DigitalRoute

APL Reference Guide

Table of Contents

APL Reference Guide

This document describes the Analysis Programming Language, APL, and contains descriptions of all the different APL commands that can be used in MediationZone.

For information about Terms and Acryonyms used in this document, see the Terminology document.

1. Language Characteristics

This chapter covers the key concepts of the APL language.

1.1 Syntax description

All APL functions are described according to the following:

returnType `function` (type1 *`parameter1`* , type2 *`parameter2`*)

Parameter	Description
`parameter1`	A description of parameter1. If necessary, an example is given.
`parameter2`	A description of parameter2. If necessary, an example is given.
Returns	A description of what the function returns. If necessary, an example is given.

Example

If there is a need to explain a function in relation to other functions and/or agents, this is done in an 'Example' clause.

```
//Global variable declaration
int myVar1;

//Function block, entry-point for execution in the Consume state.
consume {
        //Local varaible declaration
        string myVar2;

        // A comment.
        /* This is a comment over several
        lines */

        //Assignments
        myVar1=1;
        myVar2="example";

        //Function call
        myFunction(myVar2);
}


//Function declaration
int myFunction(string myParam) {
        debug(myParam);
        return 0;
}
```

1.2 Variables and Variable Scope

Variables in APL may be either global or local:

- Global - Declared outside functions and function blocks
- Local - Declared in a function or function block

```
// Declaration without initialization
int myInt1;
//Declaration with initialization
int myInt2 = 42;
```

The declaration order of variables is relevant for the initialization in the sense that earlier variables may be used as input to later variables.

Example - Declaration order of global variables

```
// Legal
int f1 = 1;
int f2 = f1 + 1;

// Illegal! The following will not compile!
int f3 = f4 + 1;
int f4 = 1;
```

The scoping rules are as follows:

- Global variables are available in all functions and blocks. Note that they cannot be accessed from another agent or by an external APL script.
- All blocks define a separate scope.
- Any *non-anonymous* block statement (statements within a {} block) defines a scope.

Example - Scope

```
int a; //Variable a is global

consume {
        int b; //Variable b is local to the consume block
    {
        //anonomous block
        int c; //Variable c is local to the consume block;
    }
    for(int d=0;d<10;d++) {
        //Variable d is local to the for-loop
    }

}
```

'Variable hiding' is not allowed. For instance, if a variable `myVar` has been declared in global scope, it is not permitted to declare another variable `myVar` anywhere else in the code.

Example - 'Variable hiding' is not allowed

```
int myVar;
consume {
   if (true){
      int myVar; //illegal! myVar is already declared as a global variable
   }
}
```

Example - 'Variable hiding' is not allowed

```
consume {
   int myVar;
   if (true){
      int myVar; //illegal! myVar already declared before on a higher level
   }
}
```

Example - 'Variable hiding' is not allowed

```
consume {
   if (true) {
      int myVar; //myVar will only be valid within this block
   } myVar = 2; //illegal! myVar does not exist here
   if (true) {
      int myVar; //myVar can be declared again as it is in a separate scope from above
   }
   int myVar; //ok! since it is a separate scope from above
}
```

Final Variables

To prevent variables from being changed, the `final` keyword is used:

```
final int myVar = 100;
```

Note!

Only numeric and string types are supported for final variables.

Persistent Variables

Note!

Applicable for batch workflows only.

All variables are by default reset between workflow activations. However, if a value of a global variable has to be saved between each activation, the persistent keyword can be used. The variable declaration as follows:

```
persistent int myVar = 100;
```

Persistent variables are read from the database between initialize and beginBatch and saved between endBatch and deinitialize. Consequently, any assignment to the persistent variable in initialize only works the first time, once you have a persistent value this will overwrite the value from initialize. Like all other variable types, persistent variable names are unique for each agent within a workflow.

Note!

A persistent variable cannot be altered after it has been entered into the database. It will be kept there until the workflow is removed or the value is overwritten, that is, when the workflow configuration is saved with a new name using the Save As... option.

Example - Persistent variables

```
persistent int counter;

initialize {
  counter = 0;
  debug( "Value in initialize: " + counter );
}

beginBatch {
   counter = counter + 1;
   debug( "Value in beginBatch: " + counter );
}
```

The debug output from `initialize` will read zero each time, while the debug output from `beginBatch` will read the actual value.
First time activated (one input batch):

Value in initialize: 0

Value in beginBatch: 1

Fifth time activated (one input batch):

Value in initialize: 0

Value in beginBatch: 5

1.3 Function Declarations

Functions may be declared and called within the APL code using normal Java syntax.

Example - Declaring functions

```
int addNumbers (int a, int b){
    return a + b;
} consume {
    debug( "1 + 1 = " + addNumbers( 1, 1 ) );
}
```

To declare functions with no return value, the void keyword should be used.

The Synchronized Keyword

In order to support multi-threading, functions updating global variables within realtime workflows must be made thread-safe. To achieve this, the function name is preceded with the `synchronized` keyword:

Example - Using the synchronized keyword

```
synchronized int updateSeqNo(){
   seqNo = seqNo +1;
   return seqNo;
}
```

It is possible to read global variables from any function block. However, to avoid race conditions with functions updating the global variables, they must only be accessed from within `synchronized` functions.

Note!

The `synchronized` keyword is ignored when used in batch agents.

To declare functions with no return value, the `void` keyword is used.

1.4 Function Blocks

APL code is divided into different function blocks that serve as execution entry-points that are applicable to the various workflow states.

1.4.1 initialize

The initialize function block is executed once for each invocation of the workflow and enables you to assign, for example, an argument with an initial value.

Note!

Avoid reading MIM parameters from within `initialize`. The order by which agents are initialized is undefined and MIM parameters are therefore not necessarily set during the initialize phase.

The udrRoute function cannot be used in the `initialize` block.

1.4.2 beginBatch and endBatch

Note!

beginBatch and endBatch are applicable for batch workflows only.

The `beginBatch` and `endBatch` function blocks are executed at the beginning and end of each batch respectively. Rules are that the beginBatch block is called when a batch collection agent emits a Begin Batch call. This occurs either at file start, or when a `hintEndBatch` call is received from any agent capable of utilizing APL code. See hintEndBatch in 29. Workflow Functions for more information.

The `endBatch` block is called every time a batch collection agent emits an End Batch. This occurs either at file end, or when a `hintEndBatch` call is received from any agent capable of utilizing APL code.

Note!

The `udrRoute` function cannot be used in the `beginBatch` and `endBatch` blocks.

1.4.3 consume

The `consume` function block is executed for each UDR or bytearray passing the agent. Within a `consume` block, validation, modification and routing can be performed. Each UDR or bytearray is referred to by the special input variable.

Built-in Variables

Variable	Description	Example
input	Read-only variable containing the current UDR. Only available in the `consume` function block.	`input.ANumber = 1234567; udrRoute (input);`

When handling several types of UDRs in the same Analysis agent, the APL code must first determine what type is currently handled, then cast it to the correct type. For an example, see 1.5 Data types.

1.4.4 drain

Note!

Drain is applicable for batch workflows only.

The drain function block is executed right before an `endBatch` block, and is treated as a final `consume` block. For instance, if a batch containing ten UDRs is processed by the agent, consume will be executed ten times before the `drain` function block is called. This is useful, for instance when collecting statistical data for a batch which is to be routed as a new UDR. The advantage with drain is that all `consume` features (except for the input built-in variable) are accessible, as opposed to the `endBatch` function block.

Example - How to use drain

```
int UDRCounter;
int file_count;
consume {
 UDRCounter=UDRCounter+1;
 udrRoute(input);
}
drain {
 myFolder.myUFDLFile.myTrailerFormat myTrailer;
 myTrailer=udrCreate(myFolder.myUFDLFile.myTrailerFormat);
 myTrailer.closingdate=dateCreateNow();
 myTrailer.numberOfUDRs=UDRCounter;
 myTrailer.sourceFileName=(string)mimGet("Disk_1",
 "Source Filename");
 udrRoute(myTrailer);
}

endBatch{
 file_count = file_count + 1;
 debug( "Number of UDRs in file:" + UDR_count );
}
```

1.4.5 cancelBatch

Note!

`cancelBatch` is applicable for batch workflows only.

The `cancelBatch` function block is executed if a Cancel Batch is emitted anywhere in the workflow. Note that End Batch and Cancel Batch are mutually exclusive - only one per batch can be executed.

If the `cancelBatch` function block is called and the Cancel Batch behavior is set to Abort Immediately the workflow will immediately abort without the `cancelBatch` function block being called. The block is only called when the preferences are set to **Abort After** or **Never Abort** . For further information about the Abort related configurations, see 3.1.8 Workflow Properties in the Desktop user's guide.

1.4.6 commit

Note!

`commit` is applicable for batch workflows only.

The `commit` function block is executed for each batch when the transaction is successful. In a `commit` block, actions that concern transaction safety can be performed. The transaction is referred to by the special `TransactionDetails` UDR that contains the transaction id.

The `udrRoute` function cannot be used in the `commit` block.

Built-in Variables

Variable	Description	Example
`transaction`	This is a read-only variable containing the current transaction. The variable is available in the `commit` and `rollback` function blocks.	`debug("commit of txn " + transaction.id);`

1.4.7 rollback

Note!

`Rollback` is applicable for batch workflows only.

The rollback function block is executed for each batch when a transaction fails. In a rollback block, actions that concern transaction safety can be performed. The transaction is referred to by the special TransactionDetails UDR that contains the transaction id.

Note!

If a transaction fails during commit, it will try to commit again, and will not be sent to the `rollback` block.

The `udrRoute` function cannot be used in the `rollback` block.

1.4.8 deinitialize

The `deinitialize` function block is executed right before the workflow stops.

If the `deinitialize` block is used in a real-time workflow it could be used to clean and close resources, for instance external connections.

1.4.9 exceptionHandling

The `exceptionHandling` function block enables you to divert exceptions from the workflow's main processing course to a separate course, where exceptions are processed according to your needs.

exceptionHandling in Batch Workflows

For example: When a workflow occasionally aborts due to an exception in the APL code, use exceptionHandling to cancel the batch.

Example - Using exceptionHandling to cancel a batch

```
consume {
 int a = 1;
 int b = 0;
 debug("The following row will generate an exception");
 float c = a/b;
}
exceptionHandling {
 debug("Type: "+ exception.type);
 debug("Message: " + exception.message);
 debug("Stacktrace: " + exception.stackTrace);
 cancelBatch("Exception caught", exception);
}
```

Note!

The `exceptionHandling` function block for batch workflows is a legacy statement that is retained for backward compatibility. It is recommended that you use try-catch, throw and finally statements instead. These are described in the next section.

Exception Handling in Batch and Real-Time Workflows

The `try-catch`, `throw` and `finally` statements can be used to handle exceptions in batch and real-time.

You use a `try-catch` block if the statements within the try block might throw an exception. `try-catch` blocks can be nested. The `try` block is followed by a `catch` block, which specifies the type of exception that it handles. You can route any UDR with udrRoute within the catch block.

If an exception is thrown, the code jumps to the `catch` block, which in this case handles all Java class Exceptions. Note that the catch in this case does not catch Throwable Exceptions.

A throw block throws an exception to the catch block which handles the exception.

A `finally` block is at the end of a catch block and runs a clean-up.

Note!

The use of these Exception Handling statements only applies to APL.

For example: When an exception is thrown due to division by zero, use the `try-catch` statement to catch the thrown exception. It is also possible to use the throw statement to throw the exception again. Use finally to ensure a clean-up is done.

Example - Using the try-catch statement

```
consume {
    int b = 0;
    try {
       debug("Hello: " + 100/b);
    }
    catch (ExceptionDetails exception) {
       debug("EXCEPTION: " + exception.message);
       // To throw exception caught, write 'throw exception'.
       throw testCreate();
       // This will cause execution of the (legacy) exceptionHandling if declared (only available in
batch workflows), otherwise
       the workflow aborts.
    }
    finally {
       debug("In finally clause " + b);
    }
}
ExceptionDetails testCreate() {
    ExceptionDetails ed = udrCreate(ExceptionDetails);
    ed.message = "DONE";
    return ed;
}
```

ExceptionDetails

The `ExceptionDetails` UDR stores the information for an exception that is caught in a catch block in batch or real-time.

Field	Description
`message (string)`	The message included in the exception
`stackTrace (string)`	The function call hierarchy from where the exception occurred
`type (string)`	Type of exception
`OriginalData (bytearray)`	This field is always empty.

1.4.10 Function Blocks Example

Example - Function Blocks Example

```
int file_count;
int UDR_count;
initialize {
 file_count = 0;
}
beginBatch {
 UDR_count = 0;
}
consume {
 udrRoute( input );
 UDR_count = UDR_count + 1;
}
endBatch {
 file_count = file_count + 1;
 debug( "Number of UDRs in file:" + UDR_count );
}
deinitialize {
 debug(" Number of executed files:" + file_count );
}
exceptionHandling {
 debug(" Exception occurred, stacktrace: "
 + exception.stackTrace);
}
```

1.5 Data types

The following table describes the different data types available in APL. The second column states if the data type is a primitive, or an object. The third column indicates the default value that an uninitialized variable will obtain.

Data type	P /O	Def val	Range	Description	Example
boolean	P	false	1 Byte	Boolean value (true|false).	`boolean aBool=true;`
byte	P	0	1 Byte, -128 to +127	Single byte integer value.	`byte aNumber=127;`
char	P	'\0'	1 Byte	A single character value.	`char aCharacter='a';`
short	P	0	2 Bytes, -32,768 to +32,767	Short integer value.	`short aNumber=31000;`
int	P	0	4 Bytes, -2,147,483,648 to +2,147,483,647	Integer values have normal Java syntax, for instance: 42, 0x2A An integer literal is always of the most narrow integer type that can contain the value. This normally works as expected, since numeric types can be implicitly converted to any wider numeric type.	`int aNumber=2147483640;`
float	P	0/0	4 Bytes, 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative)	Single-precision floating point.	`float aValue=0.5;`

double	P	0/0	8 Bytes, 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative).	Double-precision floating point.
				
				When using floating-point literals, (that is, float (f) or double (d) values) all floating-point literals without trailing d is considered float literals.
				
				Note!
				
				For instance 0.5 and 0.5d will *not* give identical values
				
				Double types may contain positive or negative infinity.
					`double aValue=432482943.1;`
					
					`double POS_INFINITY=1.0/0.0;`
					`double NEG_INFINITY=-1.0/0.0;`

long	P	0	8 Bytes, -2^63 to +(2^63 -1)	Long integer value.	`long aNumber=92233720368547;`
bigint	O	0	Unlimited	Provides storage for any integer value.	`bigint aNumber;`
bigdec	O	0	Unlimited	Provides storage for any integer value with a decimal point. Values assigned to bigdecimal variables in APL must be suffixed with b. **Example** bigdec aNumber = 123.4567b;	`bigdec aNumber;`

string	O	null	Unlimited	Values assigned to string variables in APL must be surrounded with double quotes.
				
				A character preceded by a backslash (\) is an escape sequence and has a special meaning to the compiler. The following sequences are available:
				
				• `\t` - Tab
				• `\b` - Backspace
				• `\n` - Newline
				• `\r` - Carriage return
				• `\f` - Formfeed
				• `\'` - Single quote character
				• `\"` - Double quote character
				• `\\` - Backslash character
				
				Some string comparison functions use regular expressions. If special characters such as "*", "?" are to be used as characters in the regular expression, they must be escaped with two backslashes in the APL since these strings will be parsed twice. For instance, the following function will return the index of the question mark in the string:
				`strREIndexOf("Am I right?", "\\?")`
				
				String values surrounded with triple double quotes are referred to as multiline strings. You can span the content of these string across line boundaries.
				
				Multiline strings does not support, or need, escape characters. All characters, including double quotes, are assigned as they appear.
					`string aString = "foobar";`
					`string mlString = """{`
					`"key1": "value 1",`
					`"key2": "value 2"`
					`}""";`

any	O	null		Can be assigned any value of any type.	`any aVar;`
					`aVar = 1;`
					`aVar = "foobar";`

bitset	O	null		Represents a bit string that grows as needed. The bits are indexed by non-negative integers.	`bitset aBitset;`
bytearray	O	null		Bytearray type.	`bytearray anArray;`
date	O	null		Holds date and time, including the timezone.	`date aDate;`
ipaddress	O	null		Holds IP address information. Both IPv4 and IPv6 are supported.	`ipaddress anIp = ipLocalHost();`
list <type>	O	null		List of objects of the specified type. If a list of lists is to be declared the ">" characters must be separated by at least one space, not to conflict with the ">>" operator. That is, `list<list<int> >intLists;`	`list<int>intList;`
map	O	null		Hash maps allowing key/value pairs to be defined.	`map<string, int> aMap = mapCreate(string, int);`
table	O	null		Special type used to hold table data used by the table commands.	`table aTable = tableCreate("Default", anSQL);`
UDRType	O	null		Holds a reference to a UDR. The UDRType must be a defined format or compilation will fail. There is also a special format available for generic UDRs called `drudr` and is of drudr type.	`MyDefinedType aUDR;` `drudr anyUDR;`
uuid	O	null	128 bits	Immutable universally unique identifier (UUID).	`uuid u=uuidCreateRandom();`

The variable types follow Java standard regarding object reference vs. value semantics. All types use value semantics when compared through the '==' and '!=' boolean operators and objects use reference semantics for all other operators, like assignments '='.

Example - Reference Semantics

Two date variables (objects) and two long variables (primitives) are used to illustrate the reference semantics. In the case of date variables, they will always contain the same value regardless of being updated via `myDate_A` or `myDate_B` since they point to the same object.

```
date myDate_A = dateCreateNow();
  date myDate_B = myDate_A;
  long myLong_A = 10;
  long myLong_B = myLong_A;

  if (myDate_A == myDate_B ) {
    // true
  }
  if (myLong_A == myLong_B) {
    // true
  }

  // Change value on A variables
  dateAddHours(myDate_A, 22);
  if (myDate_A == myDate_B) {
     // still true, points to same object
  }
  myLong_A = 55;
  if (myLong_A == myLong_B) {
    // false, primitives always hold their own values
  }
```

Type Casting

It is possible to cast between different types with regular Java syntax. The `instanceOf` function is used to evaluate dynamic types since an incorrect type conversion will cause the workflow to abort with a runtime error.

Example - Type casting

```
/* APL code for two UDR types */
consume {
   if( instanceOf( input, UDRType1 ) ) {
     UDRType1 udr1 = (UDRType1) input;
     // Handle UDRType1 ...
   } else {
     UDRType2 udr2 = (UDRType2) input;
     // Handle UDRType2 ...
   }
}
```

As an extension to the regular conversions, all types may be casted to string type.

Note that some types are subtypes of other types and can be directly assigned. Examples are:

- All types are subtypes of any type.
- int is a subtype (narrowing) of long.
- All UDR types are subtypes of the general base type `drudr`.
- Some UDR types are subtypes of other UDR types (depending on Ultra definitions).

There is also a number of built-in type conversion functions (refer to the section below, Access of UDR Fields).

Access of UDR Fields

UDR fields are accessed with the '.' separator. For instance, to access the field `myField` within the UDR type variable `theUDR`, the syntax would be:

```
theUDR.myField = 5;
debug( theUDR.myField );
```

If the field name is a variable instead of a string, `udrGetValue/udrSetValue` can be used to set/get field values dynamically.

Access of non-present fields will return the default value for the specific type. Field presence is evaluated with `udrIsPresent.`

When writing to a UDR field on a null object, a NullPointerException will be thrown, see the example:

Example - Access UDR fields

```
myUDR theUDR = null;
int a = theUDR.myField // Will Not throw NullPointerException
theUDR.myField = 0; // Will throw NullPointerException
```

To avoid exceptions, first make sure that the UDR is not a null object:

Example - Check for null value

```
if( theUDR != null ) {
    theUDR.myField = 0;
}
```

For further information about UDR related functions, see 2.10 UDR Functions.

Access of Input UDR

Within the `consume` block, the incoming UDR can be accessed using the special variable `input`. The `instanceOf` function may be useful in the case of dealing with several UDR types. Refer to instanceOf in 3.4 Misc Functions.

1.6 Control Flow

The following keywords are used for controlling the execution flow in the APL programming language.

Keyword	Description	Example
break	Terminates the execution of the current loop.	`while(i++ < 10) { if( i == 5 ) { break; } }`
continue	Terminates execution of the current iteration of the loop, and continues execution of the next iteration.	`while( i++ < 10) { if( i%2 == 0 //your code { else { continue; } } }`

for	`for` loop (for-each loop):
	
	The `for` loop is used to iterate through all the elements of a list in a first-to-last order. This type of loop only consists of a variable declaration and a list type. The declared variable, which is only available in the scope of the `for` loop, is assigned an element in each iteration.
	
	`for` loop (advanced):
	
	Creates a loop that consists of three expressions.
	
	The first expression performs initialization of a variable that is used for counting iterations. A variable declared in this expression is only available in the scope of the for loop.
	
	The second expression contains a condition that must evaluate to true or false. The loop iterates until the condition evaluates to false.
	
	The third expression is evaluated at the end of each loop iteration, before the next evaluation of the condition. This is typically used to update or increment the counting variable in the condition.
		`list<int> listOfInts = listCreate(int, 1,2,3); for (int i: listOfInts) { //your code }`
		
		`list<string> listOfStrings = listCreate(string," 1","2","3"); for (string s: listOfStrings) { //your code }`
		
		`for( int i=0; i<10; i++ ) { //your code }`

if else	Used to execute code with conditions.	`if (myVar == "ABC") {// do_this} else {// do_that}`

return	Stops the execution of the current function block.	`if ( ANumber == null ) { return;}`
while	Loops while condition is true. **Note!** If the APL code generates an infinity loop it can be stopped by, in the Workflow Monitor or Execution Manager, selecting the **Stop** button and **Stop Immediate**. This will however *not* work if the loop is running in the `initialize` or `deinitialize` blocks.	`while( i < 20 ) { // do_stuff}`

switch case	Allows a variable to be tested for equality against a list of values or types, where each of these is a case.
	
	The cases are evaluated against the variable in order, from top to bottom. After a match, the remaining cases will not be evaluated.
	
	Testing values:
	
	The value for a `case` must be the same data type as the variable in the switch parameter and it must be a constant or a literal.
	
	Testing types:
	
	The type for a `case` must be followed by a variable declaration as shown in the example column.
	
	A switch statement can have an optional default case, `else`, which must appear at the end of the `switch`. The default case can be used for performing a task when the variable does not match any of the other cases.
		`//Testing values switch (str) { case "one" { debug("One"); } case "two" { debug("Two"); } else debug ("other"); }`
		
		`//Testing types switch (input) { case (PulseUDR p){`
		
		`debug(p.Sequence); } case (MyUDR m) {`
		
		`debug(m.aField); } }`

1.7 Constants

The following predefined constants are available in the APL language.

Constant	Description	Example
null	Can be used to evaluate if a field has been assigned a value.	`if ( NumFld != null )`
`true` / `false`	Value of a boolean variable	`myBool = true;`
WORKFLOW_NAME	The workflow configuration name. **Note!** If used in a workflow with the name **Default.MyWorkflow.workflow_1**, this constant will hold the value Default.MyWorkflow. If the full name is needed, i.e. **Default.MyWorkflow.workflow_1**, use mimGet ("Workflow", "Workflow Name") instead.	
AGENT_NAME	Agent name	

1.8 Separators

The following separator characters are valid in the APL language.

Separator	Description	Example
`()`	Used to enclose logical expressions and to overrule standard precedence	`NumFld = (NumFld2 + 2) * 3;`
`{}`	Used to enclose blocks	`if ( ACode == "ABC" ) { // do_this // and_this }`
`.`	Separates fields in UDR value accesses	`input.myField = 5;`

1.9 Operators

The following operators are valid in the APL language.

Operator	Description	Example
Arithmetic operators:		
+	Addition (numeric types) or string concatenation (if left operand is of string type).	`string1 = string2 + string3;` `NumFld = NumFld + 1;`
Arithmetic operators (only numeric types):		
-	Subtraction	`NumFld = NumFld - 2;`
*	Multiplication	`NumFld = NumFld * 3;`
/	Division	`NumFld = NumFld / 4;`
%	Modulus	`NumFld = NumFld % 10;`
Unary operators (only primitive integer types, cannot be a UDR field):		

++	Increment	`NumFld++; //Increment after evaluation`
		`++NumFld; //Increment before evaluation`

--	Decrement	`NumFld--; //Decrement after evaluation`
		`--NumFld; //Decrement before evaluation;`

Bit operators (only integer types):		
&	Bitwise AND	`NumFld = NumFld & 1;`
|	Bitwise OR	`NumFld = NumFld | 2;`
<<	Shift bits left	`NumFld = NumFld << 1;`
>>	Shift bits right	`NumFld = NumFld >> 1;`
Boolean operators:		
==	Equal to	`if (Fld1 == 1)`
!=	Not Equal to	`if (Fld1 != 4)`
&&	Logical AND	`if (Fld1 == 1 && Fld2 != 4)`
||	Logical OR	`if (Fld1 == 1 || Fld2 != 4)`
<=	Less than or equal to	`if (Fld1 <= 5)`
<	Less than	`if (Fld1 < 5)`
>=	Greater than or equal to	`if (Fld1 >= 5)`
>	Greater than	`if (Fld1 > 5)`
!	Not	`if (! BoolFld)`

Type conversions for the arithmetic operators follow the Java standard.

