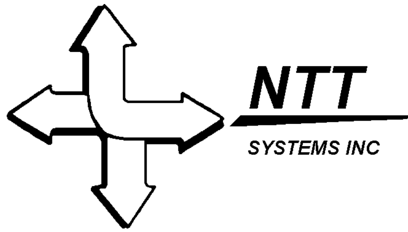
BExp and FExp only require expansion modeling. CExp is concerned with deflection that is itself based on the expansion model. These are calculated in the functions `expansionModel` and `deflectionModel` respectively.

Both begin by constructing a `modellingMatrix`. The matrix has one row for each segment of the component and one column for each parameter and internal working variable.

They then call `doBasicModelling`, passing the material designation as a parameter. `doBasicModelling` uses this to access Exp and TCond values for the material. First, area and heat resistance is calculated for each segment, and then heat loss is determined, based in part on the sum of heat resistance for all segments. Finally, average temperature, node temperature and free expansion are calculated for each segment.

When control returns to `expansionModel`, it simply sums free expansion for all segments and returns this as the result.

When control returns to `deflectionModel`, it continues in order to complete the deflection calculation. It begins by calculating Young's modulus, net expansion and deflection for each segment. Finally, it returns a result based on the sum of the net expansions and `cenman`, the load dependent deflection at the centre on the manifold plate.



7. 1250 Nozzle Calculators

Calculations for 1250's require numerous expansion calculations, but not the modeling complexity of the 250's. These nozzles present a new software problem. While all 1250 nozzles share a core set of engineering variables, the valve and thermal groups each have some of their own. Rather than creating a single calculator that included the full set, we chose to construct a separate one for each group. However, each of these is *derived*, or *subclassed*, from a calculator that implements the common components.

7.1 Series Logic

The way the 1250 calculator is constructed is unique. Recall that `formcom` calls `initTypeCalc` to build the calculator and that this is the "bottom" of the hierarchy. It calls `initGroupCalc` and it is here that the new feature is seen. The valve and thermal group modules each want a customized calculator. They accomplish this by calling `initSeriesCalc`, passing *a reference to a constructor of their own*. We raise this here, because initSeriesCalc does not build the calculator itself; it calls the constructor it receives as a parameter. This constructor will still create a "basic" 1250 calculator by calling the expected constructor, S3Calc. This will then be customized as discussed in Sections 7.2 and 7.3.

`S3Calc` creates the expected methods for calculate and for the calculation of the individual output variables. It then creates a method that will calculate all of the basic expansion values. This uses the function `S3Expansions`. Finally, it creates a method that can optionally perform group specific expansion calculations. The method name is calcGroupExps, and it is initialized to NULL, indicating that there are no custom expansion calculations. Remember that when S3Calc is done, it will return to `initSeriesCalc` that will return to `initGroupCalc` where the construction will be completed.

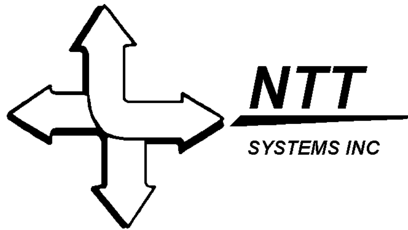
The base calculator creates the following engineering variables: A, B, C, CClear, Centre, CExp, DefH, DSExp, DeltaT, DTMan, DTP, F, FPrime, G, L, L, LMan, MExp, ML, NFExp, NIExp, NDrops, NFlange, Nozins, P, Shoe, Spring, SysExp, SysH, TShoe, X, Y, Z, TMan, and TP.

Note that pitch limits are defined at the series level since they are common to all nozzles in the series.

When it comes time to perform a calculation, the calculate method (function `calcNozzle`) will invoke the calcCommonExps method (function `S3Expansions`) and then invoke the calcGroupExps method *if* it is not NULL. Once all expansion values have been done, the output variables are calculated in the normal manner.

7.2 Valve Nozzles

`s3vgvx.js` is the *group* module for 1250-Valve nozzles. It calls `initSeriesCalc`, passing it the function `S3vgvxCalculator` as the constructor. This function first uses S3Calc to construct the base calculator and then adds the following engineering variables: CHExp, CylHeight, DTCyl, MBFExp, and MBFlange. When this has been done, `initGroupCalc` continues by initializing the variables



common to all nozzles in the group and makes the calcGroupExps method point to the valve group specific function `S3vgvxExpansions`.

Note that all calculations for 1250-valve nozzles are defined at the *group* level.

7.2.1 Valve Nozzle Types

There are two 1250 valve nozzles and the final module in the calculator is either `s3vgc.js` or `s3vxc.js`. These modules contain nothing except initialization code that sets upper and lower bounds for L and ML.

7.3 Thermal Nozzles

`s3other.js` is the *group* module for 1250-Thermal nozzles. The initialization routine `initGroupCalc` is identical to the one in the valve group, except that it creates different new variables and it points the calcGroupExps method at its `s3otherExps` function. The new variables for the thermal group are Backup and BExp.

7.3.1 Thermal Nozzle Types

There are five 1250-thermal nozzles: HT-D, HT-DC, HT-T, HT-TC, and TS. These are implemented in `s3htdc.js`, `s3htdcc.js`, `s3httc.js`, `s3httcc.js` and `s3httsc.js`. The modules contain nothing except initialization code that sets upper and lower bounds for L and ML.