Title of this help project

Table of contents

HSL Essentials

Writing methods

Writing Methods

The Hamilton Standard Language is written in text format, and is organized into statements, blocks consisting of related sets of statements, functions, one method and comments. Within a statement, the following can be used: variables, immediate data such as strings and numbers, and expressions.
The following diagram shows the general structure of an HSL program:

```
#include "Utils.hsl"      // control directives

variable v1, v2;          // global declarations
timer t1, t2;
file f1, f2;

function f()              // function definitions
{
     //...
}

function g()
{
     //...
}

method myMethod()         // "main" program
{
     //...

     if (f() == 0)
     {
          //...
          Trace("f failed\n");
     }
     if (g() == 0)
     {
          //...
          Trace("g failed\n");
     }
     //...
     return;
}
```

Method

The method marks the beginning and end of program execution. Every HSL program must have exactly one method. The method can call other functions. The method does not take any arguments and does not return a value.
The following example shows the definition of a method.

```
device Ml_Star("SystemDeckLayout.lay", "ML_STAR", hslTrue);  // Global
variable declarations
```

```
method myMethod                 // Method keyword followed by method name
()                              // No formal arguments allowed
{
     variable vol, currentPos;  // Local variable declarations
     vol = 10;                  // HSL statements
     Initialize();
     TipPickUp();
     for (currentPos = Ml_Star.SourceA.SetCurrentPosition(1);
          currentPos != 0;
          currentPos = Ml_Star.SourceA++)
     {
          Aspirate();
          Dispense();
     }
     TipEject();
     Ml_Star.StandardTips++;
}   // Method body
```

Functions

A function is a collection of HSL statements. Functions can take any number of arguments and return a scalar value of type **variable**, **string** or **sequence** or an aggregate value of type **variable**, **string**, **sequence**, **dialog**, **object**, **timer**, **event** or **file** (see also Returning a value).
The following example shows the definition of a function, which consists of a block of six statements.

```
function TrimLeadingZeros    // Function keyword followed by function name
(string barcode)             // Formal arguments
{
     variable index, length; // Variable declarations
     string null;
     null = "0";             // HSL statements
     length = barcode.GetLength();
     for (index = 0; index < length; index++)
          if (null.Compare(barcode.Mid(index, 1)) != 0)
               break;
     return(IVal(barcode.Mid(index,length)));
}   // Function body
```

Statements

A statement consists of one or more items and symbols on a line. A semicolon terminates a statement.

```
count = 5;
barcodeStr = "12345";
```

A group of statements that is surrounded by braces { } is called a block. Blocks of statements are used, for example, in functions and conditionals.

Comments

An HSL comment is written in one of the following ways:

The // (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the characters.

The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a ”single-line comment”.

```
barcodeStr = "12345";

//
This is a multi-line comment that explains the preceding code statement.
The statement assigns a value to the barcodeStr variable. The value, which
is contained between the quotation marks, is called a literal. A literal
explicitly and directly contains information; it does not refer to the
information indirectly. (The quote marks are not part of the literal.)
*/
```

Assignments and Equality

The equal sign = is used to indicate the action of assigning a value. That is, a statement could say

```
count = 5;
```

It means "Assign the value 5 to the variable `count`," or "`count` takes the value 5." If two valuesa are to be compared to determine whether they are equal, a pair of equal signs == should be used. This is discussed in detail in controlling program flow.

Expressions

An expression is something that a person can read as a Boolean or numeric expression. Expressions contain symbol characters like "+" rather than words like "added to". Any valid combination of values, variables, operators, and expressions constitutes an expression.

```
index = index + 1;
```


Variables

Variables

Variables are used to store values. They are a way to retrieve and manipulate values using names. When used effectively they can help understanding what a program does.

Declaring Variables

It is required to declare variables before using them. This is done by using the **`variable`**, **`sequence`**, **`string`**, **`device`**, **`dialog`**, **`object`**, **`timer`**, **`event`** or **`file`** keywords. Variables can be global or local to a block. Local variables are those that are only within the block.
The following code examples are of variable declaration:

```
variable count, less, factor;
string barcode;
barcode = "12345"; // The value stored in barcode is of string type. The
sentence in quotes,
                   // the value of which is assigned to barcode, is a string
```

```
literal
count = 5;          // The value stored in count has numeric type
less = count < 10; // The value stored in less has Boolean type
factor = 2.71;      // The value stored in factor has numeric type
```

Naming Variables

HSL is a case-sensitive language, so naming a variable *Counter* is different from naming it *counter*. In addition, variable names, which are restricted to the length of 255 charachters (including any namespace qualifiers), must follow certain rules:
The first character must be a letter (either uppercase or lowercase) or an underscore _.
Subsequent characters can be letters, numbers, underscores.
The variable name cannot be a keyword.

Some examples of valid variable names:

```
_count
Part9
Number_Of_Items
```

Some invalid variable names:

```
96_nunc_flat        // Starts with a number
S&W                 // Ampersand & is not a valid character for variable names
```

If a variable is declared without assigning any value to it, the variable will exist, however, its value is undefined.

```
variable index;
index = index + 1; // The value in index is undefined
```

It is not possible to use a variable that has never been declared; if tried to use such a variable, an error is generated at parse time.

```
count = 1;          // The variable count has not been declared
```

Conversion
Since HSL is a loosely-typed language, objects of the type **`variable`** have no fixed type. Instead, they have a type equivalent to the type of the value they contain. It is possible to convert a variable into a different type; for this, explicit conversion functions, IStr(), FStr(),IVal() and FVal() are provided.

```
variable start, end;
string text;
start = 1;
end = 10;
text = "Count from ";
text = text + IStr(start);
text = text + " to ";
text = text + IStr(end);
```

After this code is executed, the text variable contains "Count from 1 to 10". The number data have been converted into string form.

Data types

Data Types

What Are the HSL Data Types?
The main types are variables (**`variable`**), sequences (**`sequence`**), strings (**`string`**), devices (**`device`**), dialogs (**`dialog`**), automation objects (**`object`**), timers (**`timer`**), events (**`event`**), files (**`file`**)and errors (**`error`**).

String Data Type

Strings are delineated by double quotation marks. A string is an object, with specialelement functions. The following are examples of strings:
valid:

```
"This is a \n multi-line text "
"\"This is a text in double quotes""
"42"
```

invalid:

```
""Text"" // error: double quotes in character string
```

A string can contain zero or more characters. When it contains zero, it is called a zero-length string "".

Number Data Type

HSL supports both integer and floating-point numbers. Integers can be positive, 0, or negative; a floating-point number can contain either a decimal point, an uppercase E, which is used to represent "ten to the power of" in scientific notation, or both. The following are examples of numbers:

valid:

```
-1
0x1A
010
172
1.5
1.523E-1
```

invalid:

```
.5         // error: missing integer part
1.523E-1.5 // error: exponent not an integer
1.5 E 2    // error: space before(after) character E
```

Integers can be represented in base 10 (decimal), and base 16 (hexadecimal).
Hexadecimal integers are specified by a leading **0x** (the x must be lowercase) and can contain digits 0 through 9 and letters A through F (either uppercase or lowercase). The letter**E** is a permissible digit in hexadecimal notation and does not signify an exponential number. The letters A through F are used to represent, as single digits, the numbers that are 10 through 15 in base 10. That is, 0xF is equivalent to 15, and 0x10 is equivalent to 16.
Hexadecimal numbers can be negative, but cannot be fractional. A number that begins with a single 0 and contains a decimal point is a decimal floating-point number; a number that begins with 0x and contains a decimal point generates an error.
Some example numbers:

```
0.0001, 1E-4, 1.0E-4  // Three floating-point numbers, equivalent to each
other
3.45e2                // A floating-point number, equivalent to 345
```

```
42                  // An integer number
0377                // An integer number
0xff                // A hexadecimal integer, equivalent to 255
0x37CF              // A hexadecimal integer, equivalent to 14287
```

Booleans

In a comparison, any expression that evaluates to 0 is taken to be false, and any statement that evaluates to a number other than 0 is taken to be true.

For more information on comparisons, see controlling program flow.

Operators

Operators

HSL has a full range of operators, including arithmetic, logical, bit-wise, and assignment operators.

Symbol	Operator	Associativity
|| && | &	logical OR logical AND bitwise OR bitwise AND	left left left left
< <= == != >= >	less than less than or equal to equal not equal greater than or equal to greater than	left left left left left left
%	remainder	left
+ -	add subtract	left left
* /	multiply divide	left left
!	not	right
^	power	right
-	unary minus operator	right
++ --	increment decrement	left left
=	assignment	right

Operator Precedence

Operators in HSL are evaluated in a particular order. This order is known as the operator precedence.

Each box in the preceding table contains operators of the same priority, whereby an operator has always lower priority than the operators in the following section.
Parentheses are used to alter the order of evaluation. The expression within parentheses is fully evaluated before its value is used in the remainder of the statement.
An operator with higher precedence is evaluated before one with lower precedence. For example:
z = 78 * (96 + 3 + 45)
This expression contains five operators: =, *, (), +, and +. According to precedence, they are evaluated in the following order: (), *, +, +, =.
Evaluation of the expression within the parentheses is first: There are two addition operators, and they have the same precedence: 96 and 3 are added together and 45 is added to that total, resulting in a value of 144.
Multiplication is next: 78 and 144 are multiplied, resulting in a value of 11232.
Assignment is last: 11232 is assigned into z.

Controlling program flow

Controlling Program Flow

Why Control the Flow of Execution?
Often, a program is required to do different things under different conditions. There are several kinds of conditions that can be tested. All conditional tests in HSL are Boolean, so the result of any test is either **true** or **false**. Boolean or numeric type values can be freely tested.
HSL provides control structures for a range of possibilities. The simplest control structures are the conditional statements.

Using Conditional Statements

HSL supports **if** and **if...else** conditional statements. **if** statements test a condition, and if the condition meets the test, some HSL code written by the programmer is executed. (In the **if...else** statement, different code is executed if the condition fails the test.) The simplest form of an **if** statement can be written entirely on one line, but multi-line **if** and **if...else** statements are much more common.
The following examples demonstrate the syntax which can be used with **if** and **if...else** statements. The first example shows the simplest kind of Boolean test. If (and only if) the item between the parentheses evaluates to **true**, the statement or block of statements after the **if** is executed.

```
if (v2 != 0)
     v3 = v1 / v2;
else
     Trace("Division by zero !");
```

If only one of several conditions must be true (using || operators), testing stops as soon as any one condition passes the test. An example of the side effect of short-circuiting is that `second()` will not be executed in the following example if `first()` returns 0 or **false**.

```
if ((first() == 0) || (second() == 0))
     // some code
```

Using Repetition, or Loops

There are several ways to execute a statement or block of statements repeatedly. In general, repetitive execution is called *looping*. It is typically controlled by a test of some variable, the value of which is changed each time the loop is executed. HSL supports many types of loops: **for** loops, **while** loops, and the **loop** statement.

Using **for** Loops
The **for** loop specifies a counter variable, a test condition, and an action that updates the counter. Just before each time the loop is executed (this is called one pass or one iteration of the loop), the condition is tested. After the loop is executed, the counter variable is updated before the next iteration begins.
If the condition for looping is never met, the loop is never executed at all. If the test condition is always met, an infinite loop results. While the former may be desirable in certain cases, the latter rarely is.

```
// The update expression (currentPos = Ml_Star.SourceA++ in the following
example)is executed at the end of the loop, after the
// block of statements that forms the body of the loop is executed, and
before the condition is tested.

device Ml_Star("SystemLayout.lay", "ML_STAR", hslTrue);
variable vol, currentPos;
vol = 10;

Initialize();
TipPickUp();

for (currentPos = Ml_Star.SourceA.SetCurrentPosition(1);
     currentPos != 0;
     currentPos = Ml_Star.SourceA++)
{
     Aspirate();
     Dispense();
}

TipEject();
Ml_Star.StandardTips++;
```

Using while Loops

The **while** loop is very similar to a **for** loop. The difference is that a **while** loop does not have a built-in counter variable or an update expression. If there are already a changing condition that is reflected in the value assigned to a variable, and if this condition shall be used to control repetitive execution of a statement or block of statements, use a **while** loop.

```
device Ml_Star("SystemLayout.lay", "ML_STAR", hslTrue);
variable vol, currentPos;
vol = 10;

Initialize();
TipPickUp();
currentPos = Ml_Star.SourceA.SetCurrentPosition(1);
while (currentPos != 0)
{
     Aspirate();
     Dispense();
     currentPos = Ml_Star.SourceA++
}

TipEject();
Ml_Star.StandardTips++;
```

Using break Statements

HSL provides a statement to stop the execution of a loop. The break statement can be used to stop the execution if some (presumably special) condition is met.

```
for (index = 0; index < length; index++)
     if (nullString.Compare(barcode.Mid(index, 1)) != 0)
```

```
        break;
```


If else statement

Statements

The following statements exist: the simple, the compound, the jump, the control statement and the error handler. Statements are executed sequentially.

```
statement :
     simple_statement ';'
     compound_statement
     flow_control_statement ';'
     control_statement ';'
     error_handler
```

Simple Statement

A simple statement is an assignment or the calculation of an expression.

```
simple_statement :
     assignment_expression
     sequence_expression
     string_expression
     device_expression
     resource_expression
     dialog_expression
     object_expression
     array_expression
     timer_expression
     event_expression
     file_expression
     expression
```

Assignment:
As assignment statement, only the simple assignment operator = is supported.

```
assignment_expression :
     id '=' simple_statement
     dmemid '=' simple_statement
```

The left operand must always be an l-value. It must not be a vector, not a structure and not a function. Either both operands must be of the integer or floating point types or both operands are string types. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.

Examples:

```
v1 = 1
v2 = v1
v3 = v1 == v2
v4 = Sin(v1 + v2)
arr[v1] = (v1 + v2)^2
point.x = point.y = v1
```

String expressions:

As string expressions, only the simple assignment and the concatenation are supported.

```
string_expression :
     stringid '=' string_expression
     stringid '=' function_reference '+' string_expression
     stringid '=' cstring '+' string_expression
     stringid '=' id '+' string_expression
     stringid '=' function_reference
     stringid '=' cstring
     stringid '=' id
     stringid '+' string_expression
     stringid '+' function_reference
     stringid '+' cstring
     stringid '+' id
     stringid
```

Both operands of the operators = and + must always be of string types. The type of the result of the assignment and the concatenation is that of the left operand and the value is that, which is after the assignment or the concatenation in the left operand.