The && and || operators have the same precedence. Both operators have right fixity. For clarity and to avoid errors, it is generally recommended to override the precedence rules by using parentheses in expressions.

Example - Operator precedence

```
consume {
    boolean a = false;
    boolean b = false;
    boolean c = true;
    boolean x;
    // The following statement will be parsed as
        // false && (false || true)
    // and evaluate to false
    x = a && b || c;
    debug(x);
    // This will evaluate to true
    x = (a && b) || c;
    debug(x);
}
```

Java and APL may differ

In some circumstances, APL and Java are handling numeric values differently. APL may be automatically expanded to avoid overflow, this applies also to constant expressions.

Example 1: A statement like: long x=1000000000000; is ok in APL, but will not compile in Java.

Example 2: An expression like: long x=1000000*1000000; works as expected in APL. But in Java, it would assign x to the value -727379968, due to overflow.

1.10 Global Code

Global Code can be either APL code, as entered in the APL Code Editor; or UFDL code, as entered in the Ultra Format Language Editor. Global APL Code is called packages and is referred to through the name under which it was saved. Correspondingly, Global UFDL Code is called modules.

Using Global APL Code

Before Global APL Code can be used in the Analysis and Aggregation agents, it must be imported:

```
import apl.<foldername>.<APL Code configurationname>
```

The package is the name under which the Global APL code was saved. Note that single function blocks from within a package may not be imported - the whole package must be imported.

Note that function names are resolved first in the imported scope, and second in the local scope.

The import statement may also be used to include UFDL modules, that is, UDR types (see the section below, Using Global UFDL Code).

Using Global UFDL Code

Lookup of type names is slightly different in APL than in Ultra definitions. In APL, the module must either be explicitly specified or imported.

The explicit specified module import is done by:

```
import ultra.<folder>.<module>;
```

The module is the name under which the Ultra format was saved.

The modules of any entries in the UDR types section of the Analysis agent are implicitly imported. For instance, if the input type `myUDR (myFolder.module1)` is configured, the complete module `myFolder.module1` is imported.

Example - Using global UFDL code

```
/* APL code without import statement */
consume {
 myFolder.module1.MyUDR newUDR =
 udrCreate( myFolder.module1.MyUDR );
 newUDR.number = input.seqNum;
 newUDR.sub = udrCreate( myFolder.module1.MySubUDR );
 newUDR.sub.subNum = 17;
udrRoute( newUDR );
}
/* APL code with import statement */
import ultra.myFolder.module1;
consume {
 MyUDR newUDR = udrCreate( MyUDR );
 newUDR.number = input.seqNum;
 newUDR.sub = udrCreate( MySubUDR );
 newUDR.sub.subNum = 17;
udrRoute( newUDR );
}
```

2. General Functions

This chapter describes functions that are commonly used for working with object data types that are available in APL.

2.1 Bit Operation Functions

The functions described in this section are used to perform operations on variables of the bitset type.

bitIsSet and bitIsCleared

The `bitIsSet` and `bitIsCleared` functions evaluate whether an individual bit is set or cleared. This can be done against either an integer, or a bitset value. For integer type, these are convenience functions that can be used instead of the regular logical operators.

```
boolean bitIsCleared
( int|bitset value,
int bitnumber )
boolean bitIsSet
( int|bitset value,
int bitnumber )
```

Parameter	Description
value	Bitset or integer to evaluate
bitnumber	The bit to evaluate. 0 (zero) is the least significant bit, 31 is the most significant.
Returns	false or true

bsCreate

The `bsCreate` function creates an empty bitset.

```
bitset bsCreate()
```

Parameter	Description
Returns	An (empty) bitset

bsSet and bsClear

The bsSet and bsClear functions set or clear a bit in a bitset.

```
void bsClear
( bitset bitset, int bitnumber )
void bsSet ( bitset bitset,
int bitnumber )
```

Parameter	Description
bitset	The bitset to modify
bitnumber	The bit to change. 0(zero) is the least significant bit, 31 is the most significant.
Returns	Nothing

2.2 Bytearray Functions

The functions described in this section are used to perform operations on variables of the `bytearray` type.

baAppend

Concatenates two bytearrays and returns the result. None of the input arrays are changed.

```
bytearray baAppend
(bytearray array1,
bytearray array2 )
```

Parameter	Description
array1	A bytearray
array2	Another bytearray to append to the end of *array1*
Returns	The concatenated bytearray

baCreate

Creates a new bytearray of the specified size.

```
bytearray baCreate( int size )
```

Parameter	Description
size	Size of the new array in bytes
Returns	A bytearray of the specified size

baCreateFromHexString

Converts a readable hex string into a bytearray.

```
bytearray baCreateFromHexString( string hexString )
```

Parameter	Description
hexString	The hex string you want to convert into a bytearray.
Returns	A new bytearray containing the data from the hex string.

Example - Using hex string

If the hex string is "100A" the returned byte array will contain 2 bytes; one with decimal value 16 (which equals hex value 10), and one with decimal value 10 (which equals hex value 0A):

```
00010000 and 00001010
```

If the hex string contains an odd number of digits/characters, a "0" will be added at the end of the string, and it will then be handled in the same way as in the example above. For example, if the hex string is "123", a "0" will be added, giving a string containing "1230". The returned bytearray will then contain 2 bytes; one with decimal value 18 (which equals hex value 12) and one with decimal value 48 (which equals hex value 30):

```
00010010 and 00110000
```

baGet

Retrieves a byte value from a bytearray. The index must be in the range `0<=index<baSize( array )` or the workflow will abort with a runtime error.

```
int baGet (bytearray array, int index)
```

Parameter	Description
array	A bytearray
index	Index of the element to retrieve
Returns	A decimal representation of the indexed byte

Example - Using baGet

Consider a bytearray with the following hex dump "00000000: 484a 4c HJL". Accessing the first element according to the example code will return "72" - the decimal value of 0x48.

```
int myVar = baGet( myBA, 0 );
```

baHexDump

Converts a bytearray into a readable hex dump string, useful for debugging.

```
string baHexDump( bytearray array )
```

Parameter	Description
array	A bytearray
Returns	A string containing the hex formatted bytearray

Example - Using baHexDump

Consider a bytearray created with the code in this example.

```
bytearray myBA = baCreate( 3 );
baSet( myBA, 0, 72 );
baSet( myBA, 1, 74 );
baSet( myBA, 2, 76 );
string myDump = baHexDump(myBA);
```

Using the bytearray as input to baHexDump will output a hex dump with the following appearance:

```
"00000000: 484a 4c HJL"
```

baInsert

Inserts one bytearray into another bytearray and returns a new bytearray containing the two merged bytearrays.

```
bytearray baInsert
( bytearray ba1,
int pos,
bytearray ba2 )
```

Parameter	Description
ba1	The bytearray into which you want to insert bytearray `ba2`
pos	The position in bytearray `ba1` where you want to insert bytearray `ba2`
ba2	The bytearray you want to insert into bytearray `ba1`
Returns	A new bytearray containing the data in bytearrays `ba1` and `ba2` arranged in the order determined by the `pos` parameter

baReplace

Replaces the data in one bytearray with the data in another bytearray and returns a new bytearray with the replaced data.

```
bytearray baReplace
( bytearray ba1,
int pos,
bytearray ba2 )
```

Parameter	Description
ba1	The bytearray in which you want bytes to be replaced
pos	The position in `ba1` where you want the replacement to start
ba2	The bytearray containing the bytes you want replace the bytes in `ba1` with
Returns	A new bytearray containing the replaced bytes arranged in the order determined by the `pos` parameter

baSet

Sets the value of a byte in the bytearray. The index must be in the range `0<=index<baSize( array )` or the workflow will abort with a runtime error.

```
void baSet
( bytearray array,
int index,
int value )
```

Parameter	Description
array	A bytearray.
index	Index of the byte to set
value	The new value of the byte. The actual value set is (*value & 0xFF*)
Returns	Nothing

baSize

Returns the size of an array.

```
int baSize( bytearray array )
```

Parameter	Description
array	A bytearray
Returns	The size of the array in bytes

2.3 Date Functions

All date related functions conforms to the standard Java date functions. Years, entered with two digits, will refer to 1900 if they are between 70 and 99 and refer to year 2000 if they are between 00 and 69.

> **Warning!**
>
> The Date functions cannot operate on null arguments.

dateAdd*

The following functions adds a specified number of years, months, days, hours, minutes, seconds, or milliseconds to a date value.

```
void dateAddYears ( date d, int years )
void dateAddMonths ( date d, int months )
void dateAddDays ( date d, int days )
void dateAddHours ( date d , int hours)
void dateAddMinutes ( date d, int minutes )
void dateAddSeconds ( date d , int seconds)
void dateAddMilliseconds ( date d, int milliseconds )
```

Parameter	Description
d/years/months/days	A date/year/month/day...
Returns	Nothing

dateCreate*

The following functions creates a full date, based on current system timestamp of the host, or a given date and/or time:

```
date dateCreateNow()
date dateCreate ( int year, int month, int day, int hour, int minute, int second )
```

Creates dates where either the date or time part is set:

```
date dateCreateFromDate ( int year, int month, int day )
date dateCreateFromTime ( int hour, int minute, int second )
date dateCreateFromMilliseconds ( long milliseconds )
```

Creates a copy of a date:

```
date dateCreateCopy( date d )
```

Parameter	Description
d/year/month/day	A date/year/month/day...
Returns	A date

dateCreateNowMilliseconds

The dateCreateNowMilliseconds function returns the current date of the operating system in milliseconds of the operating system.

```
long dateCreateNowMilliseconds()
```

Parameter	Description
Returns	The current date in milliseconds

dateDiff

The dateDiff function calculates the difference in milliseconds between two dates. The return value is date1 - date2 which means that the result may be negative.

```
long dateDiff ( date date1, date date2 )
```

Parameter	Description
date1	The date to subtract from
date2	The date to subtract with
Returns	The difference in milliseconds

dateGet*

Given a date, the following functions return the year number, month number, day number (Monday=1, Tuesday=2, etc), hour (24 hour clock), minute, second, or millisecond.

```
int dateGetYear( date d )
int dateGetMonth( date d )
int dateGetDay( date d )
int dateGetDayOfWeek( date d )
int dateGetHours24( date d )
int dateGetMinutes( date d )
int dateGetSeconds( date d )
int dateGetMilliseconds( date d )
string dateGetTZ( date d )
```

> **Note!**
>
> The function `dateGetTimeZone()` is deprecated, please use `dateGetTZ()` instead.
>
> For further information about time zone settings, see 8. Database Configuration in the System Administrator's Guide.

Parameters:

Parameter	Description
d	The date to convert
Returns	Integers stating year/month/day/hour/second for all functions except `dateGetTZ`, which returns the time zone in the way stated by the JVM's TimeZoneID.

Example - When debugging the current time zone with dateGetTZ

The syntax used when debugging the current time zone with `dateGetTZ`:

```
debug( dateGetTZ( dateCreateNow() ) );
```

dateGetAsMilliseconds

Given a date, the function `dateGetAsMilliseconds` returns the total number of milliseconds since 1 Jan, 1970. This function is the inverse of `dateCreateFromMilliseconds`.

```
long dateGetAsMilliseconds( date d )
```

Parameter	Description
d	The date to examine.
Returns	The total number of milliseconds since 1 Jan, 1970.

dateGMTOffset

The dateGMTOffset function returns the number of milliseconds diverging from GMT.

```
long dateGMTOffset( date d )
```

Parameter	Description
d	The date to examine
Returns	Number of milliseconds diverging from GMT

dateHas*

Given a date the following functions return true if the date includes a date or time part.

```
boolean dateHasDate( date d ) boolean dateHasTime( date d )
```

Parameter	Description
d	The date to examine
Returns	true or false

dateInDaylightTime

Given a date, the dateInDaylightTime functions returns true if the date is a daylight-saving date.

```
boolean dateInDaylightTime( date d )
```

Parameter	Description
d	The date to examine
Returns	`true` or `false`

dateIsLeapYear

Given a date, the dateIsLeapYear function performs a leap year evaluation.

```
boolean dateIsLeapYear( date d )
```

Parameter	Description
d	The date to examine
Returns	`true` or `false`

dateNanoseconds

The dateNanoseconds function returns the operating system uptime in nanoseconds. This is mainly used to measure code execution times.

> **Note!**
>
> The returned value will have nanosecond precision, however not necessarily nanosecond accuracy, since it is dependent on how frequently values are updated in the operating system.

```
long dateNanoseconds()
```

Parameter	Description
Returns	The current value of the most precise available system timer in nanoseconds.

dateSet*

The following functions sets different parts of a given date:

```
void dateSetYear ( date d, int year )
void dateSetMonth ( date d, int month )
void dateSetDay ( date d, int day )
void dateSetHours24 ( date d , int hours)
void dateSetMinutes ( date d, int minutes )
void dateSetSeconds ( date d , int seconds)
void dateSetMilliseconds ( date d, int milliseconds )
void dateSetTZ ( date d, string timeZone )
```

The following functions sets the complete date or time part of a date:

```
void dateSetDate ( date d, int year, int month, int day )
void dateSetTime ( date d, int hour, int minute, int second)
```

Note!

The function `dateSetTimeZone()` is deprecated. Use `dateSetTZ()` instead.

For `dateSetTimeZone()`, timezone did not take effect until date was retrieved by one of the dateGet*-functions. With `dateSetTZ()` timezone takes effect immediately.

For further information about time zone settings, see 8. Database Configuration in the System Administrator's Guide.

Parameter	Description
d/year /month /day	A date/year/month/day... **Note!** The component to be set needs to represent a valid value in order not to throw an exception at a later stage. For instance; Month= 1-12, Hours=0-23, Minutes=0-59, Milliseconds=0-999. Also, the date needs to represent a valid date. For instance February 31 is invalid.
timeZone	An optional string stating the timezone to set
Returns	Nothing

2.4 IP Address Functions

All functions support both IPv4 and IPv6. The results from some functions may however depend on the underlying system configuration.

getHostname

Returns the fully qualified domain name of the host on which the workflow is active. This is the best effort method, meaning we may not be able to return the FQDN depending on the underlying system configuration.

```
string getHostname()
```

Parameter	Description
Parameters:	None.
Returns	The hostname as a string.

getIPAddress

Returns the IP address of the local host.

```
string getIPAddress()
```

Parameter	Description
Parameters:	None
Returns	The raw IP address as a string

ipAddressString

Extracts the IP address part from a variable of type ipaddress.

```
string ipAddressString(ipaddress ipa )
```

Parameter	Description
ipa	An IP address
Returns	The IP address as a string, e g "10.0.3.22"

ipFromAddress

Converts the bytearray representation into an ipaddress type. Refer to 1.5 Data types for further information about ipaddress type.

```
ipaddress ipFromAddress ( bytearray address )
```

Parameter	Description
address	The IP address as a bytearray. IPv4 address bytearray must be 4 bytes long and IPv6 bytearray must be 16 bytes long.
Returns	The IP address as an ipaddress type.

ipFromHostname

Given a string containing the IP address (numerical presentation), or hostname, a variable of type ipaddress is returned.

> **Note!**
>
> This function can be time consuming if DNS has to be accessed.

```
ipaddress ipFromHostname ( string host )
```

Parameter	Description
host	The name of the host
Returns	The IP address provided by the DNS server

ipIsV6Address

Evaluate if the IP address represents an IPv6 address.

```
boolean ipIsV6Address()(ipaddress ipa )
```

Parameters:

Parameter	Description
ipa	The IP address to evaluate
Returns	`true` if the address is IPv6, `false` otherwise.

ipLocalHost

Returns the IP address of the local host.

```
ipaddress ipLocalHost()
```

Parameter	Description
Returns	The IP address of the local host

ipToAddress

Returns the numerical representation of an IP address for example (10.0.3.22) as elements of a bytearray.

```
bytearray ipToAddress ( ipaddress ipa )
```

Parameter	Description
ipa	An IP address
Returns	A bytearray representation of the IP address; for example [10 0 3 22].

ipToHostname

Extracts the fully qualified domain name part from a variable of type ipaddress. The host running the EC must have DNS setup. Note that this method may be time consuming.

```
string ipToHostname( ipaddress ipa )
```

Parameter	Description
ipa	An IP address.
Returns	The hostname corresponding to the given IP address.

2.5 List Functions

This section describes functions that are used to manage lists and their elements. Lists can be created to hold any type of data.

listAdd

Adds a new element to the end of an existing list.

```
void listAdd
 ( list<type> list,
 any value )
```

Parameter	Description
list	The name of the list
value	Value to add to the list. The value must be of the same type as the list is designed to accept.
Returns	Nothing

Example - Using listAdd

The following code leaves a list consisting of the two elements 7 and 5.

```
list <int> myIntList = listCreate(int, 7);
 listAdd(myIntList, 5);
```

listClear

Removes all elements from a list.

```
void listClear
( list<type> list )
```

Parameter	Description
list	The list of elements to be removed
Returns	Nothing

listCreate

Creates a list of any type, and (optionally) populates it.

```
list<type> listCreate
 ( type,
 element1, //Optional
 element2, //Optional
 ... )
```

Parameter	Description
type	The type of the elements comprising the list
element<n>	The value of an element to be added to the list. Optional parameters: if none is entered, the list will be empty.
Returns	The list

listCreateSync

Creates a synchronized list of any type, and (optionally) populates it. This function may be useful for global lists in real time workflows, where the same list may be accessed from several threads at the same time.

```
list<type> listCreateSync
( type,
element1, //Optional
element2, //Optional
... )
```

Parameter	Description
type	The type of the elements comprising the list
element<n>	The value of an element to be added to the list. Optional parameters: if none is entered, the list will be empty.
Returns	The list

listFindIndex

Calculates the index of a requested list element.
The condition is tried for each element in the list until there is a match or the end of the list is reached.

```
int listFindIndex
 ( list<type> list,
 varSpec,
 Condition,
 int startIndex ) //Optional
```

Parameter	Description
list	The list to evaluate
varSpec	Variable specification. Defines a variable to hold the list data for the condition evaluation. Can be either a valid variable name in which case the variable type is the same as the list element type, or a type can be explicitly set on the variable in which case the list elements are cast to this type and then assigned to the variable. An incorrect type will cause a runtime `ClassCastException`.
Condition	Condition for matching an item in the list.
startIndex	(Optional) index to start to search the list on. If `startIndex` is not specified, the list is searched from the beginning.
Returns	The index of the first matching element. If no element matched the condition, -1 is returned. The first element in the list is indexed 0 (zero).

Example - Using listFindIndex

A list of integers containing the following elements [50, 2, 4, 152, 4] is searched for an element equal to 4:

```
debug( listFindIndex( myList, i, i == 4 ) );
```

The output is 2, since it is the index number of the first element equal to 4 in the list.

listGet

Retrieves an element from a given position in the list.

```
any listGet
 ( list<type> list,
 int index)
```

The index must be in the range 0<=index<listSize(list) or the workflow will abort with a runtime error.

Parameter	Description
list	A list
index	Index of the element to retrieve
Returns	The value given at the defined index

listInsert

Inserts a value on a given position in a list. Note that this will NOT overwrite any existing values on the position, but only move the old element up a position.

```
void listInsert
 ( list<type> list,
 int index,
 type value )
```

Parameter	Description
list	A list
index	Where to insert the new value
value	The new value
Returns	Nothing

Example - Using listInsert

A list consists of the following elements: [2, 5, 72, 19] and the new value 127 is to be inserted on index position 2. This means the new indexes for the elements containing 72 and 19 are now 3 and 4 respectively:

```
listInsert( mylist, 2, 127);
```

`myList` now looks like this: [2, 5, 127, 72, 19]

listRemove

Removes an element from a list.

```
void listRemove
 ( list<type> list,
 int index )
```

Parameter	Description
list	A list
index	Index of the value to remove. The index must be in the range `0<=index<listSize ( list )` or the workflow will abort with a runtime error.
Returns	Nothing

listSet

Replaces an existing element in a list.

```
void listSet
 ( list<type> list,
 int index,
 type value )
```

Parameter	Description
list	A list
index	The position of the element to be replaced
value	The new value to replace the existing with
Returns	Nothing

Example - Using listSet

A list consists of the following elements: [2, 5, 72, 19] and the new value 127 should replace the value of the element on index position 2:

```
listSet( mylist, 2, 127);
```

myList now looks like this: [2, 5, 127, 19]

listSize

Gets the size of a list.

```
int listSize ( list<type> list)
```

Parameter	Description
list	A list
Returns	The number of elements in the list

listSort

Sorts a list of any type. For UDR lists, sorting is based on a field in the UDR.

For all lists except for UDR lists:

```
void listSort
 ( list<type> list,
 order ) //Optional
```

For UDR lists:

```
void listSort
 ( list<UDRtype> UDRlist,
 string field,
 order, //Optional
 udrtype ) //Optional
```

Parameter	Description
list	A list
UDRlist	A list of UDRs
field	Name of the UDR field to sort on. This field may contain any primitive type except for *any* , *bitset* and *ipaddress* .
order	Sorting order (optional). Could be one of ascending (default) or descending.
udrtype	Parameter to determine UDR type if the list is declared to hold generic UDRs. In the following bullets, the first is generic and must have the *udrtype* specified. The second does not require the *udrtype* since the UDR type is known from the declaration. 1. list<drudr> MyList; 2. list<MyUDRType> MyList; If a type name is used and the UDR cannot be converted to this type this will be a runtime error.
Returns	Nothing

Example - Using listSort

```
list<MyUDRType> myList;
// Assume we have added items to the list
listSort( myList, SequenceNumber, descending );
```

2.6 Map Functions

This section describes functions that enable use of hash maps.

mapClear

Removes all entries in a map.

```
void mapClear
 (map<any,any>  myLittleMap );
```

Parameter	Description
myLittleMap	The map to clear
Returns	Nothing

mapContains

Evaluates if a key is present in a map.

```
boolean mapContains
 ( map<any,any>  myMap ,
 any  myKey  );
```

Parameter	Description
myMap	Name of the map to evaluate
myKey	The requested key
Returns	`true` or `false`

mapCreate

Creates a new, empty map.

```
map<any,any> mapCreate
 ( any  myKeyType ,
 any  myValueType );
```

Parameter	Description
myKeyType	Defines the data type of the keys in the map
myValueType	Defines the data type of the values, associated with the keys in the map
Returns	An empty map

Example - Using mapCreate

An example of a map definition, including assignment of the first key/value pair.

```
map<string,int> myMap = mapCreate( string, int); mapSet( myMap, "Failed", 22 );
```

mapCreateSync

Creates a new, empty synchronized map. This function may be useful for global maps in real time workflows, where the same map may be accessed from several threads at the same time.

```
map<any,any> mapCreateSync
 ( any  myKeyType ,
 any  myValueType );
```

Parameter	Description
myKeyType	Defines the data type of the keys in the map
myValueType	Defines the data type of the values, associated with the keys in the map
Returns	An empty map

Example - Using mapSet

An example of a map definition, including assignment of the first key/value pair.

```
map<int,string> mySyncMap = mapCreateSync( int, string);
mapSet( mySyncMap, 55, "Successful" );
```

mapGet

Retrieves the value of a specified key in a map.

```
valueType mapGet
 ( map<key,value>  myMap ,
 keyType  key  );
```

Parameter	Description
myMap	The name of the map from which you want to retrieve a value
key	The name of the key holding the value to you want to retrieve
Returns	Returns the value associated with the key. The data type of the returned value is the same as the defined value type for the map. If there is no matching key `mapGet` will return null, false or 0 depending on type.

Note - using mapGet

If the map is `map<int,string>`. The return type of "mapGet" will be "string".

mapKeys

Returns a list of all keys present in a map. Note that the order of the elements in the list is not defined.

```
list<any>
 mapKeys(map<any,any> myMap );
```

Parameter	Description
myMap	The map to fetch all keys from
Returns	An unsorted list of keys according to: [keyA, keyB, keyX...] The data type of the list elements is the same data type as defined for the keys in the map.

mapRemove

Removes a key and its corresponding value.

```
any mapRemove
 ( map<any, any>  myMap ,
 any  myKey );
```

Parameter	Description
myMap	The map to remove a key from
myKey	The name of the key to remove
Returns	The value corresponding to the removed key

mapSet

Sets or updates a key's associated value.

```
void mapSet
 ( map<any, any>  myMap ,
 any  myKey ,
 any  myKeyValue );
```

Parameter	Description
myMap	The map to update
myKey	The name of the key to set/update
myKeyValue	The corresponding value to set/update
Returns	Nothing

mapSize

Returns the number of keys present in a map.

```
int mapSize( map<any,any> myMap );
```

Parameter	Description
myMap	The map to examine
Returns	The size of the map in terms of number of keys

mapValues

Returns a list of all values present in a map. Note that the order of the elements in the list is not defined.

```
list<any>mapValues(map<any, any> myMap );
```

Parameter	Description
myMap	The map to fetch all values from
Returns	A list of all values (unsorted): [valueA, valueB, valueX...]. The data type of the list elements is the same data type as defined for the values in the map.