Sequence expressions:
As sequence expressions, the simple assignment, the increment and the decrement operator are supported.

```
sequence_expression :
     sequenceid '=' sequence_expression
     sequenceid '=' function_reference
     sequenceid '++'
     sequenceid '--'
     sequenceid
```

Both operands of the assignment operator must always be of the `sequence` type. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
The operand of the ++ operator must always be of the type `sequence`. The ++ operator increments the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).
The operand of the -- operator must always be of the type `sequence`. The -- operator decrements the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).

Device expressions:
As device expressions, only names are supported.

```
device_expression :
     deviceid
```

Resource expressions:
As resource expressions, only names are supported.

```
resource_expression :
     resourceid
```

Dialog expressions:
As dialog expressions, only names are supported.

```
dialog_expression :
     dialogid
```

Object expressions:
HSL supports the following object expressions:

```
object_expression :
```

```
     objectid '=' object_expression
     objectid '=' function_reference
     objectid
     objectid '.' id '=' expression
     objectid '.' id '=' object_expression
```

Array expressions:
As dynamic array expressions, the simple assignment operator = and the subscription operator [ ] are supported.

```
array_expression :
     arrayid '=' arrayid
     arrayid '=' function_reference
     arrayid '[' expression ']' '=' simple_statement
     arrayid '[' expression ']'
     arrayid
```

Both operands of the assignment operator must always be of the type of a dynamic array. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
An indexed reference to an element of a dynamic array is an expression followed by an expression in square brackets. The first expression must be a dynamic array and the second expression must have an integer type. A reference to an element of a dynamic array is an l-value.

Timer expressions:
As timer expressions, only names are supported.

```
timer_expression :
     timerid '=' timer_expression
     timerid
```

Event expressions:
As event expressions, only names are supported.

```
event_expression :
     eventid '=' event_expression
     eventid
```

File expressions:
As file expressions, only names are supported.

```
file_expression :
     fileid
```

Variable expressions:
See expressions.

Compound Statement

A compound statement is either a selection, an iteration or a block statement.

```
compound_statement:
     if_statement
     iteration_statement
     block
```

Selection statement:
Selection statements select one of several continuation possibilities in the program.

```
if_statement :
     if '(' expression ')' statement
     if '(' expression ')' statement else statement
```

The expression `expression` must be of the integer or floating point type. The expression `expression` is evaluated. If it is not equal to 0, the first sub-expression is executed. In production with `else` the second sub-expression is executed, if the expression evaluates to 0. The `else` ambiguity in nested `if` statements is resolved, by assigning the `else` to the last `else`-loose `if`.

Example:

```
if (v2 != 0)
     v3 = v1 / v2;
else
     Trace("Division by zero !");
```

Iteration statement:
Iteration statements specify loops. Loops can be nested or chained.
Two types of loops are distinguished:

- conditional loops (while and for)
- loops with counter (loop)

```
iteration_statement :
     while '(' expression ')' statement
     loop '(' expression ')' statement
     for '('opt_init_expression ';' opt_expression ';' opt_for_expression ')'
     statement

opt_init_expression :
     assignment_expression

opt_expression :
     expression

opt_for_expression :
     assignment_expression
```

In the while loop, the sub-statement `statement` is repeatedly executed until the value of the expression `expression` is 0. The expression must be of the integer or a floating point type. The while loop is executed as follows:
1. The expression `expression` is calculated.
2. If `expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the while loop to the next statement in the program. If `expression` is true (not zero), the sub-statement `statement` is executed before the program is continued with step 1.
In the loop statement, the sub-statement `statement` will be executed N times, whereby N is the initial value of the expression `expression`. The expression `expression` must be of the integer type. The loop statement is executed as follows:
1. An (anonymous) loop counter is initialized to zero.
2. The expression `expression` is calculated and assigned to an (anonymous) loop limiter.
3. The expression `loop counter < loop limiter` is calculated.
4. If the expression calculated in step 3 is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the loop statement to the next statement in the program. If the expression calculated in step 3 is true (not zero), first the sub-statement `statement` is executed, then the loop counter is incremented and finally the program is continued with step 3.
In the for loop, the sub-statement `statement` is repeated executed, until the value of the expression `opt_expression is` 0 (if available). The expression `opt_expression` must be of the integer or the floating point type. The for loop is executed as follows:

1. The expression `opt_init_expression` is calculated (if available).
2. The expression `opt_expression` is calculated (if available). If the expression `opt_expression` is missing, then its value is considered as true (not zero).
3. If the expression `opt_expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the for loop to the next statement in the program. If the

expression `opt_expression` is true (not zero), first the sub-statement `statement is` executed, then the expression `opt_for_expression` is calculated and finally the program is continued with step 2.

An iteration statement is also terminated if the sub-statement`statement` contains one of the jump statements `break`, `return` or `abort`.

Examples:

```
#define MAX 12

variable v;
v = 0;
while (v < MAX)
{
      ...
      v++;
}

while (1)  /* forever */
{
      ...
}

loop (12)
{
      ...
}

for (v = 0; v < MAX; v++)
{
      ...
}

for ( ; ; )  /* forever */
{
      ...
}
```

Block Statement:
Blocks serve to combine several statements into a statement. The body of a function is a block.

```
block :
     '{' opt_control_line opt_locals opt_statement_list '}'

opt_control_line :
     opt_control_line control_line

opt_locals :
     locals

locals :
     locals type variable_list ';'
     locals declaration ';'
     locals control_line
     type variable_list ';'
     declaration ';'

declaration:
     structure
     array
     function_declaration

variable_list :
     variable_list ',' typeid opt_initializer
```

```
     typeid opt_initializer

opt_statement_list :
     statements

statements :
     statements statement
     statements control_line
     statement
```

Example:

```
{
     variable v;
     v = 1;
     //...
}
```

Jump Statement

Jump statements result in non-conditional program execution.

```
flow_control_statement :
     break
     return
     return '(' simple_statement ')'
     abort
     pause
     onerror goto id
     onerror goto '0'
     onerror resume next
     resume next
     lock
     unlock
```

Break Statement:
A break statement may only be within an iteration statement. The program is continued with the statement, which follows directly after the abandoned statement, if one is available.

Example:

```
#define MAX 12

variable v;
v = 0;
while (1)
{
     //...
     if (MAX <= v)
          break;
     v = v + 1;
}
```

Return Statement:
The return statement consists of the symbol `return`, which possibly follows an expression in parentheses. It signals that the execution of a function is terminated. The expression specifies the value returned by a function. A function returns with a return statement to the calling context. The normal termination of a function is equivalent to a return statement.

Abort Statement:
Terminates the program abnormally, by controlled aborting the execution.

Pause Statement:
Suspends the program execution at the next position where the lock and unlock statements match exactly.

Onerror Statement:
The `onerror` statement syntax can have any of the following forms:

Statement	Description
`onerror goto id`	Enables the error handler that starts at the specified label. If a runtime error occurs, the control branches to the label, activating the error handler. The specified label must be in the same function or method as the `onerror` statement; otherwise, a syntax error occurs.
`onerror resume next`	Specifies that when a runtime error occurs, the control goes to the statement immediately following the statement where the error occurred where execution continues. The `err` object's properties are reset to zero or zero-length strings ("") after an `onerror resume next` statement. The `err.Clear()` function can be used to explicitly reset `err`.
`onerror goto 0`	Disables any enabled error handler in the current function or method.

Resume Statement:
The `resume next` statement syntax has the following form:

Statement	Description
`resume next`	Resumes execution after an error handler is finished. If the error occurred in the same function as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called function, execution resumes with the statement immediately following the statement that last called out of the function containing the error-handler (or `onerror resume next` statement).

Lock and Unlock Statement:
`lock` and `unlock` are control-of-flow language keywords that enclose a series of HSL statements so that a group of HSL statements can be executed without interruption. `lock` ... `unlock` blocks can be nested.

Example:

```
lock;
onerror goto Unexpected;
ML_STAR.Aspirate();
ML_STAR.Dispense();
onerror goto 0;
unlock;
//...

Unexpected:
{
     unlock;
     //...
}
```

Control Statement

A control statement of the form

```
control_statement :
     '<<' cstring
     '<<' stringid
```

activates the parser at runtime. The current input stream of the parser as well as the current interpreter data are stacked. The file with the name `cstring` becomes the new input stream of the parser. The character eof causes the parser to close the file with the name `cstring` and to continue the translation with the preceding input stream. The file name specified in `cstring` can be relative or absolute. The specified file will be searched in the following directories in the following sequence:
1) in the current directory.
2) in the directory that is listed under the registry key Methods.
3) in the directory that is listed under the registry key Library.
4) in the directories that are listed in the PATH environment variable.
In contrast to the control directive `#include cstring,` the control statement `<< cstring` starts a new translation at run-time. Control statements can be nested.

Example:

```
method Main()
{
     Trace("+Main()");
     //...
     << "c:\\vector\\methods\\utils.hsl";
     //...
     f1();
     Trace("-Main()");
}
```

File: `utils.hsl`

```
function f1()
{
     Trace("+f1()");
     //...
     Trace("-f1()");
}
```

Error Handler

To transfer program control on a runtime error directly to a given error handler, the error handler must be labeled. The appearance of an identifier label in the source program declares an error handler. Only an `onerror goto` statement can transfer control to an error handler.

```
error_handler :
     id ':' block
```

Example:

```
method Main()
{
     variable v;
     onerror goto UnhandledError;
     //...
     return;

     UnhandledError :
     {
```

```
          //...
          err.Clear();
          abort;
     }
}
```


For loop

Statements

The following statements exist: the simple, the compound, the jump, the control statement and the error handler. Statements are executed sequentially.

```
statement :
     simple_statement ';'
     compound_statement
     flow_control_statement ';'
     control_statement ';'
     error_handler
```

Simple Statement

A simple statement is an assignment or the calculation of an expression.

```
simple_statement :
     assignment_expression
     sequence_expression
     string_expression
     device_expression
     resource_expression
     dialog_expression
     object_expression
     array_expression
     timer_expression
     event_expression
     file_expression
     expression
```

Assignment:
As assignment statement, only the simple assignment operator = is supported.

```
assignment_expression :
     id '=' simple_statement
     dmemid '=' simple_statement
```

The left operand must always be an l-value. It must not be a vector, not a structure and not a function. Either both operands must be of the integer or floating point types or both operands are string types. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.

Examples:

```
v1 = 1
v2 = v1
v3 = v1 == v2
v4 = Sin(v1 + v2)
```

```
arr[v1] = (v1 + v2)^2
point.x = point.y = v1
```

String expressions:
As string expressions, only the simple assignment and the concatenation are supported.

```
string_expression :
     stringid '=' string_expression
     stringid '=' function_reference '+' string_expression
     stringid '=' cstring '+' string_expression
     stringid '=' id '+' string_expression
     stringid '=' function_reference
     stringid '=' cstring
     stringid '=' id
     stringid '+' string_expression
     stringid '+' function_reference
     stringid '+' cstring
     stringid '+' id
     stringid
```

Both operands of the operators = and + must always be of string types. The type of the result of the assignment and the concatenation is that of the left operand and the value is that, which is after the assignment or the concatenation in the left operand.

Sequence expressions:
As sequence expressions, the simple assignment, the increment and the decrement operator are supported.

```
sequence_expression :
     sequenceid '=' sequence_expression
     sequenceid '=' function_reference
     sequenceid '++'
     sequenceid '--'
     sequenceid
```

Both operands of the assignment operator must always be of the `sequence` type. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
The operand of the ++ operator must always be of the type `sequence`. The ++ operator increments the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).
The operand of the -- operator must always be of the type `sequence`. The -- operator decrements the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).

Device expressions:
As device expressions, only names are supported.

```
device_expression :
     deviceid
```

Resource expressions:
As resource expressions, only names are supported.

```
resource_expression :
     resourceid
```

Dialog expressions:
As dialog expressions, only names are supported.

```
dialog_expression :
     dialogid
```

Object expressions:
HSL supports the following object expressions:

```
object_expression :
     objectid '=' object_expression
     objectid '=' function_reference
     objectid
     objectid '.' id '=' expression
     objectid '.' id '=' object_expression
```

Array expressions:
As dynamic array expressions, the simple assignment operator = and the subscription operator [ ] are supported.

```
array_expression :
     arrayid '=' arrayid
     arrayid '=' function_reference
     arrayid '[' expression ']' '=' simple_statement
     arrayid '[' expression ']'
     arrayid
```

Both operands of the assignment operator must always be of the type of a dynamic array. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
An indexed reference to an element of a dynamic array is an expression followed by an expression in square brackets. The first expression must be a dynamic array and the second expression must have an integer type. A reference to an element of a dynamic array is an l-value.

Timer expressions:
As timer expressions, only names are supported.

```
timer_expression :
     timerid '=' timer_expression
     timerid
```

Event expressions:
As event expressions, only names are supported.

```
event_expression :
     eventid '=' event_expression
     eventid
```

File expressions:
As file expressions, only names are supported.

```
file_expression :
     fileid
```

Variable expressions:
See expressions.

Compound Statement

A compound statement is either a selection, an iteration or a block statement.

```
compound_statement:
     if_statement
     iteration_statement
     block
```

Selection statement:
Selection statements select one of several continuation possibilities in the program.

```
if_statement :
     if '(' expression ')' statement
     if '(' expression ')' statement else statement
```

The expression `expression` must be of the integer or floating point type. The expression `expression` is evaluated. If it is not equal to 0, the first sub-expression is executed. In production with `else` the second sub-expression is executed, if the expression evaluates to 0. The `else` ambiguity in nested `if` statements is resolved, by assigning the `else` to the last `else`-loose `if`.

Example:

```
if (v2 != 0)
     v3 = v1 / v2;
else
     Trace("Division by zero !");
```

Iteration statement:
Iteration statements specify loops. Loops can be nested or chained.
Two types of loops are distinguished:

- conditional loops (while and for)
- loops with counter (loop)

```
iteration_statement :
     while '(' expression ')' statement
     loop '(' expression ')' statement
     for '('opt_init_expression ';' opt_expression ';' opt_for_expression ')'
     statement

opt_init_expression :
     assignment_expression

opt_expression :
     expression

opt_for_expression :
     assignment_expression
```

In the while loop, the sub-statement `statement` is repeatedly executed until the value of the expression `expression` is 0. The expression must be of the integer or a floating point type. The while loop is executed as follows:
1. The expression `expression` is calculated.
2. If `expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the while loop to the next statement in the program. If `expression` is true (not zero), the sub-statement `statement` is executed before the program is continued with step 1.
In the loop statement, the sub-statement `statement` will be executed N times, whereby N is the initial value of the expression `expression`. The expression `expression` must be of the integer type. The loop statement is executed as follows:
1. An (anonymous) loop counter is initialized to zero.
2. The expression `expression` is calculated and assigned to an (anonymous) loop limiter.
3. The expression `loop counter < loop limiter` is calculated.
4. If the expression calculated in step 3 is initially false (zero), the sub-statements `statement` is never executed and the control flow goes from the loop statement to the next statement in the program. If the expression calculated in step 3 is true (not zero), first the sub-statement `statement` is executed, then the loop counter is incremented and finally the program is continued with step 3.
In the for loop, the sub-statement `statement` is repeated executed, until the value of the expression `opt_expression is 0` (if available). The expression `opt_expression` must be of the integer or the floating point type. The for loop is executed as follows:

1. The expression `opt_init_expression` is calculated (if available).
2. The expression `opt_expression` is calculated (if available). If the expression `opt_expression` is

missing, then its value is considered as true (not zero).
3. If the expression `opt_expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the for loop to the next statement in the program. If the expression `opt_expression` is true (not zero), first the sub-statement `statement is` executed, then the expression `opt_for_expression` is calculated and finally the program is continued with step 2.

An iteration statement is also terminated if the sub-statement `statement` contains one of the jump statements `break`, `return` or `abort`.

Examples:

```
#define MAX 12

variable v;
v = 0;
while (v < MAX)
{
      ...
      v++;
}

while (1)  /* forever */
{
      ...
}

loop (12)
{
      ...
}

for (v = 0; v < MAX; v++)
{
      ...
}

for ( ; ; )  /* forever */
{
      ...
}
```

Block Statement:
Blocks serve to combine several statements into a statement. The body of a function is a block.

```
block :
     '{' opt_control_line opt_locals opt_statement_list '}'

opt_control_line :
     opt_control_line control_line

opt_locals :
     locals

locals :
     locals type variable_list ';'
     locals declaration ';'
     locals control_line
     type variable_list ';'
     declaration ';'

declaration:
     structure
     array
     function_declaration
```

```
variable_list :
     variable_list ',' typeid opt_initializer
     typeid opt_initializer

opt_statement_list :
     statements

statements :
     statements statement
     statements control_line
     statement
```

Example:

```
{
     variable v;
     v = 1;
     //...
}
```

Jump Statement

Jump statements result in non-conditional program execution.

```
flow_control_statement :
     break
     return
     return '(' simple_statement ')'
     abort
     pause
     onerror goto id
     onerror goto '0'
     onerror resume next
     resume next
     lock
     unlock
```

Break Statement:
A break statement may only be within an iteration statement. The program is continued with the statement, which follows directly after the abandoned statement, if one is available.

Example:

```
#define MAX 12

variable v;
v = 0;
while (1)
{
     //...
     if (MAX <= v)
          break;
     v = v + 1;
}
```

Return Statement:
The return statement consists of the symbol `return`, which possibly follows an expression in parentheses. It signals that the execution of a function is terminated. The expression specifies the value returned by a function. A function returns with a return statement to the calling context. The normal termination of a function is equivalent to a return statement.

Abort Statement:

Terminates the program abnormally, by controlled aborting the execution.

Pause Statement:
Suspends the program execution at the next position where the lock and unlock statements match exactly.

Onerror Statement:
The `onerror` statement syntax can have any of the following forms:

Statement	Description
`onerror goto id`	Enables the error handler that starts at the specified label. If a runtime error occurs, the control branches to the label, activating the error handler. The specified label must be in the same function or method as the `onerror` statement; otherwise, a syntax error occurs.
`onerror resume next`	Specifies that when a runtime error occurs, the control goes to the statement immediately following the statement where the error occurred where execution continues. The `err` object's properties are reset to zero or zero-length strings ("") after an `onerror resume next` statement. The `err.Clear()` function can be used to explicitly reset `err`.
`onerror goto 0`	Disables any enabled error handler in the current function or method.

Resume Statement:
The `resume next` statement syntax has the following form:

Statement	Description
`resume next`	Resumes execution after an error handler is finished. If the error occurred in the same function as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called function, execution resumes with the statement immediately following the statement that last called out of the function containing the error-handler (or `onerror resume next` statement).

Lock and Unlock Statement:
`lock` and `unlock` are control-of-flow language keywords that enclose a series of HSL statements so that a group of HSL statements can be executed without interruption. `lock` ... `unlock` blocks can be nested.

Example:

```
lock;
onerror goto Unexpected;
ML_STAR.Aspirate();
ML_STAR.Dispense();
onerror goto 0;
unlock;
//...

Unexpected:
{
     unlock;
     //...
}
```

Control Statement

A control statement of the form

```
control_statement :
     '<<' cstring
     '<<' stringid
```

activates the parser at runtime. The current input stream of the parser as well as the current interpreter data are stacked. The file with the name `cstring` becomes the new input stream of the parser. The character eof causes the parser to close the file with the name `cstring` and to continue the translation with the preceding input stream. The file name specified in `cstring` can be relative or absolute. The specified file will be searched in the following directories in the following sequence:
1) in the current directory.
2) in the directory that is listed under the registry key Methods.
3) in the directory that is listed under the registry key Library.
4) in the directories that are listed in the PATH environment variable.
In contrast to the control directive `#include cstring,` the control statement `<< cstring` starts a new translation at run-time. Control statements can be nested.

Example:

```
method Main()
{
     Trace("+Main()");
     //...
     << "c:\\vector\\methods\\utils.hsl";
     //...
     f1();
     Trace("-Main()");
}
```

File: `utils.hsl`

```
function f1()
{
     Trace("+f1()");
     //...
     Trace("-f1()");
}
```

Error Handler

To transfer program control on a runtime error directly to a given error handler, the error handler must be labeled. The appearance of an identifier label in the source program declares an error handler. Only an `onerror goto` statement can transfer control to an error handler.

```
error_handler :
     id ':' block
```

Example:

```
method Main()
{
     variable v;
     onerror goto UnhandledError;
     //...
     return;

     UnhandledError :
     {
```

```
          //...
          err.Clear();
          abort;
     }
}
```


While loop

Statements

The following statements exist: the simple, the compound, the jump, the control statement and the error handler. Statements are executed sequentially.

```
statement :
     simple_statement ';'
     compound_statement
     flow_control_statement ';'
     control_statement ';'
     error_handler
```

Simple Statement

A simple statement is an assignment or the calculation of an expression.

```
simple_statement :
     assignment_expression
     sequence_expression
     string_expression
     device_expression
     resource_expression
     dialog_expression
     object_expression
     array_expression
     timer_expression
     event_expression
     file_expression
     expression
```

Assignment:
As assignment statement, only the simple assignment operator = is supported.

```
assignment_expression :
     id '=' simple_statement
     dmemid '=' simple_statement
```

The left operand must always be an l-value. It must not be a vector, not a structure and not a function. Either both operands must be of the integer or floating point types or both operands are string types. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.

Examples:

```
v1 = 1
v2 = v1
v3 = v1 == v2
v4 = Sin(v1 + v2)
arr[v1] = (v1 + v2)^2
point.x = point.y = v1
```

String expressions:
As string expressions, only the simple assignment and the concatenation are supported.

```
string_expression :
     stringid '=' string_expression
     stringid '=' function_reference '+' string_expression
     stringid '=' cstring '+' string_expression
     stringid '=' id '+' string_expression
     stringid '=' function_reference
     stringid '=' cstring
     stringid '=' id
     stringid '+' string_expression
     stringid '+' function_reference
     stringid '+' cstring
     stringid '+' id
     stringid
```

Both operands of the operators = and + must always be of string types. The type of the result of the assignment and the concatenation is that of the left operand and the value is that, which is after the assignment or the concatenation in the left operand.

Sequence expressions:
As sequence expressions, the simple assignment, the increment and the decrement operator are supported.

```
sequence_expression :
     sequenceid '=' sequence_expression
     sequenceid '=' function_reference
     sequenceid '++'
     sequenceid '--'
     sequenceid
```

Both operands of the assignment operator must always be of the `sequence` type. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
The operand of the ++ operator must always be of the type `sequence`. The ++ operator increments the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).
The operand of the -- operator must always be of the type `sequence`. The -- operator decrements the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).

Device expressions:
As device expressions, only names are supported.

```
device_expression :
     deviceid
```

Resource expressions:
As resource expressions, only names are supported.

```
resource_expression :
     resourceid
```

Dialog expressions:
As dialog expressions, only names are supported.

```
dialog_expression :
     dialogid
```

Object expressions:
HSL supports the following object expressions:

```
object_expression :
     objectid '=' object_expression
     objectid '=' function_reference
     objectid
     objectid '.' id '=' expression
     objectid '.' id '=' object_expression
```

Array expressions:
As dynamic array expressions, the simple assignment operator = and the subscription operator [ ] are supported.

```
array_expression :
     arrayid '=' arrayid
     arrayid '=' function_reference
     arrayid '[' expression ']' '=' simple_statement
     arrayid '[' expression ']'
     arrayid
```

Both operands of the assignment operator must always be of the type of a dynamic array. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
An indexed reference to an element of a dynamic array is an expression followed by an expression in square brackets. The first expression must be a dynamic array and the second expression must have an integer type. A reference to an element of a dynamic array is an l-value.

Timer expressions:
As timer expressions, only names are supported.

```
timer_expression :
     timerid '=' timer_expression
     timerid
```

Event expressions:
As event expressions, only names are supported.

```
event_expression :
     eventid '=' event_expression
     eventid
```

File expressions:
As file expressions, only names are supported.

```
file_expression :
     fileid
```

Variable expressions:
See expressions.

Compound Statement

A compound statement is either a selection, an iteration or a block statement.

```
compound_statement:
     if_statement
     iteration_statement
     block
```

Selection statement:
Selection statements select one of several continuation possibilities in the program.

```
if_statement :
```

```
     if '(' expression ')' statement
     if '(' expression ')' statement else statement
```

The expression `expression` must be of the integer or floating point type. The expression `expression` is evaluated. If it is not equal to 0, the first sub-expression is executed. In production with `else` the second sub-expression is executed, if the expression evaluates to 0. The `else` ambiguity in nested `if` statements is resolved, by assigning the `else` to the last `else`-loose `if`.

Example:

```
if (v2 != 0)
     v3 = v1 / v2;
else
     Trace("Division by zero !");
```

Iteration statement:
Iteration statements specify loops. Loops can be nested or chained.
Two types of loops are distinguished:

- conditional loops (while and for)
- loops with counter (loop)

```
iteration_statement :
     while '(' expression ')' statement
     loop '(' expression ')' statement
     for '('opt_init_expression ';' opt_expression ';' opt_for_expression ')'
     statement

opt_init_expression :
     assignment_expression

opt_expression :
     expression

opt_for_expression :
     assignment_expression
```

In the while loop, the sub-statement `statement` is repeatedly executed until the value of the expression `expression` is 0. The expression must be of the integer or a floating point type. The while loop is executed as follows:
1. The expression `expression` is calculated.
2. If `expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the while loop to the next statement in the program. If `expression` is true (not zero), the sub-statement `statement` is executed before the program is continued with step 1.
In the loop statement, the sub-statement `statement` will be executed N times, whereby N is the initial value of the expression `expression`. The expression `expression` must be of the integer type. The loop statement is executed as follows:
1. An (anonymous) loop counter is initialized to zero.
2. The expression `expression` is calculated and assigned to an (anonymous) loop limiter.
3. The expression `loop counter < loop limiter` is calculated.
4. If the expression calculated in step 3 is initially false (zero), the sub-statements `statement` is never executed and the control flow goes from the loop statement to the next statement in the program. If the expression calculated in step 3 is true (not zero), first the sub-statement `statement` is executed, then the loop counter is incremented and finally the program is continued with step 3.
In the for loop, the sub-statement `statement` is repeated executed, until the value of the expression `opt_expression is` 0 (if available). The expression `opt_expression` must be of the integer or the floating point type. The for loop is executed as follows:

1. The expression `opt_init_expression` is calculated (if available).
2. The expression `opt_expression` is calculated (if available). If the expression `opt_expression` is missing, then its value is considered as true (not zero).

3. If the expression `opt_expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the for loop to the next statement in the program. If the expression `opt_expression` is true (not zero), first the sub-statement `statement is` executed, then the expression `opt_for_expression` is calculated and finally the program is continued with step 2.

An iteration statement is also terminated if the sub-statement `statement` contains one of the jump statements `break`, `return` or `abort`.

Examples:

```
#define MAX 12

variable v;
v = 0;
while (v < MAX)
{
      ...
      v++;
}

while (1)  /* forever */
{
      ...
}

loop (12)
{
      ...
}

for (v = 0; v < MAX; v++)
{
      ...
}

for ( ; ; )  /* forever */
{
      ...
}
```

Block Statement:
Blocks serve to combine several statements into a statement. The body of a function is a block.

```
block :
     '{' opt_control_line opt_locals opt_statement_list '}'

opt_control_line :
     opt_control_line control_line

opt_locals :
     locals

locals :
     locals type variable_list ';'
     locals declaration ';'
     locals control_line
     type variable_list ';'
     declaration ';'

declaration:
     structure
     array
     function_declaration
```

```
variable_list :
     variable_list ',' typeid opt_initializer
     typeid opt_initializer

opt_statement_list :
     statements

statements :
     statements statement
     statements control_line
     statement
```

Example:

```
{
     variable v;
     v = 1;
     //...
}
```

Jump Statement

Jump statements result in non-conditional program execution.

```
flow_control_statement :
     break
     return
     return '(' simple_statement ')'
     abort
     pause
     onerror goto id
     onerror goto '0'
     onerror resume next
     resume next
     lock
     unlock
```

Break Statement:
A break statement may only be within an iteration statement. The program is continued with the statement, which follows directly after the abandoned statement, if one is available.