2.7 String Functions

> **Warning!**
>
> The String functions cannot operate on null arguments.

String Concatenation

Strings are concatenated with the arithmetic '+' operator:

```
string str1 = string str2 + string str3 ...
```

strEndsWith

Returns `true` if *`str`* ends with *`substr`* , otherwise `false`.

```
boolean strEndsWith
 ( string  str ,
 string  substr )
```

Parameter	Description
str	String to examine
substr	String to look for
Returns	`true` or `false`

strEqualsIgnoreCase

Compares two strings and returns `true` if they are equal, and `false` if they are not. This comparison ignores case.

```
boolean strEqualsIgnoreCase
 ( string  str1 ,
 string  str2  )
```

Parameter	Description
str1	A string to compare
str2	Another string to compare
Returns	`true` or `false`

strIndexOf

Returns the first position where *substr* can be found in *str* . If *substr* is not found, `-1` is returned. The position starts to count from 0 (zero), which is the first character.

```
int strIndexOf
( string str ,
string substr,
int startIndex ) //Optional
```

Parameter	Description
str	String to examine
substr	String to look for
startIndex	Index where to start the string search. If *startIndex* is not specified, the string is searched from the beginning.
Returns	The position of the first character of *substr* within *str* . If not found, `-1` is returned.

strLastIndexOf

Returns the last position where *substr* can be found in *str* . If *substr* is not found, `-1` is returned. The position starts to count from 0 (zero), which is the first character.

```
int strLastIndexOf
 ( string  str ,
 string  substr, int startIndex  ) //Optional
```

Parameter	Description
str	String to examine
substr	String to look for
startIndex	Index where to start the string search. If *startIndex* is not specified, the string is searched from the beginning.
Returns	The position of the last character of *substr* within *str* . If not found, -1 is returned.

strInsert

Inserts a string into another at a specific position, and returns the result string.

```
string strInsert
 ( string  str1 ,
 int  position ,
 string  str2  )
```

Parameter	Description
str1	The string that *str2* is inserted into
str2	The string to insert
position	Position where *str2* will be inserted. Position 0 (zero) indicates prior to the first character of *str1* .
Returns	The result string. Note, the *str1* and *str2* are not modified.

strLength

Returns the number of characters in a string.

```
int strLength( string  str  )
```

Parameter	Description
str	String to examine
Returns	The number of characters in the string

strREContains

Returns `true` if a string contains a substring, else `false`. The function is case sensitive.

```
boolean strREContains
 ( string  str ,
 string  regexp  )
```

Parameter	Description
str	String to examine
regexp	The substring to look for. Regular expressions are allowed.
Returns	`true` or `false`

Note!

Regular expressions according to Java syntax applies. For further information, see http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

Warning!

Using nonconstant regular expressions may consume large amounts of memory and should therefore be avoided.

strREIndexOf

Returns the first position where a regular expression can be found in a string. If the regular expression is not found, `-1` is returned. The position starts to count from 0 (zero), which is the first character.

```
int strREIndexOf
 ( string  str ,
 string  regexp  )
```

Parameter	Description
str	String to examine
regexp	Regular expression to be used when examining *str* , using `case sensitive` and `contains` (not `matches`).
Returns	The position of the first character of *regexp* within *str* . If not found, `-1` is returned.

Example - Using strREIndexOf

```
strREIndexOf( "Hello There!", "[Tt]he" ) // Returns 6
```

Note!

Regular expressions according to Java syntax applies. For further information, see http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

Warning!

Using nonconstant regular expressions may consume large amounts of memory and should therefore be avoided.

strREMatches

Returns `true` if the first stated string matches the content of the second string completely. The function is case sensitive.

```
boolean strREMatches
 ( string  str ,
 string  regexp  )
```

Parameter	Description
str	The first string
regexp	The second string. Regular expressions are allowed
Returns	`true` or `false`

Example - Using strREMatches

```
strREMatches( "abc", "ab." )
```

Will return `true`.

```
strREMatches( "abc", "a..c" )
```

Will return `false`.

```
strREMatches( "abc", "a[a-z]c" )
```

Will return `true`.

```
strREMatches( "abc", "a[A-Z]c" )
```

Will return `false`.

```
strREMatches( numberString, "[0-9]*" )
```

Will check that the variable `numberString` only contain digits. It will return `true` for "123" and `false` for "123F".

Note!

Regular expressions according to Java syntax applies. For further information, please refer to http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

Warning!

Using nonconstant regular expressions may consume large amounts of memory and should therefore be avoided.

strREReplaceAll

Replaces existing substrings within a string with new values.

```
string strREReplaceAll
 ( string  str ,
 string  old ,
 string  new  )
```

Parameter	Description
str	String to change
old	The substring to replace. Regular expressions are allowed.
new	The new substring to replace the old. If an empty string is entered, the old will be removed without inserting any new value.
Returns	The result string

Example - Using strREReplaceAll

```
string strREReplaceAll
 ( "flower", "low", "orm" )
```

This call will change the string "flower" to the string "low". If the *new* parameter was left empty ("orm" in this case), the resulting string would have been "fer".

Note!

Regular expressions according to Java syntax applies. For further information, please refer to http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

Warning!

Using nonconstant regular expressions may consume large amounts of memory and should therefore be avoided.

strReplaceChars

Replaces characters in a string at a given position with another string and returns the new string. If *str2* is longer than *str1* , then the resulting string will be expanded to fit the complete *str2* .

```
string strReplaceChars
 ( string  str1 ,
 int  position ,
 string  str2  )
```

Parameter	Description
str1	The base string
position	Position where *str2* will start replacement in *str1* . Position 0 (zero) indicates prior to the first character of *str1* .
str2	String to use for replacement
Returns	The result string. Note, the *str1* and *str2* are not modified.

Example - Using strReplaceChars

The following example returns the string: `Hi Walter`

```
strReplaceChars("Hi Sister", 3, "Walt");
```

strSplit

Splits a string where a given regular expression is matched into a list of strings. The function will try to match the regular expression and split the string as many times as possible.

```
list<string> strSplit
 ( string  str ,
 string  regex  )
```

Parameter	Description
str	The base string
regex	The regular expression which is to be used to split elements
Returns	A list of strings

This function would typically be used for splitting strings of values that are comma separated, colon separated, or similar. The list returned will present the substrings in the order they occur in the original string.

Example - Using strSplit

```
strSplit("one,two,three", ",");
```

Will return the following list:

```
one
two
three
```

```
strSplit("name:date:time", ":");
```

Will return the following list:

```
name
date
time
```

```
strSplit("person a person b person c", "person ");
```

Will return the following list:

```
a
b
c
```

strStartsWith

Returns `true` if *`str`* starts with *`substr`* , otherwise `false`.

```
boolean strStartsWith
 ( string  str ,
 string  substr  )
```

Parameter	Description
`str`	String to examine
`substr`	String to look for
Returns	`true` or `false`

strSubstring

Returns a substring from a given string. The extracted substring is from position *`start`* to *`end`* .

```
string strSubstring
 ( string  str ,
 int  start ,
 int  end  )
```

Parameter	Description
str	The base string
start	Position where to start the substring extraction. Position 0 (zero) points to the first character of *str* .
end	Position where to end the extraction. That is, the index of the letter after the last letter in the substring.
Returns	The substring

Example - Using strSubstring

The following call returns the string "the".

```
strSubstring("hi there", 3, 6);
```

strToLower

Turns all letters in a string to lower-case.

```
string strToLower( string  str  )
```

Parameter	Description
str	The base string
Returns	The result string. Note, the *str* is not modified.

strToUpper

Turns all letters in a string to capital letters.

```
string strToUpper( string  str  )
```

Parameter	Description
str	The base string
Returns	The result string. Note, the *str* is not modified.

strTrim

Removes leading and trailing spaces from a string and returns the result.

```
string strTrim( string  str  )
```

Parameter	Description
str	The base string
Returns	The result string. Note, the *str* is not modified.

2.8 Type Conversion Functions

This section describes functions that are used to perform conversions between different types.

baToBigInt

Converts a bytearray to a big integer (two complement).

```
void baToBigInt
 (bigint bigintVar,
 bytearray baValue)
```

Parameter	Description
bigintVar	Bigint variable to set
baValue	Bytearray value
Returns	Nothing

baToHexString

Converts a bytearray to a string containing the hexadecimal values.

```
string baToHexString(bytearray baValue)
```

Parameter	Description
baValue	Bytearray value
Returns	A hexadecimal string

baToStr

Converts a bytearray to a string.

```
string baToStr
 (bytearray array,
 string charEncoding) //Optional
```

Parameter	Description
array	The bytearray to convert
charEncoding	The character encoding to use. If no encoding is specified, ISO-8859-1 (Latin 1) is used. If you specify character encoding, it must match the character encoding included in the standard charsets found in https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html or the extended charset of the target Java implementation. If you specify a character encoding that cannot be found, a compilation error occurs.
Returns	The result string

dateToString

Converts a date to a string and assigns the result to a string variable (or field). Returns true if the conversion succeeded. The only time the conversion can fail, is if the date is null or the format is invalid.

```
boolean dateToString
 (string stringVar,
 date dateValue,
 string format) //optional
```

Parameter	Description
stringVar	String identifier to set.
dateValue	The date value.
format	If no format string is specified, the default system format is used as specified in the property `mz.server.dateformat`. For further information about this property, see 2.6.1 Cell Properties in the System Administrator's Guide. The date formatting is based on standard Java syntax. For further information, see http://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html.
Returns	`true` or `false`.

strToBA

Converts a string to a bytearray and assigns the result to a bytearray variable (or field).

```
void strToBA
 (bytearray array,
 string stringValue,
 string charEncoding) //Optional
```

Parameter	Description
array	The resulting bytearray
stringValue	The string value
charEncoding	The character encoding to use. If no encoding is specified, ISO-8859-1 (Latin 1) is used. If you specify character encoding, it must match the character encoding included in the standard charsets found in https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html or the extended charset of the target Java implementation. If you specify a character encoding that cannot be found, a compilation error occurs.
Returns	Nothing

strToBigInt

Converts a string to a big integer. If the conversion was successful `true` is returned.

If one of the keywords `dec` or `hex` are specified, the string is assumed to be decimal or hexadecimal respectively. Decimal is the default.

```
boolean strToBigInt
 (bigint bigintVar,
 string stringValue,
 dec|hex) //Optional
```

Parameter	Description
bigintVar	Big integer variable to set
stringValue	String value
Returns	`true` or `false`

strToDate

Converts a string to a date and assigns the result to a date variable (or field). Returns `true` if the conversion succeeded.

You can enable lenient interpretation of the date/time in the string to be converted by setting the Cell property `mz.drdate.lenient` to true in the `cell.conf`. With lenient interpretation, a date such as "January 32, 2016" will be treated as being equivalent to the 31nd day after January 1, 2016. With strict (non-lenient) interpretation, an invalid date will cause the function to leave the submitted date variable unchanged. The default value of `mz.drdate.lenient` is `false`.

```
boolean strToDate
 (date dateVar,
 string stringValue,
 string format, //Optional
 string timeZone) //Optional
```

Parameter	Description
dateVar	Date identifier to set
stringValue	Date value

format	If no format string is specified, the default system format is used as specified in `$MZ_HOME/common/config/cell/default/master/cell.conf`, the `mz.server.dateformat` property.
	
	The date formatting is based on standard Java syntax. For further information, see http://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html.
	
	Note!
	
	Even though the syntax conforms to `SimpleDateFormat`, it is not directly based on this class.
	
	You can enable date format handling based on the `SimpleDateFormat` class in the installed Java version by setting the Execution Context property `mz.use.drdateformat` to `false`. This enables use of additional patterns that are available in the installed Java version.
	
	`$ mzsh topo set topo://container:<container>/pico:<pico name>/val:config.properties. mz.use.drdateformat false`
	
	When `mz.use.drdateformat` is set to `true` (default), the function does not apply the timezone offset at conversion. The timezone can be specified but the date is not modified.
	
	Note!
	
	Unlike `SimpleDateFormat` the `strToDate` function does not accept the Locale parameter and uses the default JVM locale instead, which may result in parsing errors.
	
	If this is an issue, it can be solved by setting the locale to US_en in the JVM arguments of the relevant EC/ECSA.
	
	`$ mzsh topo set topo://container:<container>/pico:<pico name>/obj:config.jvmargs \ 'usercountry: ["-Duser.country=US"] userlanguage: ["-Duser.language=en"]'`

timeZone	An optional string stating the time zone to set. It is recommended that you specify the time zone id using the long format, e g "America/St_Johns". It is possible to also use the abbreviated format, e g "PST". However, this can lead to ambiguous results and should be avoided. If an invalid time zone format is entered, no error is returned. Instead, the time zone is automatically set to "GMT". If the time zone is specified in the `stringValue` , it overrides the `timeZone` parameter.
Returns	`true` or `false`.

strToDouble

Converts a decimal string to a double and assigns the result to a double variable (or field). Returns `true` if the conversion succeeded.

```
boolean strToDouble
 (double doubleVar,
 string stringValue)
```

Parameter	Description
doubleVar	Double identifier to set.
stringValue	String value.
Returns	`true` or `false`.

strToFloat

Converts a decimal string to a float and assigns the result to a float variable (or field). Returns `true` if the conversion succeeded.

```
boolean strToFloat
 (float floatVar,
 string stringValue)
```

Parameter	Description
floatVar	Float identifier to set
stringValue	String value
Returns	`true` or `false`

strToInt

Converts a decimal or hexadecimal string to an integer and assigns the result to an integer variable (or field). Returns `true` if the conversion succeeded.

If one of the keywords `dec` or `hex` are specified, the string is assumed to be decimal or hexadecimal respectively. Decimal is the default.

```
boolean strToInt
 (int intVar,
 string stringValue,
 dec|hex) //Optional
```

Parameter	Description
intVar	Integer identifier to set.
stringValue	String value.
Returns	`true` or `false`.

Example - Using strToInt

Provided that the field `CallTime` is a string containing a hexadecimal string, it converts its content to an integer.

```
int a;
int b;
strToInt( a, CallTime, hex );
strToInt( b, "12345" ); /* dec is default */
```

strToLong

Converts a decimal or hexadecimal string to a long and assigns the result to an long variable (or field). Returns `true` if the conversion succeeded.

If one of the keywords `dec` or `hex` are specified, the string is assumed to be decimal or hexadecimal respectively. Decimal is the default.

```
boolean strToLong
 (long longVar,
 string stringValue,
 dec|hex) //Optional
```

Parameter	Description
longVar	Long identifier to set
stringValue	String value
Returns	`true` or `false`

strToBigDec

Converts a decimal or hexadecimal string to a BigDecimal and assigns the result to a BigDecimal variable (or field). Returns true if the conversion succeeded.

```
boolean strToBigDec
 (bigdec bigdecVar,
 string stringValue)
```

Parameter	Description
bigdecVar	BigDecimal identifier to set
stringValue	String value
Returns	`true` or `false`

udrToString

Converts a UDR to a string. Each UDR will be preceded with the internal class name, and each of its field will be preceded with the field name.

```
string udrToString(drudr myUDR)
```

Parameter	Description
myUDR	The UDR to convert

Returns	A string containing the UDR fields. The output will be in the following format:
	`Field values for:`
	`<internal UDR reference>`
	`field1: <value>`
	`field2: <value>`

2.9 Type Comparison Functions

instanceOf

Returns `true` if the value is of the specified type.

```
boolean instanceOf
 ( any  myValue ,
 any  myType  )
```

Parameter	Description
myValue	The value to evaluate. Can be of any type, even primitive types.
myType	Any type to evaluate against
Returns	true or false

Example - Using instanceOf

```
consume{
 if ( instanceOf( input, module1.type1 ) ) {
        udrRoute( ( module1.type1 )input, "link_1" );
 } else if ( instanceOf( input, module2.type2 ) ) {
        udrRoute( ( module2.type2 )input, "link_2" );
 } else {
        // handle error case
 }
}
```

You can use variables of the types list and map in myValue, but the the contained types are not tested. In order to avoid confusion, it is recommended to always use any as a contained type in the myType argument.

Example - Using instanceOf with a contained type in the myType argument

```
consume {
    // The contained types are int and string.
    map<int, string> myMap = mapCreate(int, string);
    list<string> myList = listCreate(string);

    // The expressions below are evaluated to true,
    // since the contained types are not tested.

    if ( instanceOf( myMap, map<any,any>) ) {
       debug("This is a map.");
    }

    if ( instanceOf( myList, list<any>) ) {
       debug("This is a list.");
    }
 }
```

2.10 UDR Functions

udrAddError

Adds an error description message to the UDR, which will be visible from the Error Correction System Inspector window or the Data Veracity web ui, Error Code column. A UDR can have several error descriptions.

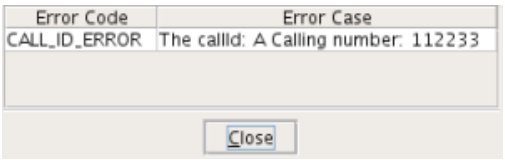
```
void udrAddError ( drudr myUDR, string ecsErrorCode, string myString )
```

Parameter	Description
myUDR	The UDR to add error message to
ecsErrorCode	An ECS Error Code as defined from the Error Correction System Inspector and Data Veracity web ui
myString	Any string to associate with the Error Code. This parameter comprises the Error Case found in the ECS or Data Veracity. Optional.
Returns	Nothing

Example - Using udrAddError

The following example appends error text to a UDR, which when displayed in ECS or Data Veracity will look like the following figure:

```
udrAddError( input, "CALL_ID_ERROR",
 "The callId: " +
 input.callId +
 " Calling number: " +
 input.anum );
```



Error Correction System - Inspector window

To display the Error Viewer, double-click the Error Code element for the specific UDR

udrClearErrors

Removes all error descriptions, added to a UDR. This is useful when collecting UDRs from ECS or Data Veracity for a new validation which may send them back to ECS or Data Veracity. Adding a new error description, if one already exists, will make it impossible to auto-assign the UDR to a reprocessing group when it arrives to either Data Veracity or the ECS.

```
void udrClearErrors( drudr myUDR )
```

Parameter	Description
myUDR	The UDR to clear errors from
Returns	Nothing

udrClone

Clones a UDR. It is used when having multiple routes leaving the agent, where any of the routes changes the content of a UDR. For complex UDR structures, this can be a heavy operation, and must be used with care.

```
drudr udrClone( drudr  myUDR  )
```

Parameter	Description
myUDR	The UDR to clone
Returns	A clone of *myUDR*

udrContainsError

Returns `true` if an error description has been added to the UDR.

```
boolean udrContainsError( drudr  myUDR  )
```

Parameter	Description
myUDR	The UDR to evaluate for error descriptions
Returns	`true` or `false`

udrCreate

Creates a new UDR of the specified type. All fields are included, except for the optional.

```
drudr udrCreate( type  UDRType  )
```

Parameter	Description
UDRType	A defined UDR type
Returns:	A UDR of the specified type

udrDecode

Decodes a bytearray into a list of UDRs, and returns an error message if the operation fails. To route the UDRs to a subsequent agent, loop through the resulting list routing each element individually.

```
string udrDecode
 ( string  decoderName ,
 list<drudr>  UDRlist ,
 bytearray  indata ,
 boolean  fullDecode  ) //Optional
```

Parameter	Description
decoderName	The name of a defined decoder
UDRlist	The list in which the resulting, decoded UDRs will be saved
indata	The input data to decode
fullDecode	States if Full Decode will be applied. By default, this parameter is set to `true`. For further information, see the Ultra Reference Guide.
Returns	A string containing nothing (`null`) if the operation succeeded, and an error message if it failed

Example - Using udrDecode

A list must be created, not only declared, previously used by `udrDecode`:

```
list<drudr> myList = listCreate(drudr);

if (udrDecode("myFolder.myFormat.myDecoder",
 myList, input) == null) {
 // Do something
} else {
 // Error handling
}
```

udrEncode

Encodes a UDR.

```
bytearray udrEncode
 ( string  encoderName ,
 drudr  myUDR  )
```

Parameter	Description
encoderName	The name of a defined encoder
myUDR	The UDR to encode
Returns	A bytearray

udrForceDecode

By default, a Decoder only evaluates record sizes and field termination marks. It does not read the field values to evaluate for validity. This is done for each field when it is actually accessed, either from a agent utilizing APL code, or from an Encoder.

The `udrForceDecode` function will decode each field within a UDR. If a field is not valid, the agent will abort. This function is equal to the Full Decode option in the Decoder window. Note that the function has a negative impact on performance, and must be used mainly for testing purposes.

```
string udrForceDecode
 ( drudr  myUDR ,
 boolean  abortOnFail  ) //Optional
```

Parameter	Description
myUDR	The UDR to fully decode
abortOnFail	An optional argument. If not stated, `true` is assumed. If set to `false`, the workflow does not abort if decoding fails.
Returns	If decoding succeeds, `null` is returned. If not - and `abortOnFail` is set to `false` - an error message is returned.

udrGetErrorCodes

Returns a list of strings containing all error codes added to the UDR.

```
list <string> udrGetErrorCodes( drudr  myUDR  )
```

Parameter	Description
myUDR	The UDR
Returns	A list of strings with all error codes added to the UDR

udrGetErrorMessages

Returns a list of strings containing all error messages added to the UDR.

```
list <string> udrGetErrorMessages( drudr  myUDR  )
```

Parameter	Description
myUDR	The UDR
Returns	A list of strings with all error messages added to the UDR

udrGetErrorsAsString

Returns a string containing all error information that has been added to the UDR.

```
string udrGetErrorsAsString( drudr  myUDR  )
```

Parameter	Descripton
myUDR	The UDR
Returns	A string with all error information added to the UDR

udrGetFields

Returns a list of the field names and field information in a UDR.

```
list<UDRFieldInfo> udrGetFields
 ( drudr myUDR )
```

Parameter	Description
myUDR	The UDR.
Returns	`list<UDRFieldInfo>` A list of the fields in the UDR and the field information. See the information below on `UDRFieldInfo`.

`UDRFieldInfo` is the UDR that is populated with the field information of the UDR that contains a list of fields.

The following fields are included in `UDRFieldInfo`:

Field	Description
`fieldName (string)`	The name of the field
`fieldType (string)`	The field type, e g string, int, double etc.
`isOptional (boolean)`	This indicates if the field is optional or not.
`isReadOnly (boolean)`	This indicates if the field is read-only or not.

udrGetValue

Returns the value of `fieldName` in the `myUDR`.

```
any udrGetValue
 ( drudr  myUDR ,
 string  fieldname  )
```

Parameter	Description
`myUDR`	The UDR containing the field of interest
`fieldname`	A string, exactly matching the name of an existing field. If the field is OPTIONAL and is not present, the function returns `null`. Constructed field names, that is subUDR.subField are allowed. If a field in the path is not present, `null` is returned in this case as well.
Returns	Any, depending on the field type

udrIsPresent

Returns true if the UDR field is present.

```
boolean udrIsPresent(identifier  field  )
```

Parameter	Description
`field`	The full field name including the UDR name. For instance, `input.IN.moRecord`.
Returns	`true` or `false`. If a field is optional and not present, accessing the field will return the default null value that is null for object types, 0 for numeric types and false for boolean types.

Example - Using udrIsPresent

The function is recursive. If the optional field *`myField`* is to be accessed, and it is nested in three levels: `fieldA.fieldB.myfield`, and all levels are optional, only one `if`-statement is required:

```
if ( udrIsPresent( input.fieldA.fieldB.myField ) ) {
 // Some kind of operation.
}
```

udrMass Functions

The udrMass functions enable you to manage a decoder which purpose is to process large amounts of collected UDRs, while maintaining control over UDRs tailing bytes in between calls to **consume** .

The udrMass functions include:

- udrMassCreateDecoder
- udrMassClearBuffer
- udrMassDecode
- udrMassEOFCheck
- udrMassGetDataTail

udrMassCreateDecoder

Creates a decoder object.

```
any udrMassCreateDecoder(string  decoderName )
```

Parameter	Description
decoderName	The name of the decoder object.
Returns:	A decoder object.