Example:

```
#define MAX 12

variable v;
v = 0;
while (1)
{
     //...
     if (MAX <= v)
          break;
     v = v + 1;
}
```

Return Statement:
The return statement consists of the symbol `return`, which possibly follows an expression in parentheses. It signals that the execution of a function is terminated. The expression specifies the value returned by a function. A function returns with a return statement to the calling context. The normal termination of a function is equivalent to a return statement.

Abort Statement:
Terminates the program abnormally, by controlled aborting the execution.

Pause Statement:
Suspends the program execution at the next position where the lock and unlock statements match exactly.

Onerror Statement:
The `onerror` statement syntax can have any of the following forms:

Statement	Description
`onerror goto id`	Enables the error handler that starts at the specified label. If a runtime error occurs, the control branches to the label, activating the error handler. The specified label must be in the same function or method as the `onerror` statement; otherwise, a syntax error occurs.
`onerror resume next`	Specifies that when a runtime error occurs, the control goes to the statement immediately following the statement where the error occurred where execution continues. The `err` object's properties are reset to zero or zero-length strings ("") after an `onerror resume next` statement. The `err.Clear()` function can be used to explicitly reset `err`.
`onerror goto 0`	Disables any enabled error handler in the current function or method.

Resume Statement:
The `resume next` statement syntax has the following form:

Statement	Description
`resume next`	Resumes execution after an error handler is finished. If the error occurred in the same function as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called function, execution resumes with the statement immediately following the statement that last called out of the function containing the error-handler (or `onerror resume next` statement).

Lock and Unlock Statement:
`lock` and `unlock` are control-of-flow language keywords that enclose a series of HSL statements so that a group of HSL statements can be executed without interruption. `lock` ... `unlock` blocks can be nested.

Example:

```
lock;
onerror goto Unexpected;
ML_STAR.Aspirate();
ML_STAR.Dispense();
onerror goto 0;
unlock;
//...

Unexpected:
{
     unlock;
     //...
}
```

Control Statement

A control statement of the form

```
control_statement :
     '<<' cstring
     '<<' stringid
```

activates the parser at runtime. The current input stream of the parser as well as the current interpreter data are stacked. The file with the name `cstring` becomes the new input stream of the parser. The character eof causes the parser to close the file with the name `cstring` and to continue the translation with the preceding input stream. The file name specified in `cstring` can be relative or absolute. The specified file will be searched in the following directories in the following sequence:
1) in the current directory.
2) in the directory that is listed under the registry key Methods.
3) in the directory that is listed under the registry key Library.
4) in the directories that are listed in the PATH environment variable.
In contrast to the control directive `#include cstring,` the control statement `<< cstring` starts a new translation at run-time. Control statements can be nested.

Example:

```
method Main()
{
     Trace("+Main()");
     //...
     << "c:\\vector\\methods\\utils.hsl";
     //...
     f1();
     Trace("-Main()");
}
```

File: `utils.hsl`

```
function f1()
{
     Trace("+f1()");
     //...
     Trace("-f1()");
}
```

Error Handler

To transfer program control on a runtime error directly to a given error handler, the error handler must be labeled. The appearance of an identifier label in the source program declares an error handler. Only an `onerror goto` statement can transfer control to an error handler.

```
error_handler :
     id ':' block
```

Example:

```
method Main()
{
     variable v;
     onerror goto UnhandledError;
     //...
     return;

     UnhandledError :
     {
```

```
          //...
          err.Clear();
          abort;
     }
}
```


Loop statement

Statements

The following statements exist: the simple, the compound, the jump, the control statement and the error handler. Statements are executed sequentially.

```
statement :
     simple_statement ';'
     compound_statement
     flow_control_statement ';'
     control_statement ';'
     error_handler
```

Simple Statement

A simple statement is an assignment or the calculation of an expression.

```
simple_statement :
     assignment_expression
     sequence_expression
     string_expression
     device_expression
     resource_expression
     dialog_expression
     object_expression
     array_expression
     timer_expression
     event_expression
     file_expression
     expression
```

Assignment:
As assignment statement, only the simple assignment operator = is supported.

```
assignment_expression :
     id '=' simple_statement
     dmemid '=' simple_statement
```

The left operand must always be an l-value. It must not be a vector, not a structure and not a function. Either both operands must be of the integer or floating point types or both operands are string types. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.

Examples:

```
v1 = 1
v2 = v1
v3 = v1 == v2
v4 = Sin(v1 + v2)
arr[v1] = (v1 + v2)^2
point.x = point.y = v1
```

String expressions:
As string expressions, only the simple assignment and the concatenation are supported.

```
string_expression :
     stringid '=' string_expression
     stringid '=' function_reference '+' string_expression
     stringid '=' cstring '+' string_expression
     stringid '=' id '+' string_expression
     stringid '=' function_reference
     stringid '=' cstring
     stringid '=' id
     stringid '+' string_expression
     stringid '+' function_reference
     stringid '+' cstring
     stringid '+' id
     stringid
```

Both operands of the operators = and + must always be of string types. The type of the result of the assignment and the concatenation is that of the left operand and the value is that, which is after the assignment or the concatenation in the left operand.

Sequence expressions:
As sequence expressions, the simple assignment, the increment and the decrement operator are supported.

```
sequence_expression :
     sequenceid '=' sequence_expression
     sequenceid '=' function_reference
     sequenceid '++'
     sequenceid '--'
     sequenceid
```

Both operands of the assignment operator must always be of the `sequence` type. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
The operand of the ++ operator must always be of the type `sequence`. The ++ operator increments the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).
The operand of the -- operator must always be of the type `sequence`. The -- operator decrements the current position of the calling object by (used - current + 1) positions in iteration order (seeelement functions).

Device expressions:
As device expressions, only names are supported.

```
device_expression :
     deviceid
```

Resource expressions:
As resource expressions, only names are supported.

```
resource_expression :
     resourceid
```

Dialog expressions:
As dialog expressions, only names are supported.

```
dialog_expression :
     dialogid
```

Object expressions:
HSL supports the following object expressions:

```
object_expression :
     objectid '=' object_expression
     objectid '=' function_reference
     objectid
     objectid '.' id '=' expression
     objectid '.' id '=' object_expression
```

Array expressions:
As dynamic array expressions, the simple assignment operator = and the subscription operator [ ] are supported.

```
array_expression :
     arrayid '=' arrayid
     arrayid '=' function_reference
     arrayid '[' expression ']' '=' simple_statement
     arrayid '[' expression ']'
     arrayid
```

Both operands of the assignment operator must always be of the type of a dynamic array. The type of the result of the assignment is that of the left operand and the value is that, which is after the assignment in the left operand.
An indexed reference to an element of a dynamic array is an expression followed by an expression in square brackets. The first expression must be a dynamic array and the second expression must have an integer type. A reference to an element of a dynamic array is an l-value.

Timer expressions:
As timer expressions, only names are supported.

```
timer_expression :
     timerid '=' timer_expression
     timerid
```

Event expressions:
As event expressions, only names are supported.

```
event_expression :
     eventid '=' event_expression
     eventid
```

File expressions:
As file expressions, only names are supported.

```
file_expression :
     fileid
```

Variable expressions:
See expressions.

Compound Statement

A compound statement is either a selection, an iteration or a block statement.

```
compound_statement:
     if_statement
     iteration_statement
     block
```

Selection statement:
Selection statements select one of several continuation possibilities in the program.

```
if_statement :
```

```
     if '(' expression ')' statement
     if '(' expression ')' statement else statement
```

The expression `expression` must be of the integer or floating point type. The expression `expression` is evaluated. If it is not equal to 0, the first sub-expression is executed. In production with `else` the second sub-expression is executed, if the expression evaluates to 0. The `else` ambiguity in nested `if` statements is resolved, by assigning the `else` to the last `else`-loose `if`.

Example:

```
if (v2 != 0)
     v3 = v1 / v2;
else
     Trace("Division by zero !");
```

Iteration statement:
Iteration statements specify loops. Loops can be nested or chained.
Two types of loops are distinguished:

- conditional loops (while and for)
- loops with counter (loop)

```
iteration_statement :
     while '(' expression ')' statement
     loop '(' expression ')' statement
     for '('opt_init_expression ';' opt_expression ';' opt_for_expression ')'
     statement

opt_init_expression :
     assignment_expression

opt_expression :
     expression

opt_for_expression :
     assignment_expression
```

In the while loop, the sub-statement `statement` is repeatedly executed until the value of the expression `expression` is 0. The expression must be of the integer or a floating point type. The while loop is executed as follows:
1. The expression `expression` is calculated.
2. If `expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the while loop to the next statement in the program. If `expression` is true (not zero), the sub-statement `statement` is executed before the program is continued with step 1.
In the loop statement, the sub-statement `statement` will be executed N times, whereby N is the initial value of the expression `expression`. The expression `expression` must be of the integer type. The loop statement is executed as follows:
1. An (anonymous) loop counter is initialized to zero.
2. The expression `expression` is calculated and assigned to an (anonymous) loop limiter.
3. The expression `loop counter < loop limiter` is calculated.
4. If the expression calculated in step 3 is initially false (zero), the sub-statements `statement` is never executed and the control flow goes from the loop statement to the next statement in the program. If the expression calculated in step 3 is true (not zero), first the sub-statement `statement` is executed, then the loop counter is incremented and finally the program is continued with step 3.
In the for loop, the sub-statement `statement` is repeated executed, until the value of the expression `opt_expression is` 0 (if available). The expression `opt_expression` must be of the integer or the floating point type. The for loop is executed as follows:

1. The expression `opt_init_expression` is calculated (if available).
2. The expression `opt_expression` is calculated (if available). If the expression `opt_expression` is missing, then its value is considered as true (not zero).

3. If the expression `opt_expression` is initially false (zero), the sub-statement `statement` is never executed and the control flow goes from the for loop to the next statement in the program. If the expression `opt_expression` is true (not zero), first the sub-statement `statement is` executed, then the expression `opt_for_expression` is calculated and finally the program is continued with step 2.

An iteration statement is also terminated if the sub-statement `statement` contains one of the jump statements `break`, `return` or `abort`.

Examples:

```
#define MAX 12

variable v;
v = 0;
while (v < MAX)
{
      ...
      v++;
}

while (1)  /* forever */
{
      ...
}

loop (12)
{
      ...
}

for (v = 0; v < MAX; v++)
{
      ...
}

for ( ; ; )  /* forever */
{
      ...
}
```

Block Statement:
Blocks serve to combine several statements into a statement. The body of a function is a block.

```
block :
     '{' opt_control_line opt_locals opt_statement_list '}'

opt_control_line :
     opt_control_line control_line

opt_locals :
     locals

locals :
     locals type variable_list ';'
     locals declaration ';'
     locals control_line
     type variable_list ';'
     declaration ';'

declaration:
     structure
     array
     function_declaration
```

```
variable_list :
     variable_list ',' typeid opt_initializer
     typeid opt_initializer

opt_statement_list :
     statements

statements :
     statements statement
     statements control_line
     statement
```

Example:

```
{
     variable v;
     v = 1;
     //...
}
```

Jump Statement

Jump statements result in non-conditional program execution.

```
flow_control_statement :
     break
     return
     return '(' simple_statement ')'
     abort
     pause
     onerror goto id
     onerror goto '0'
     onerror resume next
     resume next
     lock
     unlock
```

Break Statement:
A break statement may only be within an iteration statement. The program is continued with the statement, which follows directly after the abandoned statement, if one is available.

Example:

```
#define MAX 12

variable v;
v = 0;
while (1)
{
     //...
     if (MAX <= v)
          break;
     v = v + 1;
}
```

Return Statement:
The return statement consists of the symbol `return`, which possibly follows an expression in parentheses. It signals that the execution of a function is terminated. The expression specifies the value returned by a function. A function returns with a return statement to the calling context. The normal termination of a function is equivalent to a return statement.

Abort Statement:
Terminates the program abnormally, by controlled aborting the execution.

Pause Statement:
Suspends the program execution at the next position where the lock and unlock statements match exactly.

Onerror Statement:
The `onerror` statement syntax can have any of the following forms:

Statement	Description
`onerror goto id`	Enables the error handler that starts at the specified label. If a runtime error occurs, the control branches to the label, activating the error handler. The specified label must be in the same function or method as the `onerror` statement; otherwise, a syntax error occurs.
`onerror resume next`	Specifies that when a runtime error occurs, the control goes to the statement immediately following the statement where the error occurred where execution continues. The `err` object's properties are reset to zero or zero-length strings ("") after an `onerror resume next` statement. The `err.Clear()` function can be used to explicitly reset `err`.
`onerror goto 0`	Disables any enabled error handler in the current function or method.

Resume Statement:
The `resume next` statement syntax has the following form:

Statement	Description
`resume next`	Resumes execution after an error handler is finished. If the error occurred in the same function as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called function, execution resumes with the statement immediately following the statement that last called out of the function containing the error-handler (or `onerror resume next` statement).

Lock and Unlock Statement:
`lock` and `unlock` are control-of-flow language keywords that enclose a series of HSL statements so that a group of HSL statements can be executed without interruption. `lock` ... `unlock` blocks can be nested.

Example:

```
lock;
onerror goto Unexpected;
ML_STAR.Aspirate();
ML_STAR.Dispense();
onerror goto 0;
unlock;
//...

Unexpected:
{
     unlock;
     //...
}
```

Control Statement

A control statement of the form

```
control_statement :
     '<<' cstring
     '<<' stringid
```

activates the parser at runtime. The current input stream of the parser as well as the current interpreter data are stacked. The file with the name `cstring` becomes the new input stream of the parser. The character eof causes the parser to close the file with the name `cstring` and to continue the translation with the preceding input stream. The file name specified in `cstring` can be relative or absolute. The specified file will be searched in the following directories in the following sequence:
1) in the current directory.
2) in the directory that is listed under the registry key Methods.
3) in the directory that is listed under the registry key Library.
4) in the directories that are listed in the PATH environment variable.
In contrast to the control directive `#include cstring,` the control statement `<< cstring` starts a new translation at run-time. Control statements can be nested.

Example:

```
method Main()
{
     Trace("+Main()");
     //...
     << "c:\\vector\\methods\\utils.hsl";
     //...
     f1();
     Trace("-Main()");
}
```

File: `utils.hsl`

```
function f1()
{
     Trace("+f1()");
     //...
     Trace("-f1()");
}
```

Error Handler

To transfer program control on a runtime error directly to a given error handler, the error handler must be labeled. The appearance of an identifier label in the source program declares an error handler. Only an `onerror goto` statement can transfer control to an error handler.

```
error_handler :
     id ':' block
```

Example:

```
method Main()
{
     variable v;
     onerror goto UnhandledError;
     //...
     return;

     UnhandledError :
     {
```

```
          //...
          err.Clear();
          abort;
     }
}
```


Functions

Functions

What Is a Function?
HSL functions perform actions. They can also return results. Sometimes these are the results of calculations or comparisons.
Functions combine several operations under one name and enable the streamlining of the written code. It is possible to write a set of statements, name it, and then execute the entire set at any time, just by calling it and passing to it the information it needs.
Information is passed to a function by enclosing the information in parentheses after the function's name. Pieces of information that are being passed to a function are called arguments or parameters. Some functions do not take any arguments at all; some functions take one argument; some take several. There are even functions for which the number of arguments depends on how the function is used.
HSL supports two kinds of functions: those that are built into the language, and those created by the programmer.

Special Built-in Functions
The HSL language includes several built-in functions. Some of them enable the handling of expressions and special characters, and convert strings to numeric values. For example, IStr() and FStr() are used to convert numbers to strings.
Consult the library functions and element functions for more information about these and other built-in functions.

Creating Own Functions
Programmers can create their own functions and use them wherever needed. A function definition consists of a function header and a block of HSL statements.
The `TrimLeadingZeros` function in the following example takes as its argument a barcode string, and trims leading zero characters from the string before returning with the number representation of the barcode.

```
function TrimLeadingZeros     // Function keyword followed by function name
(string barcode)              // Formal arguments
{
     variable index, length;  // Variable declarations
     string null;
     null = "0";              // HSL statements
     length = barcode.GetLength();
     for (index = 0; index < length; index++)
          if (null.Compare(barcode.Mid(index, 1)) != 0)
               break;
     return(IVal(barcode.Mid(index,length)));
}                             // Function body
```

Objects

Objects

What Are Objects?
In HSL, objects are collections of data elements and element functions. An element function is a function that is a member of an object (e.g. `device.GetSequence("SomeSequence")`), and a data element is a value that is a member of an object (e.g. `device.SomeSequence`).

Reserved keywords

Keywords

The following words are reserved words and can only be used with their pre-defined meaning:

Keywords		
abort	goto	resume
break	if	sequence
char	ifdef	short
const	ifndef	static
device	include	struct
debug	lock	string
define	long	synchronized
dialog	loop	timer
echo	method	unlock
else	namespace	variable
endif	next	void
error	object	while
event	once	__filename__
file	onerror	
float	pause	
for	pragma	
function	private	
global	return	

Predefined constants

Predefined Constants

Since the following constants are built into HSL, i.e. it is not necessary to explicitly define them before use.

These constants can be used anywhere in the code to represent the values shown.

Name	Value	Meaning
hslTrue	1	True
hslFalse	0	False
hslInfinite		Infinite time-out value.
hslOKOnly	0	Display OK button only.
hslOKCancel	1	Display OK and Cancel button.
hslAbortRetryIgnore	2	Display Abort, Retry and Ignore button.
hslYesNoCancel	3	Display Yes, No and Cancel button.
hslYesNo	4	Display Yes and No button.
hslRetryCancel	5	Display Retry and Cancel button.
hslDefButton1	0	The first button is the default button. hslDefButton1 is the default unless hslDefButton2, or hslDefButton3 is specified.
hslDefButton2	256	The second button is the default button.
hslDefButton3	512	The third button is the default button.
hslError	16	Display Error Message icon.
hslQuestion	32	Display Warning Query icon.
hslExclamation	48	Display Warning Message icon.
hslInformation	64	Display Information Message icon.
hslOK	1	OK button was selected.
hslCancel	2	Cancel button was selected.
hslAbort	3	Abort button was selected.
hslRetry	4	Retry button was selected.
hslIgnore	5	Ignore button was selected.
hslYes	6	Yes button was selected.
hslNo	7	No button was selected.
hslInteger	"i"	The input value is an integer.
hslFloat	"f"	The input value is a float.
hslString	"s"	The input value is a string.
hslRead	"r"	Opens the file for reading. The file must exist.
hslWrite	"w"	Opens an empty file for writing. If the file already exists, its contents is deleted.
hslAppend	"a"	Opens the file for writing at the end of the file. If the file does not exist, a new file is created.
hslHide	1	Hides the window and activates another window.
hslShow	2	Activates the window and displays it in its current size and position.
hslShowMaximized	3	Activates the window and displays it as a maximized window.
hslShowMinimized	4	Activates the window and displays it as a minimized window.
hslSynchronous	1	The execution of the running HSL program will be blocked until the program to execute terminates.
hslAsynchronous	2	The execution of the running HSL program will not be blocked until the program to execute terminates.

hslCSVDelimited	","	Fields in the file data source are delimited by commas (default).
hslTabDelimited	"\t"	Fields in the file data source are delimited by tabs.
hslFixedLength	""	Fields in the text file are of a fixed width.
	"*"	Fields in the file data source are delimited by asterisks. The asterisk can be substituted by any character except the double (") quotation mark.
hslAsciiText	"\n"	Fields in the file data source are delimited by newline characters.
hslCurrent	0	Starts at the current row (default).
hslFirst	1	Starts at the first row.
hslLast	2	Starts at the last row.
HSL_RUNTIME		Parser constant, always defined at run-time, but not at edit-time (see remark below).

Remarks

The Parser constant HSL_RUNTIME will always be defined at run-time, but not at edit-time, i.e the parsing at edit-time can be sped up by using the predefined constant HSL_RUNTIME as follows:

Split the HSL library into two files. The first file defines the library functions with an empty implementation. The second file, which is included by the first file, provides the real implementation of all functions defined in the first file. The implementation being used at edit-time and at run-time, respectively, is controlled by the predefined constant HSL_RUNTIME.

Example

```
#ifndef __HSLMthLib_hsl__
#define __HSLMthLib_hsl__ 1

// #define MTH_DEVELOP 1

#ifdef MTH_DEVELOP
#ifndef HSL_RUNTIME
#define HSL_RUNTIME 1
#endif
#endif

// Interface to Math functions
#ifndef HSL_RUNTIME
function MthMin(variable number1, variable number2)  {}
function MthMax(variable number1, variable number2)  {}
function MthRound(variable number, variable numDecimalPlaces) {}
function MthSin(variable number)  {}
function MthCos(variable number)  {}
function MthTan(variable number)  {}
// ...
#endif

// Implementation of Math functions
#ifdef HSL_RUNTIME
#include "HSLMthLibImpl.hsl"
#endif

#endif
```

Variable scope

Variable Scope

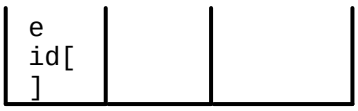
HSL has two main scopes: global and local. If a variable is declared outside of any function definition, it is a global variable, and its value is accessible and modifiable throughout the program. If a variable is declared inside of a function definition, it is a local variable. It is created and destroyed every time the function is executed; it cannot be accessed by anything outside the function.
A local variable can have the same name as a global variable, but it is entirely distinct and separate. Consequently, changing the value of one variable has no effect on the other. Inside the function in which the local variable is declared, only the local version has meaning.

Passing formal parameters

Passing Formal Parameters

When a parameter Is passed to a function by value, a separate copy of that parameter is made, a copy that exists only inside the function. If, on the other hand, a parameter is passed by reference, and the function changes the value of that parameter, it is changed everywhere in the program.

Overview of passing formal parameters		
Type (type)	**By value**	**By reference**
sequence	yes	yes
variable	yes	yes
string	yes	yes
device	NA	always
dialog	NA	always
object	NA	always
timer	NA	always
event	NA	always
file	NA	always
typ	NA	always



The following example illustrates the use of passing formal parameters by value:

```
function AddSequence(
     variable index,      // index of source sequence
     string seqBaseName,  // base name of source sequence
     sequence targetSeq)  // target sequence (copy)
{
     string seqName;      // name of source sequence
     sequence sourceSeq;  // source sequence
     variable currentPos; // current position
     seqName = seqBaseName + IStr(index);
     sourceSeq = ML_STAR.GetSequence(seqName);
     for (currentPos = sourceSeq.SetCurrentPosition(1);
          currentPos != 0;
          currentPos = sourceSeq.Increment(1))
     {
          targetSeq.Add(sourceSeq.GetLabwareId(),sourceSeq.GetPositionId());
     }
     return(targetSeq);
}

method Main()
{
     variable index;
     variable numberOfPlates;
     sequence plates;
     numberOfPlates = InputBox("Enter number of plates:", "Get Input", "i");
     for (index = 1; index <= numberOfPlates; index++)
          plates = AddSequence(index, "Plate_", plates);
}
```

The efficiency of the same example can be improved by passing the formal parameter of the type `sequence` by reference:

```
function AddSequence(
     variable index,      // index of source sequence
     string seqBaseName,  // base name of source sequence
     sequence& targetSeq) // target sequence (reference)
{
     string seqName;      // name of source sequence
     sequence sourceSeq;  // source sequence
     variable currentPos; // current position
     seqName = seqBaseName + IStr(index);
     sourceSeq = ML_STAR.GetSequence(seqName);
     for (currentPos = sourceSeq.SetCurrentPosition(1);
          currentPos != 0;
          currentPos = sourceSeq.Increment(1))
     {
          targetSeq.Add(sourceSeq.GetLabwareId(),sourceSeq.GetPositionId());
     }
}

method Main()
{
     variable index;
     variable numberOfPlates;
     sequence plates;
     numberOfPlates = InputBox("Enter number of plates:", "Get Input", "i");
     for (index = 1; index <= numberOfPlates; index++)
          AddSequence(index, "Plate_", plates);
```

```
}
```


Returning a value

Returning a Value

The `return` statement is used to stop the execution of a function and to return an optional scalar or aggregate value to the calling context of the function. The type of an optional scalar return value can be `variable`, `string` or `sequence` and the type of an optional aggregate return value can be one of the following : `variable`, `string`, `sequence`, `dialog`, `object`, `timer`, `event` or `file`.
The following example illustrates the use of the return statement:

```
function AddSequence(
     variable index,      // index of source sequence
     string seqBaseName,  // base name of source sequence
     sequence targetSeq)  // target sequence (copy)
{
     string seqName;      // name of source sequence
     sequence sourceSeq;  // source sequence
     variable currentPos; // current position
     seqName = seqBaseName + IStr(index);
     sourceSeq = ML_STAR.GetSequence(seqName);
     for (currentPos = sourceSeq.SetCurrentPosition(1);
          currentPos != 0;
          currentPos = sourceSeq.Increment(1))
     {
          targetSeq.Add(sourceSeq.GetLabwareId(), sourceSeq.GetPositionId());
     }
     return(targetSeq);
}
```

Overview of returning values from a function		
Type(type)	**By value**	**By reference**
variable	always	NA
sequence	always	NA
string	always	NA
device	NA	NA
dialog	NA	NA
object	NA	NA
timer	NA	NA
event	NA	NA
file	NA	NA
type	alway	NA

id[]	s	

Special characters

Special Characters

HSL provides special characters that allow for the inclusion of characters which cannot be directly typed into strings. Each of these characters begins with a backslash. The backslash is an escape character used to inform the HSL parser that the next character is special.

Escape Sequence	Character
\n	Line feed (newline)
\t	Horizontal tab
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash

Notice that since the backslash itself is used as the escape character, it is not possible to type a backslash directly within a string. For any backslash to be included in a string, two of them together (\\) must be typed.

```
file f;
string mode;
mode = hslWrite;
f.Open("c:\\temp\\test.hsl", mode);
f.WriteString("c:\\temp\\test.hsl");
```


Using message boxes

Using Message Boxes

Message Box

The message box function has two required arguments: the text string to be displayed to the user and the title of the message box. Per default the message box provides an OK button so the user can close it and it is modal, i.e., the user must close the message box before continuing.
An optional third parameter enables the specification of any combination of predefined icons and buttons (see MessageBox).

```
variable type, answer;
string message, title;

message = "Are you sure you want to abort the method ?";
title = "Method : CreateElisa.hsl";
type = hslYesNo | hslExclamation;
```

```
answer = MessageBox(message, title, type);
if (hslYes == answer)
     abort;
```

Input Box

The input box provides a text field in which the user can type an answer in response to the prompt. This box has an OK button and is modal. The user must enter a value and close it before continuing.

```
string user;
user = InputBox("Enter your name here:", "Input Box", "s");
```


Using the Shell function

Using the Shell Function

The Shell function has three arguments, the program to be started (including any command-line options), the style of the window in which the program is to be run (for graphical user interface programs) and the concurrency of the program to start. This means that a program started with concurrency 'hslAsynchronous' might not finish executing before the statements following the Shell Function are executed.