Example - Using udrMass* APL functions

```
any myDecoder;
initialize {
 myDecoder = udrMassCreateDecoder("Default.ascii.asciiFile");
}
drain {
 list<drudr> myList = listCreate(drudr);
 string msg = udrMassEOFCheck(myDecoder, myList, false);
 if (msg == null) {
 // Do something
 } else {
 // Error handling
 }
}
consume {
 list<drudr> myList = listCreate(drudr);
 string msg = udrMassDecode(myDecoder, myList, input, false);
 if (msg == null) {
 // Do something
 } else {
 // Error handling
 }
}
```

udrMassClearBuffer

The `udrMassClearBuffer` function clears the remaining buffer in the decoder.

```
void udrMassClearBuffer(any theDecoder)
```

Parameter	Description
theDecoder	The decoder object that `udrMassCreateDecoder` generates
Returns:	Nothing

udrMassDecode

The `udrMassDecode` function Decodes the bytearray input data by using the decoder, and generates it as a list.

> **Note!**
>
> See the example above, Using udrMass* APL Functions.

```
string udrMassDecode(
 any  theDecoder ,
 list<drudr> myList ,
 bytearray  inputData ,
 boolean  forceDecode )
```

Parameter	Description
`theDecoder`	The decoder object that `udrMassCreateDecoder` generates
`myList`	The product of the encoding process is saved as a list of UDRs.
`inputData`	The input data that is to be encoded into UDRs
`forceDecode`	Use `true` to force decoding (a full decode).
Returns:	If decoding fails, an error message is returned. Otherwise, returns `null`.

udrMassEOFCheck

The udrMassEOFCheck function attempts to decode any remaining bytes that are left behind and stored in the decoder. If the function executes and decoding is successful, the procedure sends `endBatch` to the decoder, the returned string is empty, and the *`myList`* parameter contains the UDRs. Otherwise, an error message is generated.

```
string udrMassEOFCheck(
 any theDecoder,
 list<drudr> myList,
 boolean forceDecode)
```

Parameter	Description
`theDecoder`	The decoder object that `udrMassCreateDecoder` generates
`myList`	The data that `udrMassEOFCheck` decodes
`forceDecode`	Use `true` to force decoding (a full decode).
Returns:	If decoding fails, an error message is returned. Otherwise, returns `null`.

udrMassGetDataTail

The udrMassGetDataTail function makes the decoder generate the remaining bytes that it holds. The bytes are still stored in the decoder.

> **Note!**
>
> This function enables you to investigate the data that is stored in the decoder. Do not use it to reset the internal array of the remaining bytes.

```
bytearray udrMassGetDataTail(any  theDecoder )
```

Parameter	Description
theDecoder	The decoder object that `udrMassCreateDecoder` generates
Returns:	The raw bytearray data that the decoder in `udrMassDecode` uses as input. For further information see the section above, `udrMassDecode`.

udrRoute

The `udrRoute` function sends a UDR to a workflow route.

If no call to the route function is made, the agent will not have any outgoing routes in the Workflow Editor environment - it will not be possible to link to a subsequent agent.

The function routes a UDR to all the output routes, or a named output route. If the *clone* keyword is used, the UDR is cloned before output.

> **Note!**
>
> The `udrRoute` function cannot be used within global APL scripts, that is in code saved in the APL Code Editor.

```
void udrRoute
 ( drudr  myUDR ,
 string  routeName , //Optional
  clone  ) //Optional
```

Parameter	Description
myUDR	The UDR to send on
routeName	The route on which to send the UDR
clone	The UDR is cloned before routing on the output/outputs.
Returns	Nothing

udrSetValue

Sets the field named *fieldName* in the UDR *myUDR* to *value* .

```
void udrSetValue
 ( drudr  myUDR ,
 string  fieldName ,
 any  value  )
```

Parameter	Description
myUDR	The UDR, for which to set the field value
fieldName	A string, exactly matching an existing field. Constructed field names in the form of subUDR.subField are allowed. Trying to set a non-existing field will cause runtime error.
value	A value depending on the field type
Returns	Nothing

udrUnsetPresent

In the output files the Encoder maps by default, null fields as present but empty. This can be avoided by clearing a flag, Is Present, using the `udrUnsetPresent` function.

The field can be changed to `Is Present` again either via udrSetValue (see the section above, udrSetValue) or regular assignment of fields.

```
void udrUnsetPresent(identifier field )
```

Parameter	Description
`field`	The full name of the optional field including the UDR name. For instance, `input.IN.moRecord`.
Returns	Nothing

2.11 UUID Functions

This section describes functions that facilitate creation and use of immutable universally unique identifiers (UUID).

For general information about UUIDs, see https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html.

For information about the UUID type in MediationZone, see 1.5 Data types

uuidCreateFromHexString

This function takes a string representation of a UUID and converts it to the `uuid` type.

```
uuid uuidCreateFromHexString ( string aString )
```

Parameter	Description
uuidString	A string representation of a UUID
Returns	A UUID based on the input string The format of the string is validated but not the content. The function returns null if the format of the string is incorrect.

Example - uuidCreateFromHexString

The following example converts a string to a UUID:

```
uuid getUuid (string s) {
 uuid u = uuidCreateFromHexString(s);
 if(u==null) {
  //Incorrect UUID format
 }
 return u;
}
```

uuidCreateRandom

This function generates a random UUID.

```
uuid uuidCreateRandom ( void )
```

Parameter	Description
Returns	A version 4 UUID (randomly generated UUID)

Example - uuidCreateRandom

The following example creates a random UUID:

```
uuid getUuid() {
    return uuidCreateRandom();
}
```

uuidGetVersion

This function returns the version of a UUID. The version number describes the type of the UUID, e g time-based, DCE security, name-based, and randomly generated UUID. For instance, a UUID generated by the function uuidCreateRandom is 4 (randomly generated UUID).

```
int uuidGetVersion ( uuid aUuid )
```

Parameter	Description
aUuid	The UUID for which you want to retrieve the version.
Returns	The version of the UUID, or -1 if the UUID in the argument is null.

Example - uuidGetVersion

The following example retrieves the version from a UUID:

```
uuid getUuid (string s) {
    uuid u = uuidCreateFromHexString(s);
    if(u==null) {
        debug("Incorrect UUID format");
    } else if (uuidGetVersion(u)<0) {
        debug("Invalid UUID");
    }
    return u;
}
```

uuidString

This function converts a UUID to a string.

```
string uuidString ( aUuid )
```

Parameter	Description
aUuid	The UUID that you want to convert
Returns	The UUID represented as a string

Example - uuidString

The result of the following two examples are identical:

```
uuid aUuid = uuidCreateRandom();
string s = (string) aUuid;
```

```
uuid aUuid = uuidCreateRandom();
string s = uuidString(aUuid);
```

2.12 OAuth Functions

This section describes functions that relates to OAuth operations.

validateJwt

Validates an incoming OAuth JWT.

```
JwtValidation validateJwt(
string openIdServer,
string token,
map <string, any> claimsToValidate, //Optional
string algorithm //Optional)
```

Parameters

Parameter	Description
openIdServer	The destination URL of the token to be verified.
token	The value of the token.
claimsToValidate	An optional map field to declare the claims as well as the corresponding value to validate against the token.
algorithm	An optional field to verify the signing algorithm used by the token. The possible values could be RSA256, RSA384, RSA512, ECDSA256, ECDSA384, ECDSA512. By default, the APL function uses RSA256.
Returns	An error message on validation failure. Null on validation success.

Example

Example of the validateJwt function with claims and algorithm optional values populated.

```
string token =
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Im5PbzNaRHJPRFhFSzFqS1doWHNsSFJfS1hFZyIsImtpZCI6Im5Pbz";
map<string, any> claimsToValidate = mapCreate(string, any);
mapSet(claimsToValidate, "appidacr", "2");
mapSet(claimsToValidate, "aud", "ae47e8fd-b2be-4626-a7b5-19d28961ba1e");
string error_message = JwtValidation.validateJwt("https://10.60.10.30/endpoint", token,
claimsToValidate, "RSA512");
```

3. Aggregation Functions

This chapter describes the functions, function blocks, and variables that are available for use in the Aggregation agent APL code.

3.1 Function Blocks

Use the following functions blocks in the Aggregation agent for general processing:

- `initialize` in Batch and Real-Time
- `deinitialize` in Batch and Real-Time
- `drain` only in Batch
- `beginBatch` only in Batch
- `endBatch` only in Batch
- `cancelBatch` only in Batch
- `commit` only in Batch
- `rollback` only in Batch

Note!

The `session` variable is not available within the function blocks listed above. For more information about these function blocks, see 1.4 Function Blocks.

Use the following function blocks, in which the session variable is available, for Aggregation specific functions.

- `consume` in Batch and Real-Time
- `command` only in Real-time
- `sessionInit` in Batch and Real-Time
- `timeout` in Batch and Real-Time

consume - Batch and Real-Time

The `consume` block is called each time a UDR that matches a session, or causes a new session to be created, arrives. Thus, the consume block is not necessarily called for each arriving UDR. The UDR will be available through the input variable and the session through the session variable.

command - Real-Time Only

The command block is called for all flushed sessions. The flush action is initiated with an Agent Command from the Command Tab of the Aggregation agent in the Workflow Monitor or through the Command Line Tool mzsh. Within this block the instruction variable is available and contains a string, optionally forwarded by the user. For further information, see 3.5 Flush Sessions and 3.2 Variables.

sessionInit - Batch and Real-Time

The `sessionInit` block is called each time a new session is created. The UDR that caused the session to be created will be available through the input variable and the new session through the session variable. The sessionInit block is optional. It is, however, recommended that you use this block since it makes the code easier to follow.

timeout - Batch and Real-Time

The `timeout` block handles sessions that for some reason have been left hanging in the system. For instance, if the stop UDR was not received within a timeout interval.

Between `drain` and `endBatch` for each batch passing the Aggregation agent (batch workflow) or as defined in the agent configuration (real-time workflow), the `timeout` block is called for all outdated sessions. The outdated session will be available through the `session` variable.

Initially, make sure the code in the timeout block handles the session properly, then there are two alternatives on how to handle the session:

- If the session is not needed anymore, remove the session with `sessionRemove`.
- If relevant UDRs are still expected, set a new timeout with `sessionTimeout`.

If none of them are used, outdated sessions that may never be used again will remain in the system forever and will always be active for comparison against incoming data, which will have a negative impact on performance.

If there are old sessions with no timeout set, correct the `timeout` block and then open the Aggregation Session Inspector. If there is no use for the sessions, they may be deleted, otherwise set a new timeout for these sessions and they will be handled by the timeout block as soon as a new batch is processed by the Aggregation agent.

3.2 Variables

The following built-in variables are specific to the Aggregation agent.

Variable	Description
session - Batch and Real-Time	A reference to the current session to be used to access variables defined in the Association tab. A session will remain in the database until manually removed. A route to ECS or an alternative route will not remove it. The variable is available in the `consume`, `sessionInit` and `timeout` function blocks. **Example - Using session** `session.duration = input.duration + session.duration;`
instruction - Real-Time Only	An optionally inserted string, belonging to the currently flushed session. The variable is available in the `command` function block only. **Example - Using instruction** `input.info = instruction;`

3.3 Functions

The following functions may be used in the aggregation APL code. All functions described in the Analysis Agent section in the Desktop user's guide are also available.

sessionMirror - Couchbase Storage Only

This function retrieves a list of sessions from the storage specified by the Mirror Profile.

```
list<session> sessionMirrors ( session)
```

Parameter	Description
session	This is a built-in variable. For information about `session`, see 3.2 Variables.
Returns	A list of mirror sessions. **Note!** The return value always contains one element. The use of the `list` data type serves to support future requirements.

Example - Using sessionMirror

```
sessionInit {
    session.anum = input.anum;
    session.bnum = input.bnum;
    session.total_duration = 0;
}

consume {
    session.total_duration = session.total_duration +  input.duration;
    sessionTimeout(session, 3600);
    //Get mirror sessions
    list<any> allMirrorSessions = sessionMirrors(session);
// Do not use sessionMirrorLastErrorCode
  // in asynchronous mode.
    if (0 == sessionMirrorLastErrorCode()) {
        Default.session.session aggregateTotal =  udrClone(session);
        aggregateTotal.total_duration =  session.total_duration +  getTotalDuration
(allMirrorSessions);
        debug(aggregateTotal.total_duration);
    } else {
       logError("sessionMirrorLastErrorCode:" + sessionMirrorLastErrorCode() +
"sessionMirrorLastErrorMessage:" +sessionMirrorLastErrorMessage());
    }
}

timeout{
    list<any> allMirrorSessions = sessionMirrors(session);
// Do not use sessionMirrorLastErrorCode
 // in asynchronous mode.
    if (0 == sessionMirrorLastErrorCode()) {
        Default.session.session aggregateTotal =  udrClone(session);
        aggregateTotal.total_duration = session.total_duration +  getTotalDuration(allMirrorSessions);
        debug(aggregateTotal);
    } else {
        logError("sessionMirrorLastErrorCode:" + sessionMirrorLastErrorCode() +
"sessionMirrorLastErrorMessage:" +sessionMirrorLastErrorMessage());
    }
}

int getTotalDuration( list<any> allSessions) {
    int total_duration = 0;
    if (1 == listSize(allSessions)) {
        Default.session.session s =  (Default.session.session)listGet(allSessions, 0);
        total_duration = s.total_duration;
    }
    return total_duration;
```

sessionMirrorLastErrorCode - Couchbase Storage Only

This function returns the error code from the last call to `sessionMirrors` function.

Note!

You should only use `sessionMirrorLastErrorCode` when asynchronous mode is disabled, which is the default setting. In asynchronous mode, errors are handled by the Aggregation agent and are not available in APL. For more information about using asynchronous mode, see 9.3 Aggregation Agentin the Desktop user's guide.

Since `sessionMirrors` does not return error codes and generally does not cause runtime errors, `sessionMirrorLastErrorCode` or `sessionMirrorLastErrorMessage` should be called after each operation to validate success.

```
int sessionMirrorLastErrorCode ()
```

Parameter	Description
Returns	0 - All operations were successful. 1 - An operation resulted in an unknown error. 4 - A temporary failure occurred. 5 - An operation timed out.

sessionMirrorLastErrorMessage - Couchbase Storage Only

This function returns the error message from the last call to `sessionMirrors` function.

> **Note!**
>
> You should only use `sessionMirrorLastErrorMessage` when asynchronous mode is disabled, which is the default setting. In asynchronous mode, errors are handled by the Aggregation agent and are not available in APL. For more information about using asynchronous mode, see 9.3 Aggregation Agent in the Desktop user's guide.

Since `sessionMirrors` does not return error codes and generally does not cause runtime errors, `sessionMirrorLastErrorCode` or `sessionMirrorLastErrorMessage` should be called after each operation to validate success.

```
string sessionMirrorLastErrorMessage()
```

Parameter	Description
Returns	An error message, or null if no error has occurred

Example - Using sessionMirrorLastErrorMessage

```
sessionInit {
    list<any> allMirrorSessions = sessionMirrors(session);
    if (0 == sessionMirrorLastErrorCode()) {
        //No error
    } else {
        logError("sessionMirrorLastErrorCode:" + sessionMirrorLastErrorCode() +
"sessionMirrorLastErrorMessage:" +sessionMirrorLastErrorMessage());
    }
}
```

sessionRemove - Batch and Real-Time

Removes the current session.

This function must be called when a session is considered to be closed and a new UDR has been emitted, or when a session is considered to be closed due to an expired timeout.

```
sessionRemove(session)
```

Parameter	Description
session	This is a built-in variable. For information about `session`, see 3.2 Variables.

sessionTimeout - Batch and Real-Time

Sets a time limit to call the timeout function block or, if not available, the `consume` block, within the number of seconds specified, or at an exact time. Only one timeout per session may exist, thus if a subsequent call to this function is made, the old timeout is cancelled.

Timeouts are not always called at the exact time stated. See 9.3 Aggregation Agent in the Desktop user's guide for details on when the time limit is actually invoked.

The timeout for a session is cleared each time the consume or timeout function blocks are called.

```
void sessionTimeout(DRUDR session, long timeOutSeconds)
```

Parameter	Description
session	This is a built-in variable. For information about `session`, see 3.2 Variables.
timeoutSeconds	Number of seconds. If set to 0 (zero) or? `null`, the timeout is removed. This is? useful if it is desired to reset the timeout to the default? timeout. If set to a negative value, the session will be outdated

```
sessionTimeout(session, date  timeoutDate )
```

Parameter	Description
session	This is a built-in variable. For information about `session`, see 3.2 Variables.
timeoutDate	An exact date. If set to `null`, the timeout is removed. This is useful if it is desired to reset the timeout to the default timeout. If set to a passed date, the session will be outdated.

3.4 Session Iterator Functions

This section describes the four different APL sessionIterator functions. They can be used both for Real Time and Batch workflows. An iterator is started with sessionIteratorCreate, and is ended by using sessionIteratorNext or aborted using sessionIteratorDestroy.

- sessionIteratorCreate
- sessionIteratorNext
- sessionIteratorErrorCount
- sessionIteratorDestroy

Iterating over all sessions in storage does not take any locks. This means that if a workflow updates the storage at the same time as the iteration is ongoing, sessions can be missed in the search or the same sessions can be returned twice. There is an "Error Count" on the iterator that can be used when this type of inconsistency has been detected. This is the same as the type of inconsistencies that can be detected in the Aggregation Session Inspector where sessions are not accessible.

sessionIteratorCreate

Creates a new iterator that can be used in the other three functions.

```
any sessionIteratorCreate( string profileName )
```

sessionIteratorNext

Returns the next session or null if the full session storage has been traversed.

```
drudr sessionIteratorNext( any iterator )
```

sessionIteratorErrorCount

Returns the number of detected errors during the iteration. This is a fairly normal condition when the storage is updated or removed while the iteration is ongoing. Each error will typically mean that one session was missed, but it can also mean a larger number of missed sessions if the storage was moved.

If the session iterator starts iterating over the storage on one ec, and at the same time a workflow using the profile starts on another ec, the storage is removed from the first ec and started up at the second ec and the iterator will abort. This will be logged in system log as well as the error count being increased.

```
int sessionIteratorErrorCount( any iterator )
```

sessionIteratorDestroy

Destroy the iterator to release resources to avoid a memory leak due to iterators being left in live state. This is not needed if the iteration is run to the end.

```
void sessionIteratorDestroy( any iterator )
```

Best practice to implement sessionIterators in APL

An example below of searching for a session that matches an "intField" value.
"QueryData" is here the session type of the "Default.queryagg" profile.

```
QueryData findSession(int intFieldValue) {
    int retries = 0;
    any iter = null;
    // For a real scenario, consider limiting number of retries
    while (true) {
        iter = sessionIteratorCreate("Default.queryagg");
        QueryData qd = (QueryData) sessionIteratorNext(iter);
        while (qd != null) {
            if (qd.intField == intFieldValue) {
                // Done!
                sessionIteratorDestroy(iter);
                return qd;
            }
            qd = (QueryData) sessionIteratorNext(iter);
        }
        if (sessionIteratorErrorCount(iter) > 0) {
            // We did not find the session and had errors -> retry
            ++retries;
        } else {
            // No errors, but it does not seem to exist
            return null;
        }
    }
}
```

We simply retry the search whenever an error is detected. Note also the use of sessionIteratorDestroy which should be used whenever an iterator is not run to the end.

> **Performance Note!**
>
> Considerably better performance will be achieved by running the lookups on the same EC as the storage service. This means that if you run the lookup at the same time as a workflow using the storage is running on a different EC, then it will be slower. It will still be better performance than using the Aggregation Session Inspector, and there will be no load on the platform.

3.5 Flush Sessions

This section describes how to flush aggregation sessions in real-time and batch workflows.

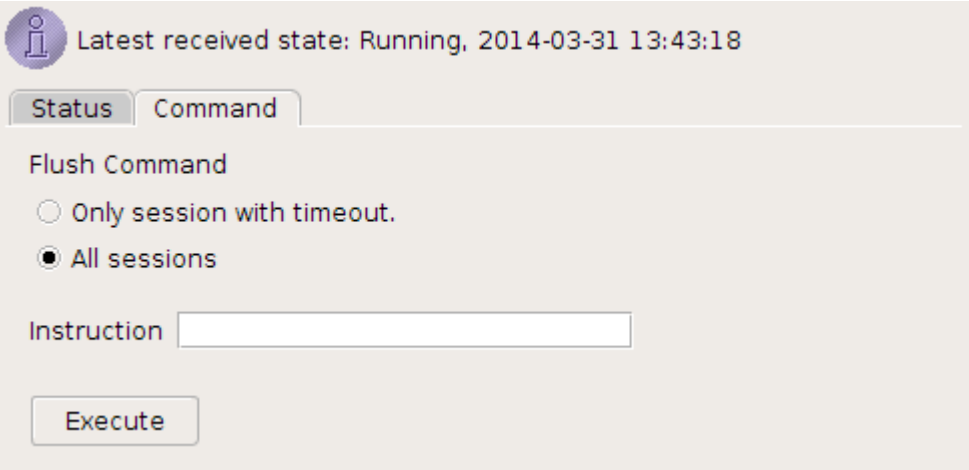
command - Real-Time Only

To flush Real-time workflow sessions, run an agent command from the **Command** Tab of the Aggregation agent in the Workflow Monitor or use the Command Line Tool *mzsh*. There is a choice of applying the command on all sessions or only on those with a timeout set. There is also an option of attaching information in a string to be available within the command function block in the Aggregation agent.

> **Note!**
>
> You cannot flush sessions that are stored in Couchbase.

Syntax to run the `command` function block from `mzsh`:

```
$ mzsh wfcommand <Workflow Name> <Agent Name> <"true"|"false"> <instruction>
```



Aggregation agent in Workflow Monitor - Command tab

> **Note!**
>
> The command function block works in the same way as the timeout function block, that is, code has to be written to route the session result and to remove the session.

Syntax to run the command function block from *mzsh*:

```
$ mzsh wfcommand <Workflow Name> <Agent Name> <"true"|"false"> <instruction>
```

Parameter	Description
`Workflow Name`	The name of the workflow that contains the Aggregation agent
`Agent Name`	The name of the Aggregation agent in the workflow
`"true"|"false"`	Specifies what sessions to apply the `command` block for. `true` means apply on all sessions, while `false` means apply only on sessions with timeout set. To have the sessions removed, the `command` block must issue `sessionRemove (session)`.
`instruction`	This is an optional string. If specified, the string will be available within the `command` function block in the Aggregation agent through the *`instruction`* variable. Refer to 3.2 Variables for further information.

Example - Using command

A simple APL example of `command`:

```
command {
  OutputUDR finalUDR = udrCreate( OutputUDR );
  finalUDR.user = session.user;
  finalUDR.IPAddress = (string)session.IPAddress;
  finalUDR.downloadedBytes = session.downloadedBytes;
  finalUDR.uploadedBytes = session.uploadedBytes;
  if (instruction == "extra") { finalUDR.extraBytes = session.extraBytes; }
  udrRoute( finalUDR );
  sessionRemove(session);
}
```

Batch Aggregation APL Flush Function

This section describes the APL function `aggregationHintFlushSessions`, which is used to timeout all stored sessions that have a timeout value in a batch Aggregation agent. The function is included in the package, *Batch Aggregation APL Flush Function*.

The APL function `aggregationHintFlushSessions` can be called from the Analysis APL code or Aggregation APL code in batch workflows.

The function is used to indicate that the `timeout` function block in the APL code for the specified Aggregation agent shall be called for all the sessions in the storage that have a timeout value. The `timeout` block execution will be performed after the `drain` function block has been executed in the specified Aggregation agent. The call to the function will not be remembered between batches.

Note!

Only sessions that have a timeout value will timeout even if the function `aggregationHintFlushSessions` has been called. To make sure that all sessions have a timeout value, set the Aggregation agent property If Timeout is Missing to Default or Abort . The Aggregation agent property If Timeout is Missing can be found under the tab called General in the Aggregation agent configuration.

When executed, the function will throw an exception if it is called from an illegal location or if the agent named *`aggregationAgentName`* is not an Aggregation agent in the workflow.

Function Definition:

```
void aggregationHintFlushSessions(string aggregationAgentName )
```

Parameter	Description
`aggregationAgentName`	The name of an Aggregation agent in the batch workflow

Example - Using aggregationHintFlushSessions

For example, consider a batch workflow containing one Aggregation agent named "Agg" which has the Aggregation agent property If Timeout is Missing set to Default. Also, consider that the `beginBatch` function block and the `timeout` function block in the APL code for the Aggregation agent is defined as in the following code:

```
beginBatch {
    aggregationHintFlushSessions("Agg");
}

timeout {
    sessionRemove(session);
}
```

Finally, consider that there are 10 sessions, with timeout values in the distant future, in the storage when the end of the `drain` function block in the Aggregation agent has been reached. Then the `timeout` function block will be executed 10 times, one time for every session in the storage, and thus all the sessions in the storage will be removed.

Note!

A workflow using `aggregationHintFlushSession` may get stuck in a loop if all sessions are not removed in the `timeout` block.

4. APL Collection Strategy Functions

APL Collection Strategy configurations are used on top of pre-defined collection strategies to customize how files are collected from the local file system.