The following examples illustrate the use of the Shell Function:

Example 1

```
method Main()
{
     variable pgmIndex;    // program index
     variable retVal;      // return value
     variable windowStyle; // window style
     variable concurrency; // concurrency
     string pgmName;       // the name of the program to execute
     string cmdLine;       // command-line options
     string pathname;      // the name of the program to execute and
                           // any command-line options

     windowStyle = hslShow;
     concurrency = hslSynchronous;
     pgmName = "C:\\PROGRAM FILES\\HAMILTON\\BIN\\HXRUN.EXE ";
     cmdLine = "/T C:\\PROGRAM FILES\\HAMILTON\\METHODS\\T092";
     pathname = pgmName + cmdLine;

     for (pgmIndex = 0; pgmIndex < 4; pgmIndex++)
     {
          Trace("Running program ", pathname + IStr(pgmIndex));
          retVal = Shell(pathname + IStr(pgmIndex), windowStyle, concurrency);
          if (retVal == 0)
          {
               string errorMessage;
               errorMessage = "Failed to start program ";
               errorMessage = errorMessage + pathname;
               errorMessage = errorMessage + IStr(pgmIndex);
               MessageBox(errorMessage, "ERROR", hslError);
```

```
                break;
            }
        }
    }
```

Example 2

```
function Fork(string programs[])
{
    variable i(0);                          // program index
    variable windowStyle(hslShow);          // window style
    variable concurrency(hslAsynchronous); // concurrency
    event events[];                         // synchronization events
    variable retVal(0);

    Trace("+Fork");

    events.SetSize(programs.GetSize());

    for (i = 0; i < programs.GetSize(); i++)
    {
        Trace("Starting program ", programs[i]);
        retVal = Shell(programs[i], windowStyle, concurrency, events[i]);
        if (0 == retVal)
        {
            Trace("Shelling out program ", programs[i], " failed");
            events[i].SetEvent();
        }
    }

    Trace("-Fork");

    return(events);
}

function Join(event events[])
{
    variable i(0);        // event index
    variable timeout(60); // timeout in seconds

    Trace("+Join");

    for (i = 0; i < events.GetSize(); i++)
    {
        Trace("Waiting for event ", events[i]);
        if (0 == events[i].WaitEvent(timeout))
        {
            Trace("Waiting for event ", events[i], " failed");
        }
        timeout = 10;
    }

    Trace("-Join");
}

method Main()
{
    variable i(0);          // program index
    variable nubrOfPgms(4); // number of programs to execute asynchronously
    string programs[](4);   // names of programs to execute asynchronously
    event events[](4);      // synchronization events
    string cmdLine;         // command-line options
    string pathname;        // name of program to execute and any
                            // command-line options
```

```
     string pgmName;         // the name of the program to execute

     Trace("+Main");

     pgmName = "C:\\PROGRAM FILES\\HAMILTON\\BIN\\HXRUN.EXE ";
     cmdLine = "/t C:\\PROGRAM FILES\\HAMILTON\\METHODS\\T0387_";
     pathname = pgmName + cmdLine;

     for (i = 0; i < nubrOfPgms; i++)
          programs[i] = pathname + IStr(i+1);

     events = Fork(programs);

     // ...

     Join(events);

     // ...

     Trace("-Main");
}
```


Using dynamic arrays

Using Dynamic Arrays

Understanding Arrays

A dynamic array is a variable that can contain multiple values. Arrays are useful when a number of values of the same type need to be stored, but the number of values is unknown of if it is not desired to create individual variables to store them all.
Loops, together with a couple of special functions can be used for working with dynamic arrays, to assign values to or to retrieve values from the various elements of a dynamic array.

Creating Arrays

Two types of arrays can be created in HSL: fixed-size arrays (vectors) and dynamic arrays. A fixed-size array has a fixed number of elements, and is useful only when the exact number of elements in the array is known when the code is written. Since this is a rare occasion, most of the time dynamic arrays are created.
Fixed-size arrays can be of data type **`char`**, **`short`**, **`long`**, **`float`** and dynamic arrays can be of data type **`variable`**, **`sequence`**, **`string`**, **`dialog`**, **`object`**, **`timer`**, **`event`** and **`file`**. The data type for an array specifies the data type for each element of the array; e.g., each element of an array of type **`string`** can contain a **`string`** value. The following code fragment declares a dynamic array variable of the type **`string`**:

```
string barcodes[];
```

Once a dynamic array has been declared, the array can be resized by using theSetSize() element function. To resize the dynamic array, a value for the upper bound has to be provided. The upper bound of an array refers to the ending index for the array.

```
barcodes.SetSize(10);
```

Assigning one Dynamic Array to Another:
If two dynamic arrays have the same data type, one array can be assigned to another. Assigning one array to another of the same type copies all elements from the first array and stores them to the second array.

For example, the following code fragment assigns one string array to another:

```
string barcodes_1[], barcodes_2[];
//...
barcodes_1 = ML_STAR.LoadCarrier();
barcodes_2 = barcodes_1;
```

This type of assignment works only for dynamic arrays of the *same* type. The two arrays must both be dynamic arrays, and they must be declared as the exact same type: if one is type **`string`**, the other must be type **`string`**. It cannot be of the type **`variable`** or any other data type. If one array's elements are to be assigned to an array of a different type, a loop must be created and each element must be assinged on at a time.

Returning an Array from a Function:
In addition to assigning one array variable to another, a function can be called which returns an array and assign that to another array, as in the following code fragment:

```
function Get3DPoint()
{
     string prompt;
     variable index, point[];
     prompt = "Enter x"
     for (index = 1; index <= 3; index++)
          point.AddAsLast(InputBox(prompt + IStr(index), "Get Input", "i"));
     return point;
}

variable point[];
point = Get3DPoint();
```

Passing an Array to a Function:
An array can be declared in one function and then passed along to another function to be modified. The function that modifies the array does not necessarily need to return an array. Arrays are always passed by reference. When the second function modifies the array, it modifies the original array. Therefore, when the execution returns to the first procedure, the array variable refers to the modified array.

```
function Trace3DPoint(variable point[])
{
     variable index;
          for (index = 0; index < point.GetSize(); index++)
               Trace("x", IStr(index), " = ", point[index]);
}

variable point[];
point = Get3DPoint();
Trace3DPoint(point);
```

Remarks:

- Arrays of type **`device`** cannot be created.
- Array elements created by the function SetSize() are owned by the array.
- Array elements of type **`variable`**, **`string`** or **`sequence`** which are set or added by one of the functions SetAt() or AddAsLast() are passed by value. The same is true when assigning one array of these types to another.
- Array elements of type **`variable`**, **`string`** or **`sequence`** which are get by one of the functions GetAt() or Operator[ ]() are returned by value. If the subscript operator is used to get a target object for a call to an element function, the object is returned by reference. I.e. the expression stringArr[i] is equivalent to stringArr.GetAt(i), the expression stringArr[i].MakeUpper() is equivalent

to stringArr.ElementAt(i).MakeUpper(). Whereas the first expression returns a copy of the element at the desired index, the second returns a temporary reference to the element pointer within the array. An example:

This will work as expected:

```
sequence sequenceArr[];
sequenceArr.SetSize(10);
// the following line adds a labware item to the ssequence sequenceArr[0]
sequenceArr[0].Add("nunc_96_flat_l1", "1");
```

- Array elements of the types **dialog**, **object**, **timer**, **event** or **file** which are set or added by one of the functions SetAt() or AddAsLast() are passed by reference. The same is true when assigning one array to another.
- Array elements of the types **dialog**, **object**, **timer**, **event** or **file** which are get by one of the functions GetAt() or Operator[ ]() are returned by reference. The same is true when assigning one array to another.
- Array elements of the type **device** which are set or added by one of the functions SetAt() or AddAsLast() are rejected. The same is true when assigning one array of these types to another.
- When passing an array as parameter to a function, the array is always passed by reference. The same is true when passing objects of the types **device**, **dialog**, **object**, **timer**, **event** or **file**.
- When returning an array containing references to local objects (**dialog**, **object**, **timer**, **event**, **file**) from a function, and later accessing the array elements form outside of the function, a runtime error will be reported. An example:

This will not work:

```
function f()
{
     timer t;
     timer timerArr[];
     timerArr.AddAsLast(t);
     return(timerArr);
}

method m()
{
     timer timerArr[];
     timerArr = f();
     timerArr[0].SetTimer(10); /* error: failed to set timer */
}
```

This would work:

```
function f()
{
     timer timerArr[];
     timerArr.SetSize(1);
     return(timerArr);
}

method m()
{
     timer timerArr[];
     timerArr = f();
     timerArr[0].SetTimer(10);
}
```

Overview of setting/getting elements to/from an array:

Type	By value	By reference
variable	always	NA
sequence	always	NA
string	always	NA
device	NA	NA
dialog	NA	always
object	NA	always
timer	NA	always
event	NA	always
file	NA	always

More examples using dynamic arrays:

1) Array of strings:

```
method Main()
{
     variable index;
     variable upperBound;
     string barcodes[];

     ML_STAR.Initialize( "9f2777b5_cdb5_11d3_809708005ad23dc7" );

     barcode = ML_STAR.LoadCarrier( "a41b01f4_d22c_11d3_809b08005ad23dc7" );

     upperBound = barcodes.GetSize()- 1;
     for (index = 0; index <= upperBound; index++)
     {
          Trace("barcodes[ ", index, "] = ", barcodes[index]);
          //...
     }
}
```

2) Array of variables and timers:

```
function WaitForMultipleTimers(timer ta[])
{
     variable index;
     for (index = 0; index < ta.GetSize(); index++)
     {
          ta[index].WaitTimer();
          Trace("timer[", IStr(index), "] expired");
     }
}

function SetMultipleTimers(timer timers[], variable timeouts[])
{
     variable index;
     if (timers.GetSize() != timeouts.GetSize())
          return(0);
     for (index = 0; index < timers.GetSize(); index++)
          timers[index].SetTimer(timeouts[index]);
     return(1);
}
```

```
method Main()
{
     variable index, timeouts[];
     timer timers[];

     timers.SetSize(10);
     timeouts.SetSize(timers.GetSize());

     for (index = 0; index < timeouts.GetSize(); index++)
          timeouts[index] = 2*index;

     SetMultipleTimers(timers, timeouts);
     //...
     WaitForMultipleTimers(timers);
     //...
}
```


Using conditional compilation

Using Conditional Compilation

Conditional directives are used to include or exclude blocks of code from a source file, based on the definition of a single identifier. Conditional directives are useful in many aspects of development, among them:
They can be used to selectively include or exclude source file.
They can be used to place a unique include guard around the contents of an included file.

The following example demonstrates the use of conditional directives:
File MyMethod.hsl:

```
#ifndef __utils_hsl__       // redundant include guard, avoid reopening an
reprocessing
     #include "utils.hsl"   //incule file line by line in its entirety
searching for
#endif                      // the trailing #endif

method MyMethod()
{
     //...
}
```

File Utils.hsl:

```
#ifndef __utils_hsl__   /* include guard */
#define __utils_hsl__ 1

function f()
{
     //...
}
//...
#endif
```

Using Automation

Using Automation

Using Automation, components can be integrated in an HSL method in a way that appears seamless to the user. Working with objects through Automation is very similar to working with objects in HSL itself - the user sets and retrieves the object's properties and applies its functions.
To use Automation in HSL, perform these steps, which are explained by an example:

1.) Consult the component's object library.

2.) Create an instance of a class that defines an object in the component's object library by using its `CreateObject()` function and assign it to an object variable.

```
// declare objects

object outlook;

object mail;


// create objects

outlook.CreateObject("Outlook.Application");

mail = outlook.CreateItem(outlook.olMailItem);
```

3.) Work with the new object by setting its properties and applying its functions.

```
// use objects

mail.To = "bandenmatten@hamilton.ch";

mail.Subject = "Subject of mail message";

mail.Body = "Body of mail message";

mail.Send();
```

4.) Release the memory that was allocated to the object variable by using its `ReleaseObject()` function.

```
// release objects
mail.ReleaseObject();
outlook.ReleaseObject();
```

Remarks

HSL supports the automation types listed in Automation Types.
To intercept the events launched by an Automation component in HSL see Working with Events.

Working with Events

Working with Events

HSL can automatically handle asynchronous notifications from Automation objects through events if the Automation server uses the connection point method for event notification. HSL makes it easy to develop methods that can handle these notifications. HSL reads the component type library and automatically connects to the default outgoing dispatch interface.
To intercept the events launched by an Automation component in HSL, perform these steps, which are explained by an example:

1.) Consult the component's object library.

2.) Create an instance of a class that defines an object in the component's object library by using its **`CreateObject()`** function and assign it to an object variable. Set the value of the parameter **`withEvents`** to **`hslTrue`**.

```
// declare objects

variable withEvents(hslTrue);

object outlook;


// create objects

outlook.CreateObject("Outlook.Application", withEvents);
```

3.) Create an event handler function. All the event handler functions must reside in the same namespace as the object sourcing events. All the event handler functions associated with an object sourcing events will have the object name followed by an underscore as a prefix. The signature of the event handler function must conform to the signature of the associated function which is defined in the component's object library.

```
function outlook_ItemSend(object item, variable cancel)
{
     // write code to handle event here
}
```

Remarks

HSL supports the automation types listed in Automation Types.
HSL does not support events having **in/out** or **out** parameters (event will be ignored).

Understanding error handling

Understanding Error Handling

The error handling mechanism is based on the `onerror goto` statement and a global object `err` of type `error`. The `onerror goto` statement enables or disables a user-defined error handler. If an error handler is enabled, the program execution is continued with the error handler if an error occurs. However, if

no error handler is enabled, then the program execution is aborted if an error occurs. There are three types of the `onerror goto` statement: `onerror goto id`, `onerror goto 0` and `onerror resume next`.
The `onerror` statement syntax can have any of the following forms:

Statement	Description
`onerror goto id`	Enables the error handler that starts at the label specified. If a runtime error occurs, control branches to the label, activating the error handler. The specified label must be in the same function or method as the `onerror` statement; otherwise, a syntax error occurs. The statement has no effect within an error handler.
`onerror resume next`	Specifies that, when a runtime error occurs, the control goes to the statement immediately following the statement where the error occurred where the execution continues. The `err` object's properties are reset to zero or zero-length strings (“”) after an `onerror resume next` statement. The `err.Clear()` function can be used to explicitly reset `err`.
`onerror goto 0`	Disables any enabled error handler in the current function or method. The statement has no effect within an error handler.
`resume next`	Resumes execution after an error handler is finished. If the error occurred in the same function as the error handler, the execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called function, the execution resumes with the statement immediately following the statement that last called out of the function containing the error-handler (or `onerror resume next` statement). The statement has no effect outside of an error handler.

The global object `err` of type `error` is used to transfer error information from a location in the program execution to an error handler. Objects of type `error` represent run time errors and cannot be instantiated. For each valid program exists exactly one global error object with the name `err`. Objects of type `error` have pre-defined element functions. Their names are reserved. Functions to manipulate the global object `err` of type `error` are listed in element functions.

Example

```
function f()
{
     //...
     onerror goto ErrorHandler;
     //...
     return;

     ErrorHandler :
     {
          onerror resume next;
          //...
          resume next;
     }
}

method Main()
{
     variable v;
     onerror goto UnhandledError;
     //...
     f();
     //...
```

```
     return;

     UnhandledError :
     {
          //...
          err.Clear();
          abort;
     }
}
```


File handling code examples

File Handling Code Examples

Remark:
For a good design when creating the individual, programmer-defined database, follow these steps:

One. Rows of cells with only data in them should be organized in a consistent manner so that results for each potential field in the database lines up under each other in the table.
Two. The rows of data should have a heading row at the top. This saves a lot of time when compared to naming each column later used in the database table individually. Follow good naming conventions and avoid spaces or special characters in the field name.
Three. A column with unique row identifiers (key field) should be built into the table.

Example 1: Microsoft Excel File

Input: **Excel** file test.xls, table Table1

FName	LName	PNumber
Beni	Andenmatten	255
Rene	Keller	256
Thomas	Benz	257
Joerg	Jenal	413
Linus	Jegher	257

```
method Main()
{
     variable rc(0), row (0);
     variable fname("FName");     // column header, first name
     variable lname("LName");     // column header, last name
     variable pnumber("PNumber"); // column header, phone number

     variable xlsDataSource("test.xls Table1$");
     variable xlsSQLSelect1("SELECT LName FROM [Table1$] WHERE LName LIKE 'J
     %'");
     variable xlsSQLSelect2("SELECT * FROM [Table1$]");
     file f;

     Trace("+Main");

     // add the LName-column to the record
     rc = f.AddField(lname, lname, hslString);
     Trace("AddField, rc = ", rc);
```

```
	// query all last names beginning with J
	rc = f.Open(xlsDataSource, hslAppend, xlsSQLSelect1);
	Trace("Open, rc = ", rc);

	// trace all last names beginning with J
	f.Seek(0, hslFirst);
	while (!f.Eof())
	{
		rc = f.ReadRecord();
		Trace("ReadRecord, rc = ", rc);
		Trace("lname = ", lname);
	}

	// add the FName-column to the record
	rc = f.AddField(fname, fname, hslString);
	Trace("AddField, rc = ", rc);

	// add the PNumber-column to the record
	rc = f.AddField(pnumber, pnumber, hslInteger);
	Trace("AddField, rc = ", rc);

	// requery all records
	rc = f.Open("", hslAppend, xlsSQLSelect2);
	Trace("Open, rc = ", rc);

	// trace all records
	f.Seek(0, hslFirst);
	while (!f.Eof())
	{
		rc = f.ReadRecord();
		Trace("ReadRecord, rc = ", rc);
		Trace("fname = ", fname);
		Trace("lname = ", lname);
		Trace("pnumber = ", pnumber);
	}

	// add a new record to the adress list
	fname = "Stephan";
	lname = "Gieriet";
	pnumber = 300;
	rc = f.WriteRecord();
	Trace("WriteRecord, rc = ", rc);

	// trace all records with phone number 257
	for (row = f.Seek(0, hslFirst, "PNumber = 257");
		row != 0;
		row = f.Seek(0, hslCurrent, "PNumber = 257"))
	{
		rc = f.ReadRecord();
		Trace("ReadRecord, rc = ", rc);
		Trace("fname = ", fname);
		Trace("lname = ", lname);
		Trace("pnumber = ", pnumber);
	}

	// close the file data source
	rc = f.Close();
	Trace("Close, rc = ", rc);

	Trace("-Main");
}
```

Output: Excel file test.xls, table Table1

FName	Lname	PNumber
Beni	Andenmatten	255
Rene	Keller	256
Thomas	Benz	257
Joerg	Jenal	413
Linus	Jegher	257
Stephan	Gieriet	300

Example 2: Text File

Input: **Text** file test.txt:

"FName",	"LName",	"PNumber"
"Beni",	"Andenmatten",	255
"Rene",	"Keller",	256
"Thomas",	"Benz",	257
"Joerg",	"Jenal"	257

```
            ,

method Main()
{
     variable rc(0), row(0);
     variable fname("FName");.    // column header, first name
     variable lname("LName");     // column header, last name
     variable pnumber("PNumber"); // column header, phone number

     variable txtDataSource("test.txt");
     variable txtSQLSelect1("SELECT LName FROM [test#txt] WHERE LName LIKE 'J
     %'");
     variable txtSQLSelect2("SELECT * FROM [test#txt]");

     file f;

     Trace("+Main");

     // add the LName-column to the record
     rc = f.AddField(lname, lname, hslString);
     Trace("AddField, rc = ", rc);

     // query all last names beginning with J
     rc = f.Open(txtDataSource, hslAppend, txtSQLSelect1);
     Trace("Open, rc = ", rc);

     // trace all last names beginning with J
     f.Seek(0, hslFirst);
     while (!f.Eof())
     {
          rc = f.ReadRecord();
          Trace("ReadRecord, rc = ", rc);
          Trace("lname = ", lname);
     }

     // add the FName-column to the record
     rc = f.AddField(fname, fname, hslString);
     Trace("AddField, rc = ", rc);

     // add the PNumber-column to the record
     rc = f.AddField(pnumber, pnumber, hslInteger);
     Trace("AddField, rc = ", rc);

     // requery all records
     rc = f.Open("", hslAppend, txtSQLSelect2);
     Trace("Open, rc = ", rc);

     // trace all records
     f.Seek(0, hslFirst);
     while (!f.Eof())
     {
          rc = f.ReadRecord();
          Trace("ReadRecord, rc = ", rc);
          Trace("fname = ", fname);
          Trace("lname = ", lname);
          Trace("pnumber = ", pnumber);
     }

     // add a new record to the adress list
     fname = "Stephan";
     lname = "Gieriet";
     pnumber = 300;
     rc = f.WriteRecord();
     Trace("WriteRecord, rc = ", rc);
```

```
     // trace all records with phone number 257
     for ( row = f.Seek(0, hslFirst, "PNumber = 257");
          row != 0;
          row = f.Seek(0, hslCurrent, "PNumber = 257"))
     {
          rc = f.ReadRecord();
          Trace("ReadRecord, rc = ", rc);
          Trace("fname = ", fname);
          Trace("lname = ", lname);
          Trace("pnumber = ", pnumber);
     }

     // close the file data source
     rc = f.Close();
     Trace("Close, rc = ", rc);

     Trace("-Main");
}
```

Output: Text file test.txt

"FName",	"LName",	"PNumber"
"Beni",	"Andemattten",	255
"Rene",	"Keller",	256
"Thomas",	"Benz",	257
"Joerg",	"Jena	257

```
                l
                "
                ,
 "Stephan",     "    300
                G
                i
                e
                r
                i
                e
                t
                "
                ,
```

Example 3: Microsoft Jet Database

Input: **Microsoft Access** data source Nordwind.mdb
Table: Employees

Employee ID	Last Name	First Name	Title	Birth Date
1	Davolio	Nancy	Sales Representative	08.12.48
2	Fuller	Andrew	Vice President, Sales	19.02.52
3	Leverling	Janet	Sales Representative	30.08.63
4	Peacock	Margaret	Sales Representative	19.09.37
5	Buchanan	Steven	Sales Manager	04.03.55
6	Suyama	Michael	Sales Representative	02.07.63
7	King	Robert	Sales Representative	29.05.60
8	Callahan	Laura	Inside Sales Coordinator	09.01.58
9	Dodsworth	Anne	Sales Representative	27.01.66

```
method Main()
{
     file f;
     variable fname;  // First Name
     variable lname;  // Last Name
     variable bdate;  // Birth Date
     variable mdbDataSource("\\Temp\\Nordwind.mdb Employees");
     variable accSQLSelect("SELECT * FROM Employees");

     // add the FirstName, LastName and BirthDate-column to the record
     f.AddField("FirstName", fname, hslString);
     f.AddField("LastName", lname, hslString);
     f.AddField("BirthDate", bdate, hslString);

     // query all records
     f.Open(mdbDataSource, hslAppend, accSQLSelect);

     // trace all records
     f.Seek(0, hslFirst);
     while (!f.Eof())
     {
          f.ReadRecord();
          Trace("fname = ", fname);
          Trace("lname = ", lname);
          Trace("bdate = ", bdate);
```

```
    }

    // add a new record to the table Employees
    fname = "Beni";
    lname = "Andenmatten";
    bdate = "28.11.63";
    f.WriteRecord();

    // close the data source
    f.Close();
}
```

Output: Table: Employees

Employee ID	Last Name	First Name	Title	Birth Date
1	Davolio	Nancy	Sales Representative	08.12.48
2	Fuller	Andrew	Vice President, Sales	19.02.52
3	Leverling	Janet	Sales Representative	30.08.63
4	Peacock	Margaret	Sales Representative	19.09.37
5	Buchanan	Steven	Sales Manager	04.03.55
6	Suyama	Michael	Sales Representative	02.07.63
7	King	Robert	Sales Representative	29.05.60
8	Callahan	Laura	Inside Sales Coordinator	09.01.58
9	Dodsworth	Anne	Sales Representative	27.01.66
10	Andenmatten	Beni		28.11.63

Example 4: Fundamental Microsoft Jet SQL for HSL

This method demonstrates the basic mechanims of using SQL (Structured Query Language) to work with data in HSL. It delves into using SQL to

- Retrieving Records
- Restricting the Result Set
- Sorting the Result Set
- Using Aggregate Functions to Work with Values
- Updating Records in a Table
- Grouping Records in a Result Set
- Inserting Records into a Table
- Deleting Records from a Table

This method is a good place to start for inexperienced users for querying data with SQL in a database. Step over the statements in the method and read the comments to learn the most basic and most often used SQL statements. As a data source, the method uses the table shown below.
This table is stored in an ASCII text file, a Microsoft® Excel file and a Microsoft® Access database. The method demonstrates how to access the table in each of these databases.
Input: **Text** file FundamentalSQL.txt or **Excel** file FundamentalSQL.xls, table PipettingData

Labware ID	Position ID	Barcode	OD	Volume
Plate_1	1	B0003	1.2	0
Plate_1	2	A0003	0.8	0