> **Note!**
>
> The APL function blocks that are described in this chapter are automatically invoked from the collection agent and cannot be called manually.

The following functions blocks may be used in the APL Collection Strategy code.

- `initialize`
- `deinitialize`
- `prepareBaseDirList`
- `accept`
- `filterFiles`
- `preFileCollection`
- `postFileCollection`
- `begin`
- `commit`
- `rollback`

The `initialize`, `deinitialize`, `begin`, `commit`, and `rollback` function blocks are equivalent to the corresponding APL function blocks described in 1.4 Function Blocks.

The collection specific functions blocks are invoked in the following order:

1. `prepareBaseDirList`
2. `accept`
3. `filterFiles`
4. `preFileCollection`
5. `postFileCollection`

> **Note!**
>
> During run-time, when each of the functions blocks are invoked, the workflow first runs the function blocks base part and then it executes your APL extension code.
>
> The following APL functions cannot be used within an APL Collection Strategy configuration.
>
> - `udrRoute`
> - `mimSet`
> - `mimPublish`
> - `cancelBatch`
> - `hintEndBatch`
>
> In the following APL functions you cannot assign a persistent variable with a value. For information about persistent variables, see 1.2 Variables and Variable Scope
>
> - `initialize`
> - `deinitialize`
> - `commit`
> - `rollback`

initialize

The `initialize` function block is used to prepare for the collection and set up an environment to be used in a later stage, when the batch workflow is executed.

```
void initialize()
```

deinitialize

The `deinitialize` function block is used to clean-up and close resources.

```
void deinitialize()
```

prepareBaseDirList

The `prepareBaseDirList` function block prepares for a list of base directories. A base directory is the directory that includes the files to be collected. This directory is defined when configuring an agent's Base Collection Strategy. Base directory paths can be added, removed, and modified.

The `prepareBaseDirList` function block is invoked by the Collection agent right after the collection preparation activities that were initiated by the `initialize` function.

```
void prepareBaseDirList( list<string>  dirList  )
```

Parameter	Description
`dirList`	This parameter is used to define one or more collection sources. The collection order of the files is defined using the `filterFiles` function. The *`dirList`* parameter refers to a list that, by default, contains the directory that is defined when configuring an agent's Base Collection Strategy. The *`dirList`* parameter is an in/out parameter that serves both as input and output values for the `prepareBaseDirList` function.

In the following example, `prepareBaseDirList` adds a subdirectory named `sub` to the directory that is already on the base directories list. For example: If `/home/in` is a directory that is on the directory list, the `prepareBaseDirList` function adds `/home/in/sub` to the directory list.

Example - Using prepareBaseDirList

```
void prepareBaseDirList(list<string> dirList) {
    string dir = listGet(dirList, 0);
    string subDir = dir + "/sub";
    listAdd(dirList, subDir);
}
```

accept

The collection agent processes the base directory and its included files and creates an internal file list including all files to be collected. The `accept` function block is invoked by the collection agent each time a new file is about to be added to the list, and, based on each file's fileInfo object, either accepts or rejects the file.

```
boolean accept
( FileInfo  file  )
```

Parameter	Description
file	The file parameter includes the properties of the file to collect or a directory where files are stored. The FileInfo object includes: **Item** / **Description** `isDirectory(boolean)` — Set to True if FileInfo represents a directory. `isFile(boolean)` — Set to True if FileInfo represents a file. `name(string)` — The name of the file or directory `size(long)` — The size of the file or directory `timestamp(long)` — The timestamp for when the file or directory was last modified
Returns	True if the file shall be added to the list of files that are about to be collected

Example - Using accept

In this example, only files that have a name that starts with "INFILE_ascii" are collected.

```
boolean accept(FileInfo file) {
    if(file.isFile) {
        if ( strStartsWith(file.name, "INFILE_ascii") ) {
            return true;
        } else {
            return false;
        }
     } else{
        return false;
    }
}
```

filterFiles

The `filterFiles` function block is invoked by the collection agent right after the `accept` function has been executed and when the list of files has been created. Files to collect can be removed, but not added or modified. Collection order can be modified by sorting the list.

```
void filterFiles
( list<FileInfo>  fileInfoList  )
```

Parameter:

Parameter	Description
fileInfoList	The *fileInfoList* parameter contains a reference to the list that includes the files to collect. *fileInfoList* is an in/out parameter that serves both as input and output values for the `filterFiles` function.

In the following example, `file1` is not collected if there is another file with the same name that includes the prefix "ignore_" (`ignore_file1`) in the collected directory.

Example - Using filterFiles

In this example, `file1` is not collected if there is another file with the same name that includes the prefix "ignore_" (`ignore_file1`) in the collected directory.

```
void filterFiles(list<FileInfo> fileInfoList) {

    // put the files into a map
    string ignorePrefix = "ignore_";
    map<string, FileInfo> fileInfoMap = mapCreate(string, FileInfo);
    int i = listSize(fileInfoList);
    while(i > 0) {
        i = i - 1;
        FileInfo fileInfo = listGet(fileInfoList, i);
        mapSet(fileInfoMap, fileInfo.name, fileInfo);
    }

    // Remove from the map files that are indicated as files
    // that should be ignored, along with their ignore_ indicator
    // files .

    i = listSize(fileInfoList);
    while(i > 0) {
        i = i - 1;
        FileInfo fileInfo = listGet(fileInfoList, i);

        // check if the filename start with the ignore prefix
        boolean ignore = strStartsWith(fileInfo.name, ignorePrefix);

        if(ignore) {
            string fileToIgnore = strSubstring(fileInfo.name,
            strLength(ignorePrefix), strLength(fileInfo.name));
            mapRemove(fileInfoMap, fileInfo.name);
            mapRemove(fileInfoMap, fileToIgnore);
        }
    }

    // put the remaining files in the fileInfoList
    listClear(fileInfoList);

    list<FileInfo> remainingFiles = mapValues(fileInfoMap);
    i = listSize(remainingFiles);
    while(i > 0) {
        i = i - 1;
        listAdd(fileInfoList, listGet(remainingFiles, i));
    }
}
```

preFileCollection

The `preFileCollection` function block is invoked by the Collection agent right before a file is collected.

```
void preFileCollection
( string  fileName  )
```

Parameter	Description
`fileName`	The *`fileName`* parameter includes the name of the file to be collected.

postFileCollection

The `postFileCollection` function block is invoked by the Collection agent right after a file has been collected.

```
void postFileCollection
( string  fileName  )
```

Parameter	Description
fileName	The *fileName* parameter includes the name of the file that has been collected.

begin

The `begin` function block is invoked by the Collection agent during the Begin Batch processing phase and marks a start point for the file based transaction handling. On severe errors, such as when a transaction fails, the APL command `abort("reason")` should be invoked.

```
void begin
( long  transactionID  )
```

Parameter	Description
transactionID	The *transactionID* parameter is the identifier of this transaction.

commit

The `commit` function block is invoked by the Collection agent right after the End Batch processing phase and marks the end of the file-based transaction handling. On severe errors, such as when a transaction fails, the APL command `abort("reason")` should be invoked.

```
void commit
( long  transactionID ,
boolean  isRecover  )
```

Parameter	Description
transactionID	The *transactionID* parameter is the identifier of this transaction.
isRecover	The *isRecover* parameter is set to "true" if this is a recover operation. If the last commit failed, a commit recover operation is executed upon workflow startup.

rollback

The `rollback` function block is invoked in case of a system failure, right after the End Batch processing phase. On severe errors, such as when a transaction fails, the APL `abort("reason")` command should be invoked.

```
void rollback
( long  transactionID ,
boolean  isRecover  )
```

Parameter	Description
transactionID	The *transactionID* parameter is the identifier of this transaction.
isRecover	The *isRecover* parameter is set to "true" if this is a recover operation. If the last rollback failed, a rollback recover operation is executed upon workflow startup.

5. APL Container Functions

The APL Container functions enable you to manage bundles of data objects according to your specific requirements.

> **Note!**
>
> These functions support only the string data type.

romapCreateTableFromFile

The `romapCreateTableFromFile` function enables you to load a large file to the Execution Context memory, and create a look-up table from it.

The command reads a file and creates a read-only file of key-value data pairs.

```
any romapCreateTableFromFile
 ( string  fileName ,
 char  delimiter ,
 any  oldTable  )
```

Parameter	Description
`fileName`	The name of the (input) file that is about to be converted into a look-up table. **Note!** If the `romapCreateTableFromFile` function reads two different values of the same key, you cannot control nor choose which of the values is registered as the key's value in the table, as every key should be unique.
`delimiter`	The delimiter character that the (input) file includes
`oldTable`	The name of the look-up table that has previously been created with this function for the same *`fileName`* . May be Null. **Note!** This parameter enables you to increase performance by reusing a table that has already been read. It is especially practical when you only want to add a small amount of data.
Returns	A read-only table that is used in subsequent calls to `romapGetValue`

romapGetValue

The `romapGetValue` function enables you to fetch a specific value from a specific table.

```
string romapGetValue
 ( any  tableObject ,
 string  key  )
```

Parameter	Description
`tableObject`	The object that is returned from `romapCreateTableFromFile`. For further information see the section above, romapCreateTableFromFile.
`key`	The key for which the function should return the value
Returns	The value of the specified *`key`*

romapGetSize

The `romapGetSize` function returns the size of a specific table.

```
int romapGetSize( any  tableObject  )
```

Parameter	Description
`tableObject`	The object that is returned from `romapCreateTableFromFile`. For further information see the section above, romapCreateTableFromFile.
Returns	The size of the specified table

6. Audit Functions

For the audit-related functions to take effect, you must select the **Enable Audit** option in the **Audit** tab in the Workflow Properties dialog, see 3.1.8.4 Audit Tab.

> **Note!**
>
> The key or combination of keys is only unique within a batch. If a second batch is executed, a new row looking exactly the same as one from the previous batch might be received. The purpose of the key is to be able to save more than one row per batch.

For further information about Audit, see 8.3 Audit Profile in the Desktop User's Guide.

AuditAdd

The auditAdd function increases or decreases a column value in any table specified. Note that the table must be configured in the Audit Profile Editor window first, and the column must be defined as a 'Counter'.

```
void auditAdd
( string profile,
string tableAlias,
string columnAlias,
any value,
any key1, //Optional
any key2 ) //Optional
```

Parameter	Description
profile	The name of a defined Audit Profile. This declaration is case-sensitive. The *profile* declaration must include the directory name: *myFolder.myProfile*
tableAlias	Name of the table to update. It is case-sensitive and can be in uppercase or lowercase depending on the type of database used. It must also exist in the specified Audit Profile.
columnAlias	Name of the column to update. The name is case-sensitive and can be in uppercase or lowercase depending on the type of database used.
value	The value to increase/decrease the existing value with.
key<n>	The key columns in the order as entered in the Audit Profile Editor.
Returns	Nothing.

AuditSet

The auditSet function sets a column value in any table specified. Note that the table must be configured in the Audit Profile window first, and the column must be defined as being of type 'Value'.

```
void auditSet
( string profile,
string tableAlias,
string columnAlias,
any value,
any key1, //Optional
any key2 ) //Optional
```

Parameter	Description
profile	The name of a defined Audit Profile. This declaration is case-sensitive. The *profile* declaration must include the directory name: *myFolder.myProfile*
tableAlias	Name of the table to update. It is case-sensitive and can be in uppercase or lowercase depending on the type of database used. It must also exist in the specified Audit Profile.
columnAlias	Name of the column to update. The name is case-sensitive and can be in uppercase or lowercase depending on the type of database used.
value	The value to set.
key<n>	The key columns in the order as entered in the Audit Profile.
Returns	nothing.

7. Backlog Size Functions

In a real-time workflow, asynchronous agents use a queue to forward input UDRs to the workflow. This way, a workflow can be configured to include multiple queues, one for every asynchronous agent's output route. Each queue contains a UDR backlog.

> **Note!**
>
> The backlog can only handle UDRs.

queueGetSize

The `queueGetSize` function retrieves the number of UDRs of the queue that currently contains the highest number of UDRs in the workflow.

```
int queueGetSize()
```

Parameter	Description
Returns	the largest queue size.

queueGetLimit

The `queueGetLimit function` gets the configured maximum queue size limit.

```
int queueGetLimit()
```

Parameter	Description
Returns	The soft limit, the configured maximum queue size limit of the workflow queue. You configure the soft limit of the workflow queues in the Workflow Properties. For further information, see 3.1.8 Workflow Propertiesthe Desktop user's guide. **Note!** A real-time agent implementation can modify the soft limit during the agent **initialize()** phase. For further information see 1.4.1 initialize.

8. Base64 Functions

The Based64 functions are used for encoding and decoding of Base64 data.

base64Encoder

This function takes a bytearray and returns a Base64 string representing the data.

```
string base64Encoder ( bytearray data)
```

Parameter	Description
data	The bytearray to encode
Returns	A Base64 encoded string

base64Decoder

This function takes a Base 64 encoded string and returns the decoded data.

```
bytearray base64Decoder ( string data)
```

Parameter	Description
data	The string to decode
Returns	A bytearray containing the decoded data

base64UrlEncoder

This function takes a bytearray and returns a URL safe Base64 string representing the data.

```
string base64UrlEncoder ( bytearray data)
```

Parameter	Description
data	The bytearray to encode
Returns	A URL safe Base64 encoded string

base64UrlDecoder

This function takes a URL safe Base 64 encoded string and returns the decoded data.

```
bytearray base64UrlDecoder ( string data)
```

Parameter	Description
data	The string to decode
Returns	A bytearray containing the decoded data

9. Database Functions

The following sets of functions are used for managing data in databases accessed via Database profiles:

- Database Table functions - Look up and insert data into tables
- Callable Statement functions - Execute stored procedures
- Prepared Statement functions - Efficiently execute statements multiple times and has support for failover
- Database Bulk functions - Bundle multiple queries into one SQL statement

Note!

All database related APL functions use a db connection pool, and they pick a connection from the pool for each invocation. Connection properties, such as the transaction state, are handled internally by each function and are never visible in APL.

These database functions are currently designed to work with the following database data types; *character*, *string*, *integer* and *date* data types such as VARCHAR, INTEGER, NUMBER, DATE and TIMESTAMP.

Other database data types are restricted for APL database functions. The list below contains examples of such types, usually very large data types or custom objects, structures and similar types.

- Large object data types such as BLOB/CLOB
- User-defined structures/objects
- User-defined collections/arrays
- Spatial/Geometry
- JSON, XML

9.1 Database Table Functions

A special table type is used to handle database table lookups with optional caching in memory. All of the following column references may be either numeric (column index, starts at 0) or string (column name).

The table lookup only supports int, string and date types.

Note!

For MS SQL, the column type timestamp is not supported in tables accessed by MediationZone. Use column type datetime instead.

Please refer to the Notes section in the 9. Database Functions page for details on allowed database data type.

sqlExec

The sqlExec function is used when updating and inserting data into tables. It returns an integer value stating how many rows were updated /inserted. SQL errors will cause runtime errors (the workflow aborts).

```
int sqlExec (
 string dbProfile,
 string sqlQuery )
```

Parameter	Description
dbProfile	Name of the database where the table resides, including folder name.
sqlQuery	SQL query to send to the database. Note that SQL statements must not end with ';'. Calls to stored procedures must be embedded in blocks. **Examples - Valid for Oracle databases** "<SQL query>" "BEGIN <procedure>; END;"
Returns	An integer equaling the number of rows updated or inserted.

tableCreate

Returns a table that holds the result of a database query. SQL errors in the table lookup will cause runtime errors (workflow aborts).

```
table tableCreate (
 string dbProfile,
 string sqlQuery
 boolean disableCommit)
```

Parameter	Description
dbProfile	Name of the database where the table resides
sqlQuery	SQL query to send to the database. Note that SQL statements must not end with ';'. Only columns of type number, date and string are supported.
disableCommit	An optional parameter to disable the commit statement from being performed at the end of every SQL transaction for this particular function. Setting this parameter to false will result in the commit statement to be performed at the end of every SQL transaction for this particular function. By default, MediationZone will have the disableCommit set to true unless otherwise changed via this parameter. It should be noted that on recent Oracle versions, the DBLink SQL transaction behaviour has changed, where every single SQL statement for remote database transaction requires a commit or rollback statement in order to close a connection. In addition, PostgreSQL users should set this to *false* as every statement needs to be committed.
Returns	A table

Example - Using tableCreate

To avoid performance problems, the table must be read from the database as seldom as possible. For instance, once for each workflow invocation.

```
initialize {
  table myTab = tableCreate("myFolder.myProfile", "select user from subscribers" );
 }
```

tableCreateIndex

Creates an index for one or several columns of a table. This will greatly increase the efficiency of subsequent tableLookup calls on these columns using the equality operator. If the column already has an index, this function has no effect.

Note!

An index will not be created unless there are at least five rows in the table.

```
void tableCreateIndex (
 table tableValue,
 int|string column1, ... int|string columnN )
```

Parameter	Description
tableValue	A table object
columnN	A column index (starting with 0 for the first column), or name for which to create an index. At least one column must be specified.
Returns	Nothing

Example - Using tableCreateIndex

```
initialize {
  table myTab = tableCreate("myFolder.myProfile", "select id, user from anum" );
  tableCreateIndex( myTab, "id" );
 }
```

tableGet

The tableGet function returns the value of a table entry. The value is returned as an any object, which means the returned value is of the same type as the value extracted.

```
any tableGet (
 table tableValue,
 int row,
 int|string column)
```

Parameter	Description
tableValue	A table object
row	The row index. The first row is indexed 0 (zero).
column	Column index or name. The first column is indexed 0 (zero).
Returns	Any depending on the column type

tableLookup

The tableLookup function returns a table containing all the rows of the original table matching the specified column value(s). At least one pair of (column, operator, value) group must be specified.

```
table tableLookup
( table tableValue,
 int|string column1,
 string operator1,
 string|date|int value1,
 any value1a, //Optional (used only for "between" and "not between") ...
 int|column columnN,
 string operatorN,
 any valueN,
 any valueNa ) //Optional (used only for "between" and "not between")
```

Parameter	Description
tableValue	A table object
columnN	Column index or name. The first column is indexed 0 (zero).
operatorN	Operator specifying the range of the requested values. Possible operators are: • = • != • > • < • <= • >= • between - Requires two operands, (*valueN, valueNa*). • not between - Requires two operands, (*valueN, valueNa*). • starts with - The operator can only be applied to string columns.
valueN/ValueNa	Value(s) matching *columnN* and *operatorN*
Returns	A table matching the query

Example - Using tableLookup

The following example looks for a specific entry in the in-memory table returned by the preceding tableCreate command.

```
table myTab;
// To avoid performance problems, the table is read
// from the database once for each time the
// workflow is activated.
initialize {
 myTab = tableCreate( "myFolder.myProfile", "select user from subscribers" );
}
consume {
  table user = tableLookup( myTab, "user", "=", input.anum );
  if ( tableRowCount( user ) > 0) {
  // At least one entry found.
  } else {
  // No entry found.
  }
 }
```

Note!

When a column is of date type, the matching values must contain date, time, and timezone, as demonstrated in the example below.

Example - Using tableLookup when a column is of date type

```
table myTab;
initialize {
 myTab = tableCreate("myFolder.myProfile","select date_field from all_dates");
}
consume {
 date dateFilter;
 //The matching value must contain date, time, and timezone.
 strToDate(dateFilter,"01/05/2016 00:00:00","dd/MM/yyyy HH:mm:ss","CET");
 table myLookup = tableLookup(myTab,"date_field","=",dateFilter);

 if ( tableRowCount( myLookup ) > 0) {
 // At least one entry found.
 } else {
 // No entry found.
 }
}
```

tableRowCount

The tableRowCount function returns the number of rows in a table.

```
int tableRowCount (table tableValue )
```

Parameter	Description
tableValue	Table object
Returns	An integer stating the number of rows found

9.2 Callable Statements

Callable Statements enable usage of Stored Procedures with output parameters.

The Callable Statement functions are used execute stored procedures and to manage the results. The `DBErrorUDR` is used to handle SQL errors.

Note

Please refer to the Notes section on the 9. Database Functions page for details on allowed database data type.

prepareCall

To prepare a call with an out parameter, the Stored Procedure must be defined with the `prepareCall` function.

```
any CallableStatement.prepareCall
    ( string  dbProfile ,
      string  sqlProc(?,?) ,
      boolean  captureSQLError  (optional) ,
          boolean isFunction (optional) ,
          boolean inclResultParam (optional) )
```

Parameter	Description
dbProfile	Name of the database, including the folder name

sqlProc(?,?)	Name of the stored procedure, including a question mark for each parameters it requires
	
	Example - Using sql_proc
	
	Definition of a Stored Procedure, declared in the database:
	
	`create or replace procedure sql_proc(i int, OUT o char) is`
	`  begin`
	`  . . .`
	`  o:='value for o';`
	`  end;`
	`/`
	
	Note!
	
	The number of question marks must match the number of defined parameters in the Stored Procedure.

captureSQLError	Optional parameter that controls error handling. If the parameter's value is set to true, any SQL error gets captured, without disrupting the execution. For more information about how to fetch the SQL error code and message, see the section below, getError. This parameter is set to `false` by default.
isFunction	Optional parameter that indicates that the call will be made for a stored function. This parameter is set to `false` by default.
inclResultParam	Optional parameter that you can set to true to apply a result parameter of **?=** on the JDBC API stored procedure SQL escape syntax. If the isFunction parameter is set to true, the inclResultParam will be set to true by default.
Returns	Callable Statement Identifier. This object is threadsafe and is used when executing calls towards the stored procedure.

execute

The `execute` function maps to the corresponding JDBC API and could differ slightly depending on the JDBC driver.

The function is used for questions and lookups. If the database should be updated, use the function `executeUpdate` instead.

`execute` can handle several OUT parameters including a table format.

Note!

The table format OUT parameter is applicable only when using a PostgreSQL database.

To execute a call with the parameters expected by the stored procedure, the parameters must be specified in correct order.

```
any CallableStatement.execute
   (any csi,
        any param1, ..., any paramN)
```

Parameter	Description
csi	The Callable Statement Identifier that is returned from the `prepareCall` function
paramN	The values expected by the stored procedure declared in the `prepareCall` function. Parameters registered as out parameters in the stored procedure must be omitted. The parameters must have the same type as defined in the stored procedure.
Returns	The returned value is the Result Identifier of the execution. A new object is returned for every call executed.

executeQuery

The `executeQuery` function maps to the corresponding JDBC API and could differ slightly depending on the JDBC driver.

The function is used for questions and lookups. If the database should be updated, use the function `executeUpdate` instead.

`executeQuery` can handle several OUT parameters except a table format.

To execute a call with the parameters expected by the stored procedure, the parameters must be specified in correct order.

```
any CallableStatement.executeQuery
 (any csi,
 any param1, ..., any paramN)
```

Parameter	Description
csi	The Callable Statement Identifier that is returned from the `prepareCall` function
paramN	The values expected by the stored procedure declared in the `prepareCall` function. Parameters registered as out parameters in the stored procedure must be omitted. The parameters must have the same type as defined in the stored procedure.
Returns	The returned value is the Result Identifier of the execution. A new object is returned for every call executed.

executeUpdate

The `executeUpdate` function maps to the corresponding JDBC API and could differ slightly depending on the JDBC driver.

The function is used for updating the database.

To execute a call with the parameters expected by the stored procedure, the parameters must be specified in correct order.

```
any CallableStatement.executeUpdate
 (any csi,
 any param1, ..., any paramN)
```

Parameter	Description
csi	The Callable Statement Identifier that is returned from the `prepareCall` function
paramN	The values expected by the stored procedure declared in the `prepareCall` function. Parameters registered as out parameters in the stored procedure must be omitted. The parameters must have the same type as defined in the stored procedure.
Returns	The returned value is the Result Identifier of the execution. A new object is returned for every call executed.

get

The `get` function is used to retrieve the result from the executed call.

```
any CallableStatement.get
 (any resultIdentifier,
 int spIndex)
```

Parameter	Description
resultIdentifier	The Result Identifier that is returned from the `executeUpdate` function
spIndex	Index of the requested parameter from the stored procedure (type *int*). The first parameter has index 1.
Returns	The value of the out parameter **Note!** The return value must be type casted.

getUpdateCount

This function returns the number of rows that were affected by the `executeUpdate` function.

```
int CallableStatement.getUpdateCount(any resultIdentifier);
```

Parameter	Description
resultIdentifier	The Result Identifier that is returned from the `executeUpdate` function
Returns	For **Oracle** databases this will return the following statement: The number of rows in the database that were affected by the call. If an update exists -1 will be returned. For **MySQL** and **PostgreSQL** databases this will return the following statement: The number of rows in the database that were affected by the update.

getError

This function will capture potential SQL errors from the `executeUpdate` function and return a UDR that contains both the error code and the error message.

```
DatabaseFunctions.DBErrorUDR = CallableStatement.getError(any resultIdentifier);
```

Parameter	Description
resultIdentifier	The Result Identifier that is returned from the `executeUpdate` function.
Returns	Returns an error UDR. For more information about the error UDR, see the section below, DBErrorUDR. If no error occurred, null will be returned.

Example - Handle error raised by Stored Procedure

Stored Procedure definition:

```
create or replace procedure upd_item(id int, amount int) is
  begin
    if amount > 110000 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Amount is to high!');
    end if;
    ...
  end;
/
APL Code:
any s;
DatabaseFunctions.DBErrorUDR error;
initialize {
  s = CallableStatement.prepareCall("p.db","UPD_ITEM(?,?)",true);
}
consume {
  ...
  any result = CallableStatement.executeUpdate(s, id, amount);
  error = CallableStatement.getError(result);
  if (error != null) {
     //handle error
     if (error.ErrorCode == -20001) {
        udrRoute(input, "adjust_amount")
     } else {
        abort(error.ErrorCode + error.ErrorMessage);
     }
  } else {
     //no error -let's proceed
     int cnt = CallableStatement.getUpdateCount(result);
     ...
  }
}
```

DBErrorUDR

If the `executeUpdate` function generates an SQL error, the `getError` function will generate a `DBErrorUDR`.

The following fields are included in the `DBErrorUDR`:

Field	Description
`ErrorCode (int)`	The SQL error code.
`ErrorMessage (string)`	The SQL error message.

9.3 Prepared Statements

A prepared statement is used to efficiently execute a statement multiple times. The Prepared Statement related functions support database failover, which is useful in a replicated database environment, by receiving multiple database profiles.

If a workflow failure occurs using one database profile, a failover process starts to replace that profile with the next available one. If none of the supplied profiles could be used a null value will be returned.

Note!

Failovers can be triggered by all errors that occur while connecting to the database or executing commands, not exclusively due to connection problems.

Please refer to Notes section on the 9. Database Functions page for details on allowed database data type.

sqlPrepSelect

The `sqlPrepSelect` function is used when selecting data from database tables.

```
table sqlPrepSelect
    (string sqlStatement,
     any parameter [,...],
     string DBProfile [,...],
         boolean disableCommit,  //Optional)
```

Parameter	Description
sqlStatement	SQL string containing one or more parameter placeholders ('?').
parameter(s)	Value(s) to bind against parameter placeholders.
DBProfile/s)	Name of the database profile(s) to use for access.
disableCommit	An optional parameter to disable the commit statement from being performed at the end of every SQL transaction for this particular function. Setting this parameter to false will result in the commit statement to be performed at the end of every SQL transaction for this particular function. By default, MediationZone will have the disableCommit set to true unless changed by this parameter. It should be noted that on recent Oracle versions, the DBLink SQL transaction behaviour has changed, where every single SQL statement for remote database transaction requires a commit or rollback statement in order to close a connection. In addition, PostgreSQL users should set this to *false* as every statement needs to be committed.
Returns	Null in case of failure, otherwise a table, holding the SQL select result.

Example - Using sqlPrepSelect

```
final string sql = "select col2 from tab1 where col1=?";
consume {
    table tab = sqlPrepSelect(sql, myUdr.aValue,
    "Default.myPrimaryDB", "Default.mySecondaryDB");

    if ( tab != null ) {
       if ( tableRowCount(tab) > 0 ) {
               debug("Found following value in col2:"
               + tableGet(tab, 0, 0));
       }
    }
    else {
      debug("Both primary and secondary database nodes
             are unavailable");
    }
}
```

sqlPrepDynamicSelect

The `sqlPrepDynamicSelect` function is used when selecting data from database tables with dynamically created SQL statements.

```
table sqlPrepDynamicSelect
    (string sqlStatement,
     list<any> parameters,
     string DBProfile [,...],
         boolean disableCommit,  //Optional)
```

Parameter	Description
sqlStatement	SQL string containing one or more parameter placeholders ('?'). The string may also contain variables that substitute any part of the statement such as table- or column names. **Note!** Validation of sqlStatement is performed in runtime and not during configuration. Make sure that number of placeholders matches the actual number of parameters.
parameter(s)	Value(s) to bind against parameter placeholders
DBProfile/s)	Name of the database profile(s) to use for access
disableCommit	An optional parameter to disable the commit statement from being performed at the end of every SQL transaction for this particular function. Setting this parameter to false will result in the commit statement to be performed at the end of every SQL transaction for this particular function. By default, MediationZone will have the disableCommit set to true unless otherwise changed via this parameter. It should be noted that on recent Oracle versions, the DBLink SQL transaction behaviour has changed, where every single SQL statement for remote database transaction requires a commit or rollback statement in order to close a connection. In addition, PostgreSQL users should set this to *false* as every statement needs to be committed.
Returns	Null in case of failure, otherwise a table, holding the SQL select result

Example - Using sqpPrepDynamicSelect

```
final string EXT_REF_CONFIG = "Default.extrefprop";
string TABLE_KEY = externalReferenceGet(EXT_REF_CONFIG, "mytable");
consume {
        string sql = "select col2 from " + TABLE_KEY + " where col1=?";
        list <any> parameters = listCreate(any, myUDR.aValue);
        table tab = sqlPrepDynamicSelect(sql, parameters, "Default.myPrimaryDB", "Default.
mySecondaryDB");
        if ( tab != null ) {
           if ( tableRowCount(tab) > 0 ) {
              debug("Found following value in col2:" + tableGet(tab, 0, 0));
           }
        }
        else {
           abort("Both primary and secondary database nodes are unavailable");
         }
}
```

sqlPrepUpdate

The `sqlPrepUpdate` function is used when updating or inserting data into a database table.

```
int sqlPrepUpdate(
    string sqlStatement,
    any parameter [,...],
    string DBProfile [,...],
    )
```

Parameter	Description
sqlStatement	SQL string containing one or more parameter placeholders ('?')
parameter(s)	Value(s) to bind against parameter placeholders
DBProfile/s)	Name of the database profile(s) to use for access
Returns	-1 in case of failure, otherwise an integer equal to the number of rows updated or inserted

Example - Using sqpPrepUpdate

```
final string sql = "update table1 set name = ? where id = ?";

consume {
   int cnt = sqlPrepUpdate(sql, aNewName, anIdent,
    "Default.myPrimaryDB", "Default.mySecondaryDB");

   if ( cnt == -1 ) {
       debug("Both primary and secondary database nodes
    are unavailable");
   }
}
```

9.4 Database Bulk Functions

The Database Bulk functions enable you to bundle multiple queries into one SQL statement. This significantly improves the throughput due to reduced need of context switching and network traversals. The functions are intended for batch usage only.

Making an SQL Query

When making an SQL query it must follow a certain format and the order of the columns in the query must remain the same in the statement as in the `where_clause`.

Example - Statement example

Select A,B,C from Test_Table

The *`where_clause`* argument for the `sqlBulkPrepareStatement`. For further information, see the section below, sqlBulkPrepareStatement.

`A=? and B=? and C=?`

The function adds a "WHERE" to the end of the statement and an "AND" between the `static_where_clause` and the `where_clause` statements. An "OR" is added between each `where_clause` statement.

The lookup SQL query next shows one `static_where_clause` and two `where_clause`.

```
Select A,B,C  FROM Test_Table WHERE (A=? and B=? and C=?)
AND ((A=? and B=? and C=?) OR (A=? and B=? and C=?))
```

If the `where_clause` in the SQL statement contains a range condition instead of an exact lookup key, the intended result cannot directly be associated back to the UDRs. Instead a result based on the significant columns will be returned. The result rows matching the range condition must be associated back to the UDRs using APL code. For further information, see the lookup query example below.

sqlBulkPrepareStatement

This function prepares the the query. A JDBC connection is created and the `bulkSize` is set.

The returned object is a mandatory parameter to the `sqlBulkExecuteQuery()` function and is needed for the execution of the lookup SQL query.

```
any sqlBulkPrepareStatement
(string  dbProfile ,
string  statement ,
string  static_where_clause ,  //set to null if not present
string  where_clause ,
int  bulkSize ,
int  significantColumns ,
int  totalColumns ,
int  timeout ,  //Optional
boolean disableCommit ,  //Optional)
```

Parameter	Description
dbProfile	Name of the database where the table resides
statement	Statement to send to the database
static_where_clause	The static_where_clause of the SQL statement stays the same throughout the batch mode execution
where_clause	The where_clause of the SQL statement principal. The where_clause columns that are to be considered significant must correspond with the columns in the select statement. For example, when statement reads "select A, B and C from X", where A and B is significant, the where_clause must begin with "A=? and B=?".
bulkSize	The size of the bundled lookup SQL query, that is the number of UDRs with a unique UDR key to be bundled. The ideal *bulkSize* depends on database table characteristics such as structure and number of rows.
significantColumns	This parameter indicates which columns in the select statement are significant to tie a result row to its context (UDR). All columns to be used as significant columns must be included in SQL search condition, and they must be used in the same order as in the search condition. For example, when using an incoming UDR as context and a search condition matching its IMSI field, using significant columns = 1 ties all matching rows to this UDR, namely, all rows with the same IMSI as that carried by the UDR.
totalColumns	The total number of columns in the statement
timeout	The least time interval, in milliseconds, the workflow will allow before a database bulk lookup will be done
disableCommit	An optional parameter to disable the commit statement from being performed at the end of every SQL transaction for this particular function. Setting this parameter to false will result in the commit statement to be performed at the end of every SQL transaction for this particular function. By default, MediationZone will have the disableCommit set to true unless otherwise changed via this parameter. It should be noted that on recent Oracle versions, the DBLink SQL transaction behaviour has changed, where every single SQL statement for remote database transaction requires a commit or rollback statement in order to close a connection. In addition, PostgreSQL users should set this to *false* as every statement needs to be committed.
Returns:	The returned object is a “Prepared Statement” object containing the added parameters, a prepared result list and a maintained contexts list. This object is a mandatory parameter to the *sqlBulkExecuteQuery()* function and is needed for the execution of the SQL query.

Example - Lookup query

Lookup query with one of the `where_clause` conditions without an exact lookup key.

To prepare a query:

```
SELECT Column1, Column2, Column3 FROM Test_Table
WHERE Column1=? AND Column2=? AND Column3 <=?;
```

The call to the function and the values for the above example:

```
significantColumns = 2
totalColumns = 3
where_clause = "Column1=? and Column2=? and Column3 <=?"
static_where_clause = null
bulksize = 100
dbprofile = "myFolder.myProfile"
statement = "select Column1, Column2, Column3 from Test_Table"
```

The call to the APL function:

```
initialize {
    any pStatement = sqlBulkPrepareStatement("myFolder.myProfile",
    "select Column1, Column2, Column3 from Test_Table",
    null, "Column1=? and Column2=? and Column3 <=?",
    100, 2,3);
}
```

sqlBulkExecuteQuery

When a UDR enters `sqlBulkExecuteQuery` the first met criteria of `bulkSize` or `timeout` triggers the database bulk lookup.

The start of the timeout interval will be reset when a database query is done regardless if the last query was due to reached `bulkSize` or `timeout`.

Next timeout occasion is calculated by adding the time of the last database lookup to the current *`timeout`* value.

```
list <any> sqlBulkExecuteQuery(any ps,
any  context ,
any value1, any value2, ...)
```

Parameter	Description
`ps`	The object returned from a call to the `sqlBulkPrepareStatement()`
`context`	A context string, usually the UDR routed to the node
`values`	A number of values to populate the query
Returns:	A list containing objects that in their turn contains a context object and a result list. This object is a mandatory parameter to the *`sqlBulkResult*`* functions.

Example - Using sqlBulkExecuteQuery

The call to the function and the values for the example above, Example - Lookup Query:

```
ps =  pStatement object from sqlBulkPrepareStatement()
context = myUDR
values =  myValues1, myValue2, myValue3
```

Call to the APL function:

```
consume{
    list <any> result = sqlBulkExecuteQuery(pStatement,
    myUDR,
    10,2,3);
}
```

The function executes the SQL query before the `bulkSize` is reached.

```
list <any> sqlBulkDrain(any ps)
```

Parameter	Description
ps	A `preparedStatement` object returned by the `sqlBulkPrepareStatement()`, containing among others a result list and a maintained contexts list.
Returns:	A list containing objects that in their turn contain a context object and a list with results.

sqlBulkResultGetContext

The function finds the context object from one result object in the list returned either by the `sqlBulkExecuteQuery()` or the `sqlBulkDrain()` functions.

```
any sqlBulkResultGetContext(any presult)
```

Parameter	Description
presult	A result object from the list returned by the `sqlBulkExecuteQuery()` or the `sqlBulkDrain()` functions
Returns:	The context object associated with the `presult` parameter

sqlBulkResultGetSize

The function states the size of a result object in the list returned from the `sqlBulkExecuteQuery()` or the `sqlBulkDrain()` functions.

```
int sqlBulkResultGetSize(any presult)
```

Parameter	Description
presult	A result object from the list returned by the `sqlBulkExecuteQuery()` or the `sqlBulkDrain()` functions
Returns:	An integer representing the size of the result

sqlBulkResultGetValue

The function gets a result value from a result object in the list returned by either the `sqlBulkExecuteQuery()` or the `sqlBulkDrain()` functions.

```
any sqlBulkResultGetValue(any presult,
    int index,
    int column)
```

Parameter	Description
presult	A result object from the list returned by the `sqlBulkExecuteQuery()` or the `sqlBulkDrain()` functions
index	The index of the result table
column	The column number
Returns:	The value of the result object

sqlBulkElapsedNanoTime

This function accumulates the time of the execution of the database queries that is end time of execution minus start time of execution.

```
long sqlBulkElapsedNanoTime()
```

Parameter	Description
Returns:	A long integer representing the accumulated nanoseconds.

sqlBulkClose

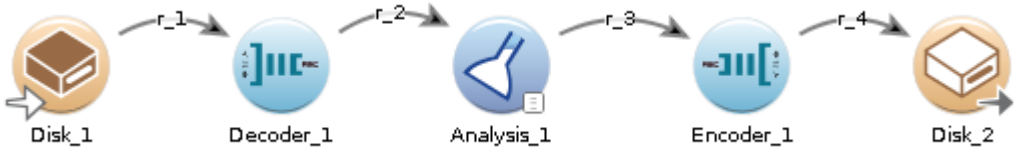
The function closes down the database connection as well as the SQL statement. If the `sqlBulkClose` is not called the connection will be closed when the workflow stops.

```
void sqlBulkClose(any  ps )
```

Parameter	Description
ps	A preparedStatement object returned by the `sqlBulkPrepareStatement()`, containing among others a result list and a maintained contexts list
Returns:	Nothing

Example Workflow

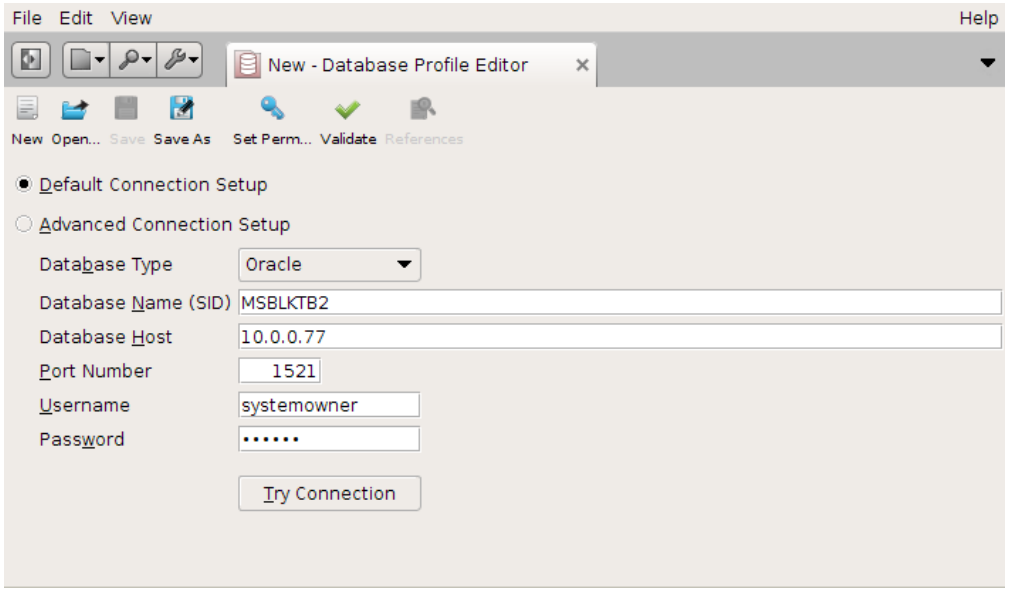
The example shows a basic workflow where a Database Bulk lookup with a range condition is performed.



Example workflow

Prerequisites

To use the sqlBulk related APL commands there must be a database profile configuration prepared. For further information, see 8.5 Database Profile in the Desktop user's guide.



Database profile example

Decoder_1 Agent

The incoming UDRs contains name and number to 10 people.

```
Katerine,Stenberg,0046123456
Kalle,Andersson,0046789123
Mia,Karlsson,0046999111
Karina,Hansson,0046222333
PerErik,Larsson,0046111999
Mikael,Grenkvist,0046333444
Jonas,Bergsten,0046555666
Petra,Sjoberg,0046777888
Karl,Kvistgren,0046444555
FiaLotta,Bergman,0046666222
```

Analysis_1 Agent

The Analysis agent does the database bulk lookup. In this case, the number of calls made by the people listed in the incoming UDRs.

When the bulksize is reached (in this case 10) the query to the database selected in the database profile is made.

```
// A preparedStatement object
any pStatementObj;

// Default.ExampleTable - the name of the database profile

initialize {
   pStatementObj = sqlBulkPrepareStatement(
   "Default.ExampleTable", "SELECT first_name, last_name,
   telephone_no, usage_min FROM EXAMPLE_TABLE",
   null, "first_name=? and last_name=? and telephone_no=? ", 10, 3, 4);
}

consume {
   debug("In consume adding queries to the bulk execute");
   processResults( sqlBulkExecuteQuery(
   pStatementObj, input, input.first_name, input.last_name,
   input.telephoneno));
}

drain {
   processResults( sqlBulkDrain (pStatementObj));
}

void processResults( list <any> resultList ) {
   if( resultList == null ) {

      // Do nothing

      return;
    }

   int i=0;

   // Iterate the results returned from the database

   while( i < listSize( resultList )) {
      any result = listGet( resultList, i );
      i=i+1;

      // A string to populate with results from the database query

      string myTotalValue;
      int resultSize = sqlBulkResultGetSize(result);

      // At end of the iteration the myTotalValue will contain a
      // comma separated usageMin value string with values meeting

      int index = 0;

      while(index < resultSize){
         // Populate myTotalValue with result if it met range condition

         int usageMinValue = (int) sqlBulkResultGetValue(result, index, 3);

         // In this test the range condition is set to a constant value.

         if( usageMinValue >= 10 ){
             myTotalValue = myTotalValue+","+usageMinValue;
         }

         index = index +1;
      }

      InternalUDR iUdr = (InternalUDR) sqlBulkResultGetContext( result );

      // Assign the populated array to the usageMin

      iUdr.usagemin = myTotalValue;
      udrRoute( iUdr );
   }
}
```

The database lookups are in this example done in a database table with a content shown here:

Id	FIRST_NAME	LAST_NAME	TELEPHONE_NO	USAGE_MIN
1	Katerine	Stenberg	46123456	10
2	Katerine	Andersson	46789123	10
3	Mia	Karlsson	46999111	10
4	Karina	Hansson	46222333	10
5	PerErik	Larsson	46111999	11
6	Mikael	Grenkvist	46333444	10
7	Jonas	Bergsten	46555666	10
8	Petra	Sjoberg	46777888	10
9	Karl	Kvistgren	46444555	10
10	FiaLotta	Bergman	46666222	10
11	Katerine	Stenberg	46123456	10
12	Katerine	Andersson	46789123	9
13	Mia	Karlsson	46999111	8
14	Karina	Hansson	46222333	7
15	PerErik	Larsson	46111999	11
16	Mikael	Grenkvist	46333444	5
17	Jonas	Bergsten	46555666	4
18	Petra	Sjoberg	46777888	3
19	Karl	Kvistgren	46444555	2
20	FiaLotta	Bergman	46666222	1

Database Table

Encoder_1 Agent

When the encoder has encoded the information from the analysis agent the result sent to the disk out agent is shown here:

```
Katerine,Stenberg,46123456,,10,10
Kalle,Andersson,46789123,,10
Mia,Karlsson,46999111,,10
Karina,Hansson,46222333,,10
PerErik,Larsson,46111999,,11,11
Mikael,Grenkvist,46333444,,10
Jonas,Bergsten,46555666,,10
Petra,Sjoberg,46777888,,10
Karl,Kvistgren,46444555,,10
FiaLotta,Bergman,46666222,,10
```

10. DNS Lookup Function

lookUpDNS

Sends a lookup query to the stated host and returns the result

```
drudr lookUpDNS
        ( string queryType,
          string queryString,
          string host,
          int port,
          int timeout,
          int retries )
```

Parameter	Description
queryType	The query type to be used, e g NAPTR, AAA, CNAME, etc. For full list, see http://www.dnsjava.org/doc/org/xbill/DNS/Type.html.
queryString	The string to be used in the query.
host	The host that should be used for the lookup.
port	The port that should be used for the lookup.
timeout	The number of milliseconds to wait before timeout.
retries	The number of retries you want to make before returning an error code and proceeding.
Returns	A `DNSResponseUDR` containing the lookup result (`list<DNSRecordUDRs>`), status code (`int`) and error message (`string`)

11. Decrypt Functions

decryptMZEncryptedString

Decrypts a string that has been encrypted using `mzsh encryptpassword`, and returns the decrypted string.

```
decryptMZEncryptedString (string encryptedString)
```

Parameter	Description
`encryptedString`	The string encrypted using `mzsh encryptpassword`
Returns	The decrypted string

12. Diameter Functions

The functions described below are used for setting the value of the Diameter AVP Origin-State-Id. The Diameter Stack agent has a corresponding MIM value named Origin State Id MIM value. For further information about the MIM values, see 9.16.3.2 Diameter Stack Agent Input/Output Data and MIM in the Desktop User's Guide.

diameterSetOriginStateId

The diameterSetOriginStateId function sets the value for the Origin State Id. This can be called in the initialize block to replace the default initialization value.

```
diameterSetOriginStateId
( string DiameterStackName,
  int newValue )
```

Parameter	Description
DiameterStackName	The name of the Diameter Stack agent for which you want to set the Origin State Id.
newValue	The new value that you want to set instead of the default initialization value.

diameterIncrementOriginStateId

The diameterIncrementOriginStateId function increments the value for the Origin State Id.

```
diameterIncrementOriginStateId
( string DiameterStackName )
```

Parameter	Description
DiameterStackName	The name of the Diameter Stack agent for which you want the Origin State Id to be incremented.

diameterGetOriginStateId

The diameterGetOriginStateID function gives the same value as accessing the MIM value.

```
diameterGetOriginStateId
( string DiameterStackName )
```

Parameter	Description
DiameterStackName	The name of the Diameter Stack agent for which you want to get the Origin State Id.

13. Distributed Storage Functions

The Distributed Storage Profile enables you to access a distributed storage solution from APL without having to provide details about its type or implementation.

The APL functions described in this section use a key-value model to read, store and remove data via the specified profile. The functions can optionally use sessions that are either committed or rolled back in order to secure transaction consistency.

Initializing Storage

dsInitStorage

The `dsInitStorage` function returns a Distributed Storage Profile object that is used in subsequent APL calls to distributed storage functions.

```
any dsInitStorage(string profid)
```

Parameter	Description
profid	A string containing a reference to a Distributed Storage Profile
Returns	A Distributed Storage Profile object

Example - Using dsInitStorage

```
any storage=null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

Note!

The `dsInitStorage` function must be used in the initialize block and not consume block.

Storing and Removing Data

The following functions enable you to perform changes on data in a distributed storage:

- `dsStore`
- `dsStoreUdr`
- `dsStoreMany`
- `dsStoreManyUdrs`
- `dsRemove`
- `dsRemoveMany`

dsStore

The `dsStore` function stores a key-value pair in a distributed storage.