Plate_1	3	C0002	0.9	0
Plate_1	4	B0002	1.5	0
Plate_1	5	A0002	1	0
Plate_1	6	C0001	1.1	0
Plate_1	7	B0001	0.7	0
Plate_1	8	A0001	0.5	0
Plate_2	1	B0006	1.1	0
Plate_2	2	A0006	1.8	0
Plate_2	3	C0005	0.9	0
Plate_2	4	B0005	1	0
Plate_2	5	A0005	1	0
Plate_2	6	C0004	1.1	0
Plate_2	7	B0004	0.7	0
Plate_2	8	A0004	0.9	0

```
method Main()
{
     file f;
     variable delim(", ");
     variable labwareID;
     variable positionID;
     variable barcode;
     variable od;
     variable volume;
     variable numberOfRecords;
     variable totalOD;
     variable averageOD;
     variable minimumOD;
     variable maximumOD;

     // RETRIEVING RECORDS

     // The most basic and most often used SQL statement is the
     // SELECT statement. SELECT statements are the workhorses of
     // all SQL statements, and they are commonly referred to as
     // select queries. Use the SELECT statement to retrieve
     // data from the database tables, and the results are usually
     // returned in a set of records (or rows) made up of any number
     // of fields (or columns). It must be designated which table or
     // tables to select from with the FROM clause. The basic
     // structure of a SELECT statement is:
     // SELECT field list FROM table list

     // To select all the fields from a table, use an asterisk (*).
     // For example, the following statement selects all the fields
     // and all the records from the table:

     f.AddField("Labware ID", labwareID, hslString, 20);
     f.AddField("Position ID", positionID, hslString, 20);
     f.AddField("Barcode", barcode, hslString, 20);
     f.AddField("OD", od, hslFloat, 0);
     f.SetDelimiter(hslCSVDelimited);
     f.Open("FundamentalSQL.txt", hslRead,
               "SELECT * FROM [FundamentalSQL#txt]");
     Trace(">>>>", "SELECT * FROM [FundamentalSQL#txt]");
     while (!f.Eof())
     {
          f.ReadRecord();
          Trace(labwareID, delim, positionID, delim, barcode, delim, od);
```

```
}
f.RemoveFields();

// To limit the fields retrieved by the query, simply use the
// field names instead. For example:

f.AddField("Position ID", positionID, hslString, 20);
f.AddField("Barcode", barcode, hslString, 20);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT [Position ID], Barcode FROM [FundamentalSQL#txt]");
labwareID = positionID = barcode = od = "";
Trace(">>>>", "SELECT [Position ID], Barcode FROM [FundamentalSQL#txt]");
while (!f.Eof())
{
     f.ReadRecord();
     Trace(labwareID, delim, positionID, delim, barcode, delim, od);
}
f.RemoveFields();

// To designate a different name for a field in the result set,
// use the AS keyword to establish an alias for that field.

f.AddField("Position Number", positionID, hslString, 20);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT [Position ID] AS [Position Number] FROM
          [FundamentalSQL#txt]");
labwareID = positionID = barcode = od = "";
Trace(">>>>", "SELECT [Position ID] AS [Position Number] FROM
[FundamentalSQL#txt]");
while (!f.Eof())
{
     f.ReadRecord();
     Trace(labwareID, delim, positionID, delim, barcode, delim, od);
}
f.RemoveFields();

// RESTRICTING THE RESULT SET

// More often than not, it is not desired to retrieve all records
// from a table rather than only a subset of those records
// based on some qualifying criteria. To qualify a SELECT
// statement, a WHERE clause must be used, which will allows the
// exact specification which records to retrieve.
// A WHERE clause can contain up to 40 such expressions, and
// they can be joined with the And or Or logical operators.
// Using more than one expression allows to further filter
// out records in the result set.

f.AddField("Labware ID", labwareID, hslString, 20);
f.AddField("Position ID", positionID, hslString, 20);
f.AddField("Barcode", barcode, hslString, 20);
f.AddField("OD", od, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT * FROM [FundamentalSQL#txt] WHERE OD < 0.8 OR 1.2 <
          OD");
labwareID = positionID = barcode = od = "";
Trace(">>>>", "SELECT * FROM [FundamentalSQL#txt] WHERE OD < 0.8 OR 1.2 <
OD");
while (!f.Eof())
{
     f.ReadRecord();
     Trace(labwareID, delim, positionID, delim, barcode, delim, od);
}
f.RemoveFields();
```

```
// SORTING THE RESULT SET

// To specify a particular sort order on one or more fields in
// the result set, use the optional ORDER BY clause. Records
// can be sorted in either ascending (ASC) or descending (DESC)
// order; ascending is the default. Fields referenced in the
// ORDER BY clause do not have to be part of the SELECT statement's
// field list, and sorting can be applied to string and numeric
// values. Always place the ORDER BY clause at the end of the
// SELECT statement.

f.AddField("Labware ID", labwareID, hslString, 20);
f.AddField("Position ID", positionID, hslString, 20);
f.AddField("Barcode", barcode, hslString, 20);
f.AddField("OD", od, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT * FROM [FundamentalSQL#txt] ORDER BY OD DESC, [Position
          ID] ASC");
labwareID = positionID = barcode = od = "";
Trace(">>>>", "SELECT * FROM [FundamentalSQL#txt] ORDER BY OD DESC,
[Position ID] ASC");
while (!f.Eof())
{
     f.ReadRecord();
     Trace(labwareID, delim, positionID, delim, barcode, delim, od);
}
f.RemoveFields();

// USING AGGREGATE FUNCTIONS TO WORK WITH VALUES

// Aggregate functions are used to calculate statistical and
// summary information from data in tables. These functions
// are used in SELECT statements, and all of them take fields
// or expressions as arguments.

// To count the number of records in a result set, use the
// Count function. Using an asterisk with the Count function
// causes Null values to be counted as well.

f.AddField("Number of Records", numberOfRecords, hslInteger, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT Count(*) AS [Number of Records] FROM
          [FundamentalSQL#txt]");
Trace(">>>>", "SELECT Count(*) AS [Number of Records] FROM
[FundamentalSQL#txt]");
f.ReadRecord();
Trace("numberOfRecords = ", numberOfRecords);
f.RemoveFields();

// To find the average value for a column or expression of
// numeric data, use the Avg function.

f.AddField("Average OD", averageOD, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT Avg(OD) AS [Average OD] FROM [FundamentalSQL#txt]");
Trace(">>>>", "SELECT Avg(OD) AS [Average OD] FROM
[FundamentalSQL#txt]");
f.ReadRecord();
Trace("averageOD = ", averageOD);
f.RemoveFields();

// To find the total of the values in a column or expression of
// numeric data, use the Sum function.
```

```
f.AddField("Total OD", totalOD, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT Sum(OD) AS [Total OD] FROM [FundamentalSQL#txt]");
Trace(">>>>", "SELECT Sum(OD) AS [Total OD] FROM [FundamentalSQL#txt]");
f.ReadRecord();
Trace("totalOD = ", totalOD);
f.RemoveFields();

// To find the minimum value for a column or expression, use
// the Min function.

f.AddField("Minimum OD", minimumOD, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT Min(OD) AS [Minimum OD] FROM [FundamentalSQL#txt]");
Trace(">>>>", "SELECT Min(OD) AS [Minimum OD] FROM
[FundamentalSQL#txt]");
f.ReadRecord();
Trace("minimumlOD = ", minimumOD);
f.RemoveFields();

// To find the maximum value for a column or expression, use
// the Max function.

f.AddField("Maximum OD", maximumOD, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT Max(OD) AS [Maximum OD] FROM [FundamentalSQL#txt]");
Trace(">>>>", "SELECT Max(OD) AS [Maximum OD] FROM
[FundamentalSQL#txt]");
f.ReadRecord();
Trace("maximumOD = ", maximumOD);
f.RemoveFields();

// UPDATING RECORDS IN A TABLE

// Updating data in a linked table is not supported by the Text
// ISAM driver. Therefore this example continues with the identical
// table, named PipettingData, in a Microsoft Excel file.

// To update all the records in a table, specify the table name,
// and then use the SET clause to specify the field or fields to
// be changed.

// Note: Updating a record in a table, commonly referred to as action
// query, does not really open the data source. So, the data source
// must not be closed.

f.Open("FundamentalSQL.xls PipettingData", hslAppend,
          "UPDATE PipettingData SET Volume = 100");

// In most cases, the UPDATE statement needs to be qualified with a
// WHERE clause to limit the number of records changed.

f.Open("FundamentalSQL.xls PipettingData", hslAppend,
          "UPDATE PipettingData SET Volume = 120 WHERE OD > 1.2");

// Trace all records

f.AddField("Labware ID", labwareID, hslString, 20);
f.AddField("Position ID", positionID, hslString, 20);
f.AddField("Barcode", barcode, hslString, 20);
f.AddField("OD", od, hslFloat, 0);
f.AddField("Volume", volume, hslFloat, 0);
f.Open("FundamentalSQL.xls PipettingData", hslRead);
```

```
labwareID = positionID = barcode = od = volume = "";
Trace(">>>>", "UPDATE PipettingData SET Volume = 100");
Trace(">>>>", "UPDATE PipettingData SET Volume = 120 WHERE OD > 1.2");
while (!f.Eof())
{
     f.ReadRecord();
     Trace(labwareID, delim, positionID, delim, barcode, delim, od,
     delim, volume);
}
f.RemoveFields();

// GROUPING RECORDS IN A RESULT SET

// The key to grouping records is that one or more fields in each
// record must contain the same value for every record in the
// group. To create a group of records, use the GROUP BY clause
// with the name of the field or fields to group with.

f.AddField("Labware ID", labwareID, hslString, 20);
f.AddField("Number of Records", numberOfRecords, hslInteger, 0);
f.AddField("Average OD", averageOD, hslFloat, 0);
f.Open("FundamentalSQL.txt", hslRead,
          "SELECT [Labware ID], Count(*) AS [Number of Records], Avg(OD)
          AS [Average OD] \
          FROM [FundamentalSQL#txt] GROUP BY [Labware ID]");
labwareID = numberOfRecords = averageOD = "";
Trace(">>>>", "SELECT [Labware ID], Count(*) AS [Number of Records],
Avg(OD) AS \
               [Average OD] FROM [FundamentalSQL#txt] GROUP BY [Labware
               ID]");
while (!f.Eof())
{
     f.ReadRecord();
     Trace(labwareID, delim, numberOfRecords, delim, averageOD);
}
f.RemoveFields();

// INSERTING RECORDS INTO A TABLE

// Deleting data in a linked table is not supported by the Text and
// Excel ISAM driver. In order to delete the record, now insert
// in a table (in this exmple the identical table named
// PipettingData) in a Microsoft Jet database.

// There are essentially two methods for adding records to a table.
// The first is to add one record at a time; the second is to add
// many records at a time. In both cases, use the SQL statement
// INSERT INTO to accomplish the task. INSERT INTO statements are
// commonly referred to as append queries.

// To add one record to a table, use the field list to define
// which fields to put the data in, and then supply the data
// itself in a value list. To define the value list, use the VALUES
// clause.

// Note: Updating a record in a table, commonly referred to as action
// query, does not really open the data source. So, the data source
// must not be closed.

f.Open("FundamentalSQL.mdb PipettingData", hslAppend,
          "INSERT INTO PipettingData VALUES ('Plate_3', '1', 'H0001', 0,
          0)");

f.AddField("Labware ID", labwareID, hslString, 20);
```

```
     f.AddField("Position ID", positionID, hslString, 20);
     f.AddField("Barcode", barcode, hslString, 20);
     f.AddField("OD", od, hslFloat, 0);
     f.AddField("Volume", volume, hslFloat, 0);
     f.Open("FundamentalSQL.mdb PipettingData", hslRead);
     labwareID = positionID = barcode = od = "";
     Trace(">>>>", "INSERT INTO PipettingData VALUES ('Plate_3', '1', 'H0001',
     0, 0)");
     while (!f.Eof())
     {
          f.ReadRecord();
          Trace(labwareID, delim, positionID, delim, barcode, delim, od);
     }
     f.RemoveFields();

     // DELETING RECORDS FROM A TABLE

     // To delete the data currently in a table, use the
     // DELETE statement, which is commonly referred to as a delete
     // query, also known as truncating a table. The DELETE statement
     // can remove one or more records from a table and generally takes
     // this form: DELETE FROM table list

     // The DELETE statement does not remove the table structure, only
     // the data that is currently being held by the table structure.
     // To remove all the records from a table, use the DELETE statement
     // and specify which table or tables from where all records shall
     // be deleted. In most cases, the DELEE statement needs to be
     // qualified with a WHERE clause to limit the number of records
     // to be removed.

     // Note: Deleting a record from a table, commonly referred to as action
     // qurey, does not really open the data source. So, the data source
     // must not be closed.

     f.Open("FundamentalSQL.mdb PipettingData", hslAppend,
               "DELETE FROM PipettingData WHERE [Labware ID] = 'Plate_3'");

     f.AddField("Labware ID", labwareID, hslString, 20);
     f.AddField("Position ID", positionID, hslString, 20);
     f.AddField("Barcode", barcode, hslString, 20);
     f.AddField("OD", od, hslFloat, 0);
     f.Open("FundamentalSQL.mdb PipettingData", hslRead);
     labwareID = positionID = barcode = od = "";
     Trace(">>>>", "DELETE FROM PipettingData WHERE [Labware ID] =
     'Plate_3'");
     while (!f.Eof())
     {
          f.ReadRecord();
          Trace(labwareID, delim, positionID, delim, barcode, delim, od);
     }
     f.RemoveFields();

     return;
}
```

Output: **Excel** file FundamentalSQL.xls, table PipettingData

Labware ID	Position ID	Barcode	OD	Volume
Plate_1	1	B0003	1.2	100
Plate_1	2	A0003	0.8	100
Plate_1	3	C0002	0.9	100

Plate_1	4	B0002	1.5	120
Plate_1	5	A0002	1	100
Plate_1	6	C0001	1.1	100
Plate_1	7	B0001	0.7	100
Plate_1	8	A0001	0.5	100
Plate_2	1	B0006	1.1	100
Plate_2	2	A0006	1.8	120
Plate_2	3	C0005	0.9	100
Plate_2	4	B0005	1	100
Plate_2	5	A0005	1	100
Plate_2	6	C0004	1.1	100
Plate_2	7	B0004	0.7	100
Plate_2	8	A0004	0.9	100

Parallelism code examples

Parallelism Code Examples

Remarks

- The units of parallel execution are functions.
- Any data has to be passed into a function that a new thread is to execute by global variables.
- Threads are synchronized by events, or timers, or by calling the Join function.
- Events are the only objects that allow one thread mutually exclusive access to a resource (data).

Example

Using functions as entry points to new threads:

```
device swap("SystemDeckLayout.lay", "ML_SWAP", hslTrue);
device star("SystemDeckLayout.lay", "ML_STAR", hslTrue);

function SWAPInit()
{
     swap.Initialize("be5b5ed0_1bf7_11d4_b878002035848439");
     return;
}

function STARInit()
{
     star.Initialize("ce5b5ed0_1bf7_11d4_b878002035848439");
     return;
}

method Main()
{
```

```
     variable rc(0);
     variable timeout(2*60);
     variable handle;
     variable handles[ ];
     onerror goto ErrorHandler;

     // initialize STAR and SWAP in parallel using separate threads

     handle = Fork("SWAPInit");
     if (0 == handle)
          err.Raise(0, "Failed to fork SWAPInit");
     handles.AddAsLast(handle);
     handle = Fork("STARInit");
     if (0 == handle)
          err.Raise(0, "Failed to fork STARInit");
     handles.AddAsLast(handle);

     // do anything else in the main thread, before waiting until STAR and
     SWAP have been initialized

     rc = Join(handles, timeout);
     if (0 == rc)
          err.Raise(1, "Failed to join handles");
     return;

     ErrorHandler :
     {
          if (hslAbort == MessageBox( err.GetDescription(), "Error - Main",
          hslError|hslAbortRetryIgnore))
               abort;
          resume next;
     }
}
```


Serial communications code example

Serial Communications Code Example

```
/*
This program demonstrates the serial communication with a RS-232 scanner.
The program is for demonstration purposes only. It is only intended to
prove that cabling is correct, the COM port and the scanner are working.
If the bar code data is displayed on the screen while using this
program, it only demonstrates that the hardware interface and scanner are
working.
The program negotiates with the scanner to use ACK/NAK flow control and
STX/ETX data formating. When the ACK/NAK option is enabled, the executor will
not transmit again unless an ACK (ASCII 006) is received after data
transmission. If a NAK (ASCII 021) is received, the executor will retransmit
the data.
When STX/ETX option is enabled, the executor will transmit a STX (ASCII 002)
character before transmitting the data and an ETX (ASCII 003) character
immediately following a data transmission, and remove a STX immediately
preceding data reception and remove an ETX immediately following a data
reception.
*/

method Main()
{
     file port;               /* communications port */
     variable barcode("");    /* place to read barcodes from the scanner */
```

```
/* Specify the timeout values */

/*
Remarks:
A value of hslInfinite, combined with zero values for both the
ReadTotalTimeoutConstant and ReadTotalTimeoutMultiplier members,
specifies that the read operation is to return immediately
with the characters that have already been received, even if no
characters have been received.

If an application sets ReadIntervalTimeout and ReadTotalTimeoutMultiplier
to hslInfinite and  sets ReadTotalTimeoutConstant to a value greater
than zero and less than hslInfinite, one of
the following occurs when the ReadRecord function is called:
If there are any characters in the input buffer, ReadRecord returns
immediately with the characters in the buffer.
If there are no characters in the input buffer, ReadRecord waits until a
character arrives and then returns immediately.
If no character arrives within the time specified by
ReadTotalTimeoutConstant, ReadRecord times out.
*/

variable ReadIntervalTimeout(hslInfinite);        /* maximum time between
read chars */
variable ReadTotalTimeoutMultiplier(hslInfinite); /* multiplier of
characters */
variable ReadTotalTimeoutConstant(10.000);        /* constant in seconds
*/
variable WriteTotalTimeoutMultiplier(0.000);      /* multiplier of
characters */
variable WriteTotalTimeoutConstant(10.000);       /* constant in seconds
*/

/* Open the port */

port.SetDelimiter(hslAsciiText);
port.AddField(1, barcode, hslString);
port.Open("COM2 9600,S,7,2,ACK/NAK,STX/ETX", hslWrite);

/* Set the timeouts */

SetCommTimeouts(port);

/* Configurate the scanner */

if (!port.WriteString("999999"))   /* enter configuration mode */
{
     MessageBox("WriteString failed", "ERROR", hslError);
     abort;
}
if (!port.WriteString("118414"))   /* enable fast beep */
{
     MessageBox("WriteString failed", "ERROR", hslError);
     abort;
}
if (!port.WriteString("318515"))   /* optional tone 6 */
{
     MessageBox("WriteString failed", "ERROR", hslError);
     abort;
}
if (!port.WriteString("999999"))   /* exit configuration mode */
{
     MessageBox("WriteString failed", "ERROR", hslError);
     abort;
```

```
     }

     /* Read barcodes */

     while (1)
     {
          barcode = "";
          if (!port.ReadRecord())
          {
               MessageBox("ReadRecord failed", "ERROR", hslError);
               abort;
          }
          Trace("barcode = ", barcode);
     }

     return;
}
```