```
void dsStore(
 any profid,
 string key,
 bytearray value,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
key	The key of a key-value pair
value	The value of a key-value pair
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to store data in non-transaction mode. If the identifier is set the transaction is completed by calling dsCommitTransaction.
Returns	Nothing

Example - Using dsStore

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    bytearray myValue = null;
    strToBA(myValue, "123");
    dsStore(storage, "mykey", myValue, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsStoreUdr

The `dsStoreUdr` function stores a key-value pair in a distributed storage.

```
void dsStoreUdr(
 any profid,
 string key,
 drudr myUDR,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
key	The key of a key-value pair
myUDR	The UDR to be stored
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to store data in non-transaction mode. If the identifier is set the transaction is completed by calling dsCommitTransaction.
Returns	Nothing

Example - Using dsStoreUDR

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    default.dataUDR udr = udrCreate(default.myultra.dataUDR);
    dsStoreUdr(storage, "mykey", udr, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
 }
}
```

dsStoreMany

The `dsStoreMany` function stores a set of key-value pairs in a distributed storage.

```
void dsStoreMany(
 any profid,
 map<string, bytearray> values,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
values	The key-value pairs to store
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to store data in non-transaction mode. If the identifier is set the transaction is completed by calling dsCommitTransaction.
Returns	Nothing

Example - Using dsStoreMany

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    map<string, bytearray> myValues =
        mapCreate(string, bytearray);
    bytearray value1;
    bytearray value2;
    strToBA(value1, "abc");
    mapSet(myValues,"first", value1);
    strToBA(value2, "def");
    mapSet(myValues,"second", value2);
    dsStoreMany(storage, myValues, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsStoreManyUdrs

The `dsStoreManyUdrs` function stores a set of key-value pairs in a distributed storage.

```
void dsStoreManyUdrs(
 any profid,
 map<string, drudr> myUDRs,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
myUDRs	The key-value pairs to store
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to store data in non-transaction mode. If the identifier is set the transaction is completed by calling dsCommitTransaction.
Returns	Nothing

Example - Using dsStoreManyUDRs

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    map<string, drudr> myUDRs = mapCreate(string, drudr);
    default.dataUDR udr1 = udrCreate(default.myultra.dataUDR);
    mapSet(myUDRs,"first", udr1);
    default.dataUDR udr2 = udrCreate(default.myultra.dataUDR);
    mapSet(myUDRs,"second", udr2);
    dsStoreManyUdrs(storage, myUDRs, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsRemove

The `dsRemove` function removes a key-value pair, identified by a key, from a distributed storage.

```
void dsRemove(
 any profid,
 string key,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
key	The key of the key-value pair to remove
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to remove data in non-transaction mode. If the identifier is set the transaction is completed by calling dsCommitTransaction.
Returns	Nothing

Example - Using dsRemove

```
any storage = null;
initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    //It is assumed here that "mykey" is already stored.
    dsRemove(storage, "mykey", null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsRemoveMany

The `dsRemoveMany` function removes a list of key-value pairs, identified by a list of keys, from a distributed storage.

```
void dsRemoveMany(
 any profid,
 list<string> keys,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
keys	The keys of the key-value pairs to remove
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to remove data in non-transaction mode. If the identifier is set the transaction is completed by calling dsCommitTransaction.
Returns	Nothing

Example - Using dsRemoveMany

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    list<string> myKeys = listCreate(string);
    int i = 0;
    // It is assumed here that "KEY0", "KEY1".."KEY9"
    // are already stored.
    while (i < 10) {
        listAdd(myKeys, "KEY" + (i));
        i = i + 1;
    }
    dsRemoveMany(storage, myKeys, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

Reading Data

The following functions enable you to read data from a distributed storage.

- `dsRead`
- `dsReadUdr`
- `dsReadMany`
- `dsReadManyUdrs`

dsRead

The `dsRead` function reads a value, identified by a key, from a distributed storage.

```
bytearray dsRead(
 any profid,
 string key,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
key	The key of a key-value pair
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to read data in non-transaction mode. If the identifier is set, the read key-value pair is locked for other transactions untildsCommitTransaction or dsRollbackTransaction is called.
Returns	A bytearray containing the value of a key-value pair

Example - Using dsRead

```
any storage = null;
initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
consume {
    //It is assumed here that "mykey" is already stored.
    bytearray myValue = dsRead(storage, "mykey", null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsReadUdr

The `dsReadUdr` function reads a set of UDRs, identified by a list of keys, from a distributed storage.

```
drudr dsReadUdr(
 any profid,
 string key,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
key	The key of a key-value pair
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to read data in non-transaction mode. If the identifier is set, the read key-value pair is locked for other transactions untildsCommitTransaction or dsRollbackTransaction is called.
Returns	A UDR identified by the key parameter

Example - Using dsReadUDR

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    //It is assumed here that "mykey" is already stored.
    default.myultra.dataUDR udr =
        (default.myultra.dataUDR) dsReadUdr(storage, "mykey",
        null);
    if(null != dsLastErrorMessage()) {
       //Error Handling
    }
}
```

dsReadMany

The dsReadMany function reads a set of key-value pairs, identified by a list of keys, from a distributed storage.

```
map<string, bytearray> dsReadMany(
 any profid,
 list<string> keys,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
keys	The keys of the key-value pairs to be read
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to read data in non-transaction mode. If the identifier is set, the read key-value pairs are locked for other transactions untildsCommitTransaction or dsRollbackTransaction is called.
Returns	A map containing the key-value pairs

Example - Using dsReadMany

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    list<string> myKeys = listCreate(string);
    int i = 0;
    // It is assumed here that "KEY0", "KEY1".."KEY9"
    // are already stored.
    while (i < 10) {
        listAdd(myKeys, "KEY" + (i));
        i = i + 1;
    }
    map<string, bytearray> myMap = mapCreate(string, bytearray);
    myMap = dsReadMany(storage, myKeys, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsReadManyUdrs

The `dsReadManyUdrs` function reads a key-value map, identified by a key, from a distributed storage.

```
map<string, drudr> dsReadManyUdrs(
 any profid,
 list< string keys,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
keys	The keys of the key-value pairs to be read
transid	The transaction identifier that is returned by dsBeginTransaction. Use a null value to read data in non-transaction mode. If the identifier is set, the read key-value pairs are locked for other transactions untildsCommitTransaction or dsRollbackTransaction is called.
Returns	A map containing the key-value pairs

Example - Using dsReadManyUDRS

```
any storage = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    list<string> myKeys = listCreate(string);
    int i = 0;
    // It is assumed here that "KEY0", "KEY1".."KEY9"
    // are already stored.

    while (i < 10) {
        listAdd(myKeys, "KEY" + (i));
        i = i + 1;
    }
    map<string, default.myultra.dataUDR> myUdrMap =
        mapCreate(string, default.myultra.dataUDR);
    myUdrMap = dsReadManyUdrs(storage, myKeys, null);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

Transaction Functions

The following functions provide transaction safety by performing commit or rollback on a set of calls to a Distributed Storage Profile:

- `dsBeginTransaction`
- `dsCommitTransaction`
- `dsRollbackTransaction`

dsBeginTransaction

The `dsBeginTransaction` function begins a new transaction and returns an object that can be used in subsequent APL calls to distributed storage functions.

```
any dsBeginTransaction(any profid)
```

Parameter	Description
profid	The Distributed Storage profile
Returns	A transaction identifier

dsCommitTransaction

The `dsCommitTransaction` function ends a transaction and commits changes that have been made using the specified transaction identifier.

```
void dsCommitTransaction(
 any profid,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
transid	The transaction identifier that is returned by dsBeginTransaction
Returns	Nothing

Example - Using dsCommitTransaction

```
any storage = null;
any transid = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    transid = dsBeginTransaction(storage);
    bytearray myValue = null;
    strToBA(myValue, "123");
    dsStore(storage, "mykey1", myValue, transid);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
    dsStore(storage, "mykey2", myValue, transid);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
    dsCommitTransaction(storage, transid);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsRollbackTransaction

The `dsRollbackTransaction` function ends a transaction and reverts changes that have been made using the specified transaction identifier.

```
void dsRollbackTransaction(
 any profid,
 any transid)
```

Parameter	Description
profid	The Distributed Storage profile
transid	The transaction identifier that is returned by dsBeginTransaction
Returns	Nothing

Example - Using dsRollbackTransaction

```
any storage = null;
any transid = null;

initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}

consume {
    transid = dsBeginTransaction(storage);
    bytearray myValue = null;
    strToBA(myValue, "123");
    dsStore(storage, "mykey1", myValue, transid);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
    dsStore(storage, "mykey2", myValue, transid);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
    dsRollbackTransaction(storage, transid); //nothing is stored
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

Iterator Functions

The following functions are used for traversing through a set of keys in a distributed storage:

- `dsCreateKeyIterator`
- `dsCreateREKeyIterator`
- `dsGetNextKey`
- `dsDestroyKeyIterator`

dsCreateKeyIterator

Note!

This iterator function is only valid for Couchbase.

The `dsCreateKeyIterator` function creates an iterator object that is used in subsequent APL calls to other iterator functions.

```
any dsCreateKeyIterator(
 any profid,
 string startkey,
 string stopkey)
```

Parameter	Description
profid	The Distributed Storage profile
startkey	The key to start iterating at
stopkey	The key to finish iterating at
Returns	An iterator object used in subsequent APL calls to dsGetNextKey and dsDestroyKeyIterator.7.5.2. dsGetNextKey

dsCreateREKeyIterator

> **Note!**
>
> This iterator function is only valid for Redis.

The `dsCreateREKeyIterator` function creates an iterator object that is used in subsequent APL calls to other iterator functions.

```
any dsCreateREKeyIterator(
 any profid,
 string searchpattern)
```

Parameter	Description
profid	The Distributed Storage profile
searchpattern	A search pattern which may include wild card '*'
Returns	An iterator object is used in subsequent APL calls to dsGetNextKey and dsDestroyKeyIterator.

dsGetNextKey

The `dsGetNextKey` function gets the next available key from a distributed storage using an iterator object. Each subsequent call to the function returns the next key in the iterator's range.

```
string dsGetNextKey(
 any profid,
 any iterator)
```

Parameter	Description
profid	The Distributed Storage Profile.
iterator	The iterator object returned by dsCreateKeyIterator.
Returns	The key of a key-value pair. A null value is returned if a key cannot be found.

Example - Using dsCreateKeyIterator

```
any storage = null;
initialize {
    storage = dsInitStorage("default.ds_profile");
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
consume {
    string startKey = "mystartkey";
    string stopKey = "mystopkey";
    any iterator =
        dsCreateKeyIterator(storage, startKey, stopKey);
    string key = dsGetNextKey(storage, iterator);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
    // It is assumed here that "mystartkey" and "mystopkey"
    // are already stored
    while ( null != key) {
        debug(dsRead(storage, key, null));
        if(null != dsLastErrorMessage()) {
            //Error Handling
        }
        key = dsGetNextKey(storage, iterator);
        if(null != dsLastErrorMessage()) {
            //Error Handling
        }
    }
    dsDestroyKeyIterator(storage, iterator);
    if(null != dsLastErrorMessage()) {
        //Error Handling
    }
}
```

dsDestroyKeyIterator

The `dsDestroyKeyIterator` function destroys an iterator object created with dsCreateKeyIterator in order to free up memory. It is good practice to always include a call to this function in the APL code though it may not be required for all types of storage.

```
void dsDestroyKeyIterator(
 any profid,
 any iterator)
```

Parameter	Description
profid	The Distributed Storage profile
iterator	The iterator object returned by dsCreateKeyIterator
Returns	Nothing

Error Handling

dsLastErrorMessage

The `dsLastErrorMessage` function returns the error message from the last call to a distributed storage function. Distributed storage functions that are not used for error handling, do not return error codes and generally do not cause runtime errors. For this reason, The dsLastErrorMessage function must be called after each operation to validate success.

```
string dsLastErrorMessage()
```

Parameter	Description
Returns	An error message, or null if no error has occurred since the last call to this function. The content of the message depends on the type of profile that is used with the Distributed Storage profile. For example, with a Couchbase profile, the error message is identical to the error message exposed by the Couchbase API.

14. Dynamic Functions

Dynamic functions enable you to dynamically call an agent, a plug-in, or a generic APL function by name.

dynamicCall

The `dynamicCall` function calls a user defined APL function.

```
any dynamicCall ( any  handler ,
                  string  functionName ,
                  list<any>  arguments  )
```

Parameter	Description
handler	The handler that is returned by `dynamicImport`, `dynamicPlugin` or `dynamicCompile`. If the handler is set to `null`, the functions that are defined in the agent are used.
functionName	The name of the function to call
arguments	A list of arguments to pass to the called function. To call a function that requires no argument, use either `null` or an empty list.
Returns	Either the result of the executed function,or `null`, if the function returns void

Example - Using dynamicCall to call an agent defined function

```
consume {
    //
    // Dynamic Call to call an agent defined function
    //
    list<any> args = listCreate(any);
    dynamicCall(null, "testA", args);

    listAdd(args, "HelloWorld");
    dynamicCall(null, "testB", args);

    listAdd(args, 21);
    int result = (int) dynamicCall(null, "testC", args);
    debug(result);
}

void testA() {
    debug("testA called");
}

void testB(string b) {
    debug("testB("+b+") called");
}

int testC(string b, int c) {
    debug("testC("+b+", "+c+") called");
    return c*2;
}
```

Example - Using dynamicCall to call a plugin function

```
any plugin;

initialize {
    //
    // Initialize DTK plugin - Using ClassName
    //
    plugin = dynamicPlugin(
                    "com.<product>.apl.MyPluginExecutor");
}

consume {
    //
    // Dynamic Call to call a plugin function
    //
    list<any> arguments;
    listAdd(arguments, 1024);
    dynamicCall(plugin,
                "myPluginExposedFunction",
                arguments);
}
```

Example - Using dynamicCall to call an APL Code defined function

```
any aplCode;

initialize {
    //
    // Dynamic Import of generic APL Code
    //
    aplCode = dynamicImport("Default.helloworld");
}

consume {
    //
    // Dynamic Call to call an APL Code defined function
    //
    list<any> arguments;
    listAdd(arguments, "HelloWorld");
    dynamicCall(aplCode,
                "myHelloFunc",
                arguments);


}
```

Example - Using dynamicCall to call a dynamically compiled function

```
any compiled;

initialize {
    compiled = dynamicCompile("""
       void testA() {
          debug("testA called");
       }
   """);
}

consume {
    //
    // Call to a dynamically compiled function
    //
    dynamicCall(compiled, "testA", null);
}
```

dynamicCompile

The `dynamicCompile` function dynamically compiles APL Code and returns a handler that is used together with `dynamicCall`.

```
any dynamicCompile ( string aplCode )
```

Parameter	Description
`aplCode`	The APL Code to be compiled
Returns	Handler to use together with `dynamicCall`

Example - Using dynamicCompile

```
any aplCode = dynamicCompile("string helloWorld() { return \"Hello World!\"; }

debug("Compiled function returned "+dynamicCall(aplCode, "helloWorld"));
```

dynamicImport

The `dynamicImport` function imports generic APL Code.

The command returns a handler that is used together with `dynamicCall`.

Note!

Use this function only during the initialization of workflow.

```
any dynamicImport ( string  confName  )
```

Parameter	Description
`confName`	The configuration name of APL Code to import.
Returns	Handler to use together with `dynamicCall`.

Example - Using dynamicImport

```
any aplCode;
initialize {
    //
    // Dynamic Import of generic APL Code
    //
    aplCode = dynamicImport("Default.helloworld");
}
```

dynamicPlugin

The `dynamicPlugin` function initializes a DTK APL plug-in. For further information, see 10. APL Plugins in the Development Toolkit User's Guide.

The command returns a handler that you use together with `dynamicCall`.

Note!

Use this function only during the initialization of workflow.

```
any dynamicPlugin ( string classOrFunctionName )
```

Parameter	Description
`classOrFunctionName`	The name of the plug-inclass implementing `DRAPLExecutor`, or a function name that is defined by the plug-in
Returns	A handler that you use together with `dynamicCall`

Example - Using dynamicPlugin

```
any plugin1;
any plugin2;
initialize {
    //
    // Initialize DTK plugin - Using ClassName
    //
    plugin1 = dynamicPlugin(
                    "com.<product>.apl.MyPluginExecutor");
    //
    // Initialize DTK plugin - Using exposed function name
    //
    plugin2 = dynamicPlugin("myPluginExposedFunction");
}
```

15. External References Functions

externalReferenceGet

This function fetches an External Reference value from an external source. For information about External References, see 8.10 External Reference Profile in the Desktop user's guide.

```
string externalReferenceGet (< ext ref profile >, < ext ref key >)
```

Note!

If you have a large number of external references, for example greater than 1000, this may impact the workflow startup and processing. It is therefore recommended that extraction is done using an APL configuration in multiple functions and that extraction is executed in the initialize block of an Analysis agent.

Example - With large number of external reverences

If you have a large number of external references, the APL configuration may look as shown below:

```
final string EXT_REF = "Default.PRF_ExtRef";
map<string, string> keyValueMap;

void loadStrings1(){

    keyValueMap = mapCreate(string, string);
    mapSet(keyValueMap, "EXT_VAL1", externalReferenceGet(EXT_REF, "EXT_VAL1"));
    mapSet(keyValueMap, "EXT_VAL2", externalReferenceGet(EXT_REF, "EXT_VAL2"));
    mapSet(keyValueMap, "EXT_VAL3", externalReferenceGet(EXT_REF, "EXT_VAL3"));
    mapSet(keyValueMap, "EXT_VAL4", externalReferenceGet(EXT_REF, "EXT_VAL4"));
    mapSet(keyValueMap, "EXT_VAL5", externalReferenceGet(EXT_REF, "EXT_VAL5"));
    ...
}

void loadStrings1000(){

    keyValueMap = mapCreate(string, string);
    mapSet(keyValueMap, "EXT_VAL1001", externalReferenceGet(EXT_REF, "EXT_VAL1001"));
    mapSet(keyValueMap, "EXT_VAL1002", externalReferenceGet(EXT_REF, "EXT_VAL1002"));
    mapSet(keyValueMap, "EXT_VAL1003", externalReferenceGet(EXT_REF, "EXT_VAL1003"));
    mapSet(keyValueMap, "EXT_VAL1004", externalReferenceGet(EXT_REF, "EXT_VAL1004"));
    mapSet(keyValueMap, "EXT_VAL1005", externalReferenceGet(EXT_REF, "EXT_VAL1005"));
    ...
}
```

It is recommended that you configure an Analysis agent so that extraction is done in the initialize block only extracting the required references for the specific agent, as shown below:

```
import apl.Default.extreflib1;

initialize{
   loadStrings1();
}

consume {
...

}
```

Parameter	Description
`ext ref profile`:	The External Reference Profile
`ext ref key`:	The External Reference key
Returns	The External Reference value **Note!** To convert the returned value type from string to another data type, use one of the APL string conversion commands. For further information see 2.8 Type Conversion Functions

16. FNTUDR Functions

The functions described below operate on values in UDRs of type `FNTUDR.` The value in an `FNTUDR` is a delimited string, representing file system paths.

The FNTUDR types are usually used in two different cases:

- Creation of a dynamic path for the output file from a disk oriented forwarding agent. Using the filename template in the forwarding agent the `FNTUDR` value is, via a MIM resource, used to publish the respective MIM value. For further information about using the `FNTUDR` in filename templates, see the relevant agent documentation in the Desktop User's Guide.
- Grouping of output data using a disk oriented forwarding agent set to expect input of `MultiForwardingUDR` type. The `MultiForwardingUDR` has two fields, the first contains the data that will be written in the output file, the second contains filename and path specified by the FNTUDR. Both fields are mandatory unless the **Produce Empty Files** check box is selected in the agent, in this case the data field is not required.

For further information about using the `MultiForwardingUDR`, please refer to respective file based forwarding agents documentation.

Example - How an FNTUDR is constructed

The following APL code shows how a `FNTUDR` is constructed with the help of the functions that are described in this section.

```
import ultra.FNT;
consume {
 FNTUDR fntudr = udrCreate(FNTUDR);
 fntAddString(fntudr, "ready_folder");
 fntAddDirDelimiter(fntudr);
 date now = dateCreateNow();
 fntAddString(fntudr, ""+dateGetYear(now) + "-"); fntAddString(fntudr, ""+dateGetMonth(now), 2, "0",
false); fntAddString(fntudr, "-");
 fntAddString(fntudr, ""+dateGetDay(now), 2, "0", false);
}
```

The fntudr variable will have a value corresponding to a path like `ready_folder/2018-10-31`, where the character '/' represent a directory delimiter that can be any character, depending on the target system.

fntAddString

The `fntAddString` function appends a text string to the specified `FNTUDR`.

```
void fntAddString(
 FNTUDR fntudr,
 string str,
 int size, (optional)
 string padding, (optional)
 boolean leftAlignment (optional) )
```

Parameter	Description
fntudr	The FNTUDR that the text string is going to be added to. A runtime error will occur, if the parameter is null.
str	The string that will be added. If the string is null a runtime error will occur. The string can be extended or truncated if the size parameter is specified and the string does not match. **Note!** Do not include directory delimiters, e g "/", in the string. To add delimiters, use the `fntAddDirDelimiter` function instead.
size	The optional parameter size defines a fixed size for the appended string. A runtime error will occur, if the size isn't greater than zero.
padding	The optional parameter padding defines a padding string that will be repeated to fill the string to the size specified in the size parameter. A default padding will be used if the argument isn't specified or if the padding string is null or an empty string.
leftAlignment	The optional parameter leftAlignment specifies if the padding shall be kept right or left of the string. If the parameters value is true the string will be left aligned and the padding will added to the right. If the parameter isn't specified, the default setting is left aligned.
Returns	Nothing

fntAddDirDelimiter

The `fntAddDirDelimiter` function adds a directory delimiter to the end of the FNTUDR.

```
void fntAddDirDelimiter(FNTUDR fntudr)
```

Parameter	Description
fntudr	The FNTUDR that the text delimiter is going to be added to. A runtime error will occur, if the parameter is null.
Returns	Nothing

17. File Functions

You use the file management APL functions to manage files in MediationZone. These functions include:

- `readFileBinary`
- `fileDelete`
- `fileMove`
- `fileListFiles`
- `fileListDirectories`

> **Note!**
>
> These functions can only be used for the directories in your local file system.

readFileBinary

Reads a binary file and returns its contents in the MediationZone bytearray data format.

```
bytearray readFileBinary (string  path  )
```

Parameter	Description
path	The physical path and name of the file
Returns	The contents as a `bytearray`

fileDelete

The fileDelete command takes the absolute path of a file and deletes that file.

If the delete is successful, null is returned. If there is an error, an error string is returned.

```
string fileDelete (string  fileName  )
```

Parameter	Description
fileName	The absolute path to the file to delete
Returns	Null if the deletion was successful, otherwise an error string

fileMove

The fileMove command takes the path to the file to be moved, and the path to the new destination.

If the move is successful, null is returned. If there is an error, an error string is returned.

```
string fileMove
(string fileName ,
 string newFileName )
```

Parameter	Description
fileName	The absolute path to the file you want to move
newFileName	The absolute path to the new destination of the file (e.g "/home/files/20110505/movedFile.csv").
Returns	Null if the file was moved successfully, otherwise an error string

fileListFiles

The fileListFiles command lists all files in a given directory.

The command only returns actual files from the directory, it does not return any sub directories or files from sub directories. If there are no files in the given directory, an empty list is returned. If there is an error, null is returned.

```
list<string> fileListFiles
(string  directory,
 regex filter (optional))
```

Parameter	Description
directory	The absolute path to the directory for which you want to list files
filter	You can enter a regular expression to filter out which files you want to list.
Returns	A list (may be empty) of the files in the given directory, or null if there is an error

> **Note!**
>
> The files will not necessarily be listed in alphabetical, or any other specific order.

fileListDirectories

The fileListDirectories command lists all sub directories in a given directory.

The command only returns sub directories in the given directory, it does not return files or sub directories to the sub directories. If there are no directories in the given directory, an empty list is returned. If there is an error, null is returned.

```
list<string> fileListDirectories
 (string  directory ,
 regex  filter  (optional))
```

Parameter	Description
directory	The absolute path to the directory for which you want to list sub directories
filter	You can enter a regular expression to filter out which directories you want to list.
Returns	A list (may be empty) of the directories in the given directory, or null if there is an error

18. Hash and Checksum Functions

The following hash and checksum functions are available in APL:

- Checksum functions- Generates a checksum based on the cksum or CRC-32 algorithms for a supplied argument.
- Hash functions - Uses the algorithms MD2, MD5, SHA-1, SHA-256, SHA-384, or SHA-512 to turn input of arbitrary length into an output of fixed length

18.1 Checksum Functions

Use the functions described below to generates a checksum for a supplied argument.

cksumReset

Resets the cksum info and prepare to start new cksum calculation.

```
void cksumReset()
```

cksumUpdate

Updates the cksum using the specified array of bytes.

```
void cksumUpdate( bytearray data )
```

Parameter	Description
bytearray	Any bytearray
Returns	Nothing

cksumResult

Gets the result of cksum (using all data that has been entered into cksumUpdate). Returns an unsigned 32 bit integer (as a long).

```
long cksumResult()
```

cksumLength

Gets the total length of all data that has been entered into cksumUpdate.

```
long cksumLength()
```

cksum Example

Example - cksum

Simple APL code (Disk_1 is the input agent in the workflow)

```
beginBatch {
 cksumReset();
}
consume {
 cksumUpdate( input );
}
endBatch {
 debug( ((string)cksumResult()) + " " + cksumLength() + " " +
 ((string)mimGet("Disk_1","Source Filename")) );
}
```

crc32

Function that computes the CRC-32 of supplied argument.

```
long crc32( any  source  )
```

Parameter	Description
source	The argument used to compute the CRC-32. Supported types are string, bytearray and number types.
Returns	The CRC-32 value

18.2 Hash Functions

Use the functions below to turn bytearray input of arbitrary length into an output of fixed length

hashDigest

Completes the hash computation by performing final operations such as padding. After this call is made the digest is reset.

```
bytearray hashDigest();
```

Parameter	Description
Returns	A hexadecimal hash sum

hashReset

Resets the digest for further use.

```
void hashReset();
```

Parameter	Description
Returns	Nothing

hashUpdate

Updates the digest using the specified array of bytes.

```
void hashUpdate(bytearray);
```

Parameter	Description
bytearray	Any type of bytearray
Returns	Nothing

hashSetAlgorithm

Uses a digest that implements the specified digest algorithm.

> **Note!**
>
> The default setting, that is if the algorithm is not set, the SHA-1 hash method is used.

```
void hashSetAlgorithm(<"hash_method">);
```

Parameter	Description
<"hash_method">	The name of the algorithm requested. Supported hash algorithms are: MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512
Returns	Nothing.

hashSetAlgorithmFromProvider

Uses a digest that implements the specified digest algorithm, as supplied from the specified provider. The digest will be reset.

```
void hashSetAlgorithmFromProvider(string,string);
```

Parameter	Description
<"unique_hash">	The name of the hash algorithm developed by the provider
Provider	The name of the provider
Returns	Nothing

Hash Example

Example - Hash

```
initialize {
 hashSetAlgorithm("MD2");
}

beginBatch {
 hashReset();
}

consume {
 hashUpdate(input);
}

drain {
 bytearray hashsum = hashDigest();
 string sHashSum = baToHexString(hashsum); // Optional
 bytearray result; // Optional
 strToBA(result, sHashSum); // Optional
 udrRoute(result);
}
```

Note!

The optional parts in the example are added to convert the hash sum to a more readable form.

19. HTTP Functions

The HTTP functions are used to exchange data over HTTP or HTTPS as either client or server.

19.1 HTTP Client Functions

The client functions are used to exchange data with a HTTP/HTTPS server. There are specific functions for GET and POST as well as functions for general HTTP requests. Either plain text or encrypted communication can be used. Basic authentication is supported, as well as the use of a keystore, and if required a truststore, for the functions with an encrypted communication channel.

httpGetURL

This function uses the GET method to retrieve content from an HTTP server.

```
string httpGetURL
    ( string query,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                  //Optional
```

Parameter	Description

query	The query including path
	
	Example - query
	`/api/v2/doc/1`
	`/api/v2/doc?id=1`

host	The name or IP address where the HTTP server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
port	The port number to contact the HTTP server on. Port 80 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	Content from the HTTP server. It will be `null` if any part of the communication fails.

httpBinaryGetURL

This function uses the GET method to retrieve binary content from an HTTP server.

```
bytearray httpBinaryGetURL
    ( string query,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                 //Optional
```

Parameter	Description

query	The query including path
	
	Example - query
	`/api/v2/img/1`
	`/api/v2/img?id=1`

host	The name or IP address where the HTTP server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
port	The port number to contact the HTTP server on. Port 80 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A bytearray from the HTTP server. It will be `null` if any part of the communication fails.

httpGetSecureURL

This function uses the GET method to retrieve content from an HTTPS server. The communication channel is encrypted.

```
string httpGetSecureURL
    ( string query,
      string host,
      int timeout,
      int port,                    //Optional
      string username              //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                 //Optional
```

Parameter	Description

query	The query including path
	
	Example - query
	
	`/api/v2/doc/1`
	`/api/v2/doc?id=1`

host	The name or IP address where the HTTPS server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
port	The port number to contact the HTTPS server on. Port 443 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	Content from the HTTPS server. It will be `null` if any part of the communication fails.

See the section below, TLS/SSL Encryption.

httpBinaryGetSecureURL

This function uses the GET method to retrieve binary content from an HTTPS server. The communication channel is encrypted.

```
bytearray httpBinaryGetSecureURL
    ( string query,
      string host,
      int timeout,
      int port,                    //Optional
      string username              //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                 //Optional
```

Parameter	Description

query	The query including path
	
	Example - query
	
	`/api/v2/img/1`
	`/api/v2/img?id=1`

host	The name or IP address where the HTTPS server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
port	The port number to contact the HTTPS server on. Port 443 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A bytearray from the HTTPS server. It will be `null` if any part of the communication fails.

See the section below, TLS/SSL Encryption.

httpPostURL

This function uses the POST method to send content to an HTTP server and receives the response.

```
string httpPostURL
    ( string path,
      string contentType,
      string content,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                  //Optional
```

Parameter	Description
path	The path **Example - path** `/api/v2/doc`
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
host	The name or IP address where the HTTP server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
port	The port number to contact the HTTP server on. Port 80 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	Content from the HTTP server. It will be `null` if any part of the communication fails.

httpBinaryPostURL

This function uses the POST method to send binary content to an HTTP server and receives the response.

```
bytearray httpBinaryPostURL
    ( string path,
      string contentType,
      bytearray content,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                 //Optional
```

Parameter	Description
path	The path **Example - path** `/api/v2/img`
contentType	The MIME type of the content **Example - contentType** `/application/octet-stream`
content	The body of the request
host	The name or IP address where the HTTP server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. Default value is 15 seconds.
port	The port number to contact the HTTP server on. Port 80 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A bytearray from the HTTP server. It will be `null` if any part of the communication fails.

httpPostSecureURL

This function uses the POST method to send content to an HTTPS server and receives the response. The communication channel is encrypted.

```
string httpPostSecureURL
    ( string path,
      string contentType,
      string content,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                 //Optional
```

Parameter	Description
path	The path **Example - path** `/api/v2/doc`
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
host	The name or IP address where the HTTPS server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. Default value is 15 seconds.
port	The port number to contact the HTTPS server on. Port 443 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A document from the HTTP server. It will be `null` if any part of the communication fails.

See the section below, TLS/SSL Encryption.

httpBinaryPostSecureURL

This function uses the POST method to send binary content to an HTTP server and receives the response. The communication channel is encrypted.

```
bytearray httpBinaryPostSecureURL
    ( string path,
      string contentType,
      bytearray content,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers, //Optional
          int connectTimeout)                 //Optional
```

Parameter	Description
path	The path **Example - path** `/api/v2/img`
contentType	The MIME type of the content **Example - contentType** `/application/octet-stream`
content	The body of the request
host	The name or IP address where the HTTPS server is running
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. Default value is 15 seconds.
port	The port number to contact the HTTPS server on. Port 443 is used by default.
username	A username for an account on the HTTP server
password	Password associated with the username
headers	Custom HTTP request headers.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A bytearray from the HTTP server. It will be `null` if any part of the communication fails.

See the section below, TLS/SSL Encryption.

httpRequest

This function makes an HTTP request and uses the specified method, e g GET, POST, PUT etc.

```
string httpRequest
    ( string method,
      string url,
      map<string,string> header,
      string contentType,
      string content,
      int timeout,
          int connectTimeout)        //Optional
```

Parameter	Description
method	The HTTP method
url	The URL of the HTTP server
header	Key-value pairs containing request header fields
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
timeout	The number of milliseconds to wait for a response. A timeout of 0 (zero) is interpreted as an infinite timeout.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A response from the HTTP server. It will be `null` if any part of the communication fails.

httpRequestBasicAuth

This function makes an HTTP request with basic authentication and uses the specified method, e g GET, POST, PUT etc.

```
string httpRequestBasicAuth
    ( string method,
      string url,
      map<string,string> header,
      string contentType,
      string content,
      int timeout,
      string username
      string password,
          int connectTimeout)        //Optional
```

Parameter	Description
method	The HTTP method
url	The URL of the HTTP server
header	Key-value pairs containing request header fields
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
timeout	The number of milliseconds to wait for a response. A timeout of 0 (zero) is interpreted as an infinite timeout.
username	A username for an account on the HTTP server
password	Password associated with the username
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A response from the HTTP server. It will be `null` if any part of the communication fails.

httpRequestSecureBasicAuth

This function makes an HTTPS request with basic authentication and uses the specified method, e g GET, POST, PUT etc.

```
string httpRequestSecureBasicAuth
    ( string method,
      string url,
      map<string,string> header,
      string contentType,
      string content,
      int timeout,
      string username,
      string password,
          int connectTimeout)        //Optional
```

Parameter	Description
method	The HTTP method
url	The URL of the HTTPS server
header	Key-value pairs containing request header fields
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
timeout	The number of milliseconds to wait for a response. A timeout of 0 (zero) is interpreted as an infinite timeout.
username	A username for an account on the HTTP server
password	Password associated with the username
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A response from the HTTPS server. It will be `null` if any part of the communication fails.

See the section below, TLS/SSL Encryption.

httpRequestSecure

This function makes an HTTPS request and uses the specified method, e g GET, POST, PUT etc

```
string httpRequestSecure
    ( string method,
      string url,
      map<string,string> header,
      string contentType,
      string content,
      int timeout,
          int connectTimeout)        //Optional
```

Parameter	Description
method	The method to be performed on the HTTPS server identified by the URL
url	The URL of the HTTPS server
header	Key-value pairs containing request header fields
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
timeout	The number of milliseconds to wait for a response. A timeout of 0 (zero) is interpreted as an infinite timeout.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
Returns	A response from the HTTPS server. It will be `null` if any part of the communication fails.

See the section below, TLS/SSL Encryption.

httpRequestDigestAuth

This function sends a request to the server using Digest Authentication.

```
string httpRequestDigestAuth
    ( string method,
      string url,
      map<string,string> header,
      string contentType,
      string content,
      int timeout,
      string username,
      string password,
      int connectTimeout,                        //Optional
      string realm)                                //Optional
```

Parameter	Description
method	The HTTP method
url	The URL of the HTTP server
header	Key-value pairs containing request header fields
contentType	The MIME type of the content **Example - contentType** `/application/json`
content	The body of the request
username	The username required for authentication
password	The password required for authentication
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
connectTimeout	The number of seconds to wait for a connection to be established. If the value is set to 0 (zero), the default timeout will be used. The default value is 15 seconds.
realm	The name of the realm
Returns	A response from the HTTP server. It will be null if any part of the communication fails.

httpCreateMultipartBody

This function creates a multipart body that can be used in HTTP requests that accept a body.

```
any httpCreateMultipartBody ( )
```

Parameter	Description
Returns	A multipart body for HTTP requests to which typed binary segments can be added.

httpCreateMultipartSegment

This function adds a typed binary segment to a multipart body that has been created using `httpCreateMultipartBody` function.

```
void httpAddMultipartSegment
    ( any body,
      string filename,
      string contentType,
      bytearray content )
```

Parameter	Description
body	A multipart body created by using the `httpCreateMultipartBody` function.
filename	The filename to be associated with the segment.
contentType	The MIME type of the content (according to RFC1341).
content	The binary content of the request.
Returns	Nothing

httpCreateMultipartSegmentWithMapping

This function adds a typed binary segment to a multipart body that has been created using httpCreateMultipartBody function and allows to specify a custom key name.

```
void httpAddMultipartSegmentWithMapping
    ( any body,
      string keyName
      string filename,
      string contentType,
      bytearray content )
```

Parameter	Description
body	A multipart body created by using the `httpCreateMultipartBody` function.
keyName	A custom key name that will be used for the fragment
filename	The filename to be associated with the segment.
contentType	The MIME type of the content (according to RFC1341).
content	The binary content of the request.
Returns	Nothing

httpMultipartPostUrl

This function uses the POST method to send multipart binary contents to an HTTP server and receives the response.

```
bytearray httpMultipartPostURL
    ( string path,
      any body,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers) //Optional
```

Parameter	Description
path	The path on the server to which we should do the POST.
body	A multipart body created by using the `httpCreateMultipartBody` function and populated using the `httpAddMultipartSegment` function.
host	The name or IP address of the HTTPS server.
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. Default value is 15 seconds.
port	The port to be used for the HTTPS server. Port 80 is used by default.
username	Username for the account to be used on the HTTPS server.
password	Password for the account to be used on the HTTPS server.
headers	Custom HTTP request headers.
Returns	A bytearray from the HTTPS server. It will be null if any part of the communication fails.

Note!

For optional parameters, you need to state null in case you state subsequent optional parameters. If there are no subsequent parameters, you do not have to state anything.

Example

```
httpMultipartPostURL( "mypath", mybody, ""myhost, 0, null, "myusername");
```

httpMultipartPostSecureUrl

This function uses the POST method to send multipart binary content to an HTTPS server and receive the response.

```
bytearray httpMultipartPostSecureURL
    ( string path,
      any body,
      string host,
      int timeout,
      int port,                    //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers) //Optional
```

Parameter	Description
path	The path on the server to which we should do the POST.
body	A multipart body created by using the `httpCreateMultipartBody` function and populated using the `httpAddMultipartSegment` function.
host	The name or IP address of the HTTPS server.
timeout	The number of seconds to wait for a response. If the value is set to 0 (zero), the default timeout will be used. Default value is 15 seconds.
port	The port to be used for the HTTPS server. Port 80 is used by default.
username	Username for the account to be used on the HTTPS server.
password	Password for the account to be used on the HTTPS server.
headers	Custom HTTP request headers.
Returns	A bytearray from the HTTPS server. It will be null if any part of the communication fails.

Note!

For optional parameters, you need to state null in case you state subsequent optional parameters. If there are no subsequent parameters, you do not have to state anything.

Example

```
httpMultipartPostSecureURL( "mypath", mybody, ""myhost, 0, null, "myusername");
```

TLS/SSL Encryption

The following functions require a keystore in order to work, and if required, the use of a truststore is supported:

- `httpGetSecureURL`
- `httpBinaryGetSecureURL`
- `httpPostSecureURL`
- `httpBinaryPostSecureURL`
- `httpRequestSecureBasicAuth`
- `httpRequestSecure`

Configure Java Keystore for Secure URL Functions

Keystore is used to store HTTP Client's credential. This certificate is sent to a server for authentication if required.

To specify a Keystore file that you want to use, set the properties `https.apl.keystore_location` and `https.apl.keystore_passphrase` in the relevant ECs. See the example below on how to set these properties using the `mzsh topo` command.

Example - Setting Java keystore properties

```
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
keystore_location <location of the Keystore file>
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
keystore_passphrase <DR encrypted password>
```

If the two properties above are not set in the relevant Execution Context <pico>`.conf`, MZ Default Keystore is used.

The property *https.apl.keystore_location* represents the location of the keystore and the property *https.apl.keystore_passphrase* represents the passphrase for that keystore.

The following command can be used to create a keystore with the Java keytool program.

```
keytool -keystore /var/opt/mz/HttpdExec.keystore -genkey -keyalg RSA
```

Note!

The Keystore passphrase must be the same as the passphrase used by the certificate.

See the JVM product documentation for more information about how to use the keytool.

Configure Java Truststore for Secure URL Functions

A truststore is a keystore that is used when deciding on what to trust - truststore stores certificates from third parties. If you receive data from an entity that you already trust, and if you can verify that the entity is that which it claims to be, you can assume that the data does in fact come from that entity.

By Default, MediationZone uses its own truststore, which always trusts any server connection.

If you want to use a specific truststore, use the `mzsh topo` command to add the property `https.apl.userdefined.truststore` to the required ECs and set the value to `true`:

```
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.userdefined.
truststore true
```

The default value of this property is `false`.

After setting the property `https.apl.userdefined.truststore` to `true`, if you want to use a specific truststore, use the `mzsh topo` command to set the following properties in the relevant ECs:

- `https.apl.truststore_location`
- `https.apl.truststore_passphrase`

Example - Setting properties to use a specific truststore

```
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
truststore_location <location of the truststore file>
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
truststore_passphrase <DR encrypted password>
```

If you do not set these two properties, the Java Default Truststore is used.

19.2 HTTP Client Helper Functions

The HTTP client helper functions are used for error handling.

httpGetLastErrorCode

This function gets the last error code set.

```
int httpGetLastErrorCode()
```

Parameter	Description
Returns	The HTTP status code. It will be `-1` if no error occurred. Note that calling the `httpGetLastErrorCode()` function should only be done when the previous POST/GET request returned null.

httpGetLastErrorMessage

This function gets the last error message.

```
string httpGetLastErrorMessage()
```

Parameter	Description
Returns	The HTTP status message. It will be `null` if no error occurred. Note that calling the `httpGetLastErrorMessage()` function should only be done when the previous POST/GET request returned null.

httpGetLastErrorResponse

This function gets the last response if the connection failed but the server sent useful data nonetheless.

```
string httpGetLastErrorResponse()
```

Parameter	Description
Returns	A document from the HTTP server. It will be an empty string if no error occurred. Note that calling the `httpGetLastErrorResponse()` function should only be done when the previous POST/GET request returned null.

httpGetLastResponseHeader

This function retrieves the header from the last HTTP response.

```
map<string,string> httpGetLastResponseHeader()
```

Parameter	Description
Returns	A map containing the response header

19.3 HTTP Server Functions

This section describes the server functions that are used to to manage requests (UDRs) from the HTTPD collector agent.

httpGetRequestCookie

This function searches for a cookie name in the httpdUDR and returns the contents.

```
string httpGetRequestCookie
( any httpdUDR ,
string cookieName )
```

Parameter	Description
httpdUDR	The request
cookieName	A cookie is made up of a key:value data object, where cookieName is the key. **Note!** The value can be "null".
Returns	The contents (value) of the cookie.

httpGetRequestCookieNames

This function retrieves a list of cookie names (keys) from the request (UDR).

```
list <string> httpGetRequestCookieNames ( any httpdUDR )
```

Parameter	Description
httpdUDR	The request
Returns	A list of cookie names

httpSetResponseCookie

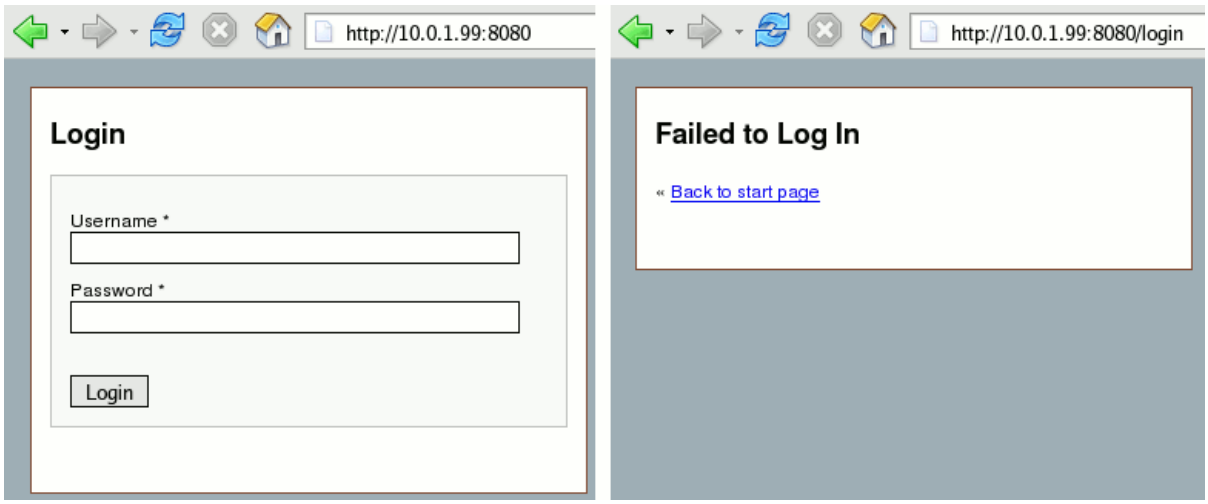
This function sets the response cookie (UDR) fields.

```
void httpSetResponseCookie
 ( any httpdUDR,
 string name,
 string value,
 boolean discard,
 string domain,
 long maxAge,
 string path,
 boolean secure )
```

Parameter	Description
httpdUDR	The response
name	The name of the cookie (the key in the key:value cookie object)
value	The contents of the cookie (the value in the key:value cookie object)
discard	Set to "true" to discard the cookie at browser close, or "false", to keep it.
domain	The domain to which the cookie should be sent
maxAge	The life-length of the cookie in seconds
path	The path within the domain that triggers the cookie
secure	Set to "true" for sending the cookie only to secure sites, or "false", for sending the cookie to any site within the domain.
Returns	Nothing

19.3.1 HTTP Server Example

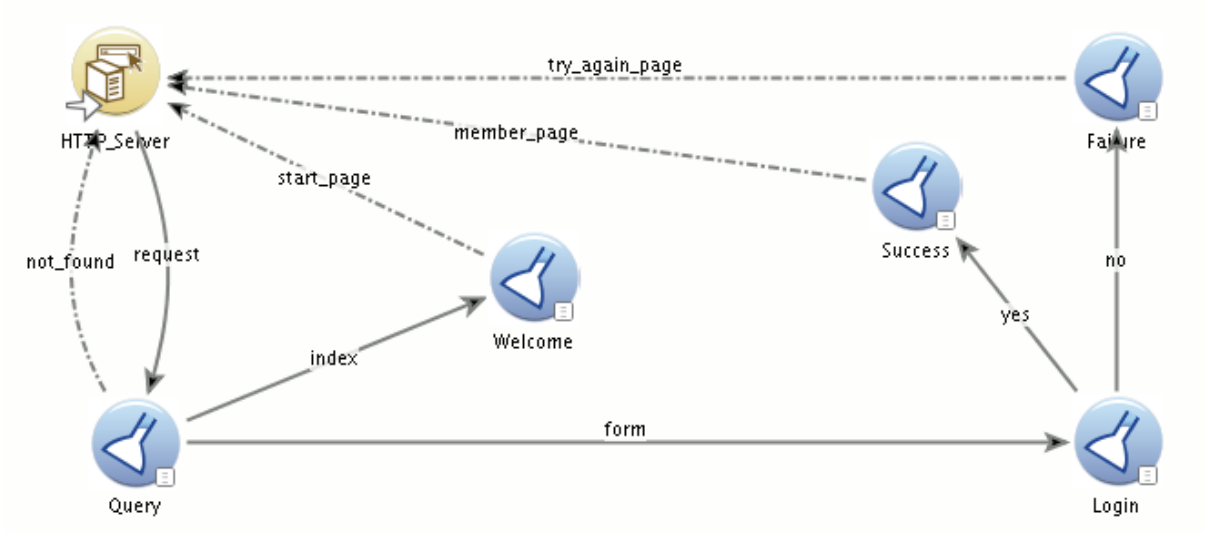
To give an example of how a workflow acting as a HTTP server may be designed, suppose there is an HTML login page requiring login name and password. The login parameters are checked against a user database, opening a new HTML page if authentication succeeded. If not, a new login attempt is requested.



The login page and the resulting message page, in case of failed authentication

An example of how such a workflow may be designed is shown below. Each of the Analysis agents will perform a validation, resulting in either the HTTP UDR being sent back with an updated `response` field to the collector, or it is sent unchanged to a subsequent agent.

1. The `HTTPD` agent sends a request to the `Query` agent which checks that the correct page is requested. If the page does not exist, an error message is sent back to the HTTPD agent on the `not_found` route.
2. If the page exists, the HTTP UDR is routed on to the `Welcome` or `Login` agents, depending on if the login page was addressed.
3. The `Login` agent checks if the user exists in the database. If authentication fails, the HTTP UDR is routed to the `Failure` agent which displays a new page stating the username/password is incorrect.
4. If authentication succeeds, a new page is opened.



A workflow acting as an HTTP Server

Note!

The data sent to the HTTP agent in the example (the `content` field) is an HTML page. Since UFDL cannot handle the HTML protocol, the field is defined as a string. If, for instance, XML was used instead (which may be handled with UFDL), this would require an Analysis agent, which turned the field into a bytearray and then decoded it (use the APL functions `strToBA` and `udrDecode`).

The Format Definition

Create an instance of the built-in HTTP format definition. Additional fields may be entered. This is useful mainly for transport of variable values to subsequent agents. In this case, re-usage of the username as parsed from the login page is desired.

```
internal MYHTTPD: extends_class ("com.digitalroute.wfc.http.HttpdUDR")
{
  string username;
};
```

The Analysis Agents

The Analysis agents handle all validation and response handling.

Query

The query agent checks if the addressed page exists. If it does not, an error message is inserted into the `response` field and the UDR is returned to the collector.

```
consume
{
  if (input.query == "/") {

    // Show welcome page.

    udrRoute(input, "index");
    return;
  }

  if (input.query == "/login") {

    // Try to log in.

    udrRoute(input, "form");
    return;
  }

  // The request is not found here.

  input.response = "HTTP ERROR: 404, Not Found";
  input.responseStatusCode = "404 Not Found";
  input.responseType = "text/plain";
  udrRoute(input, "not_found");
}
```

Welcome

Populates the `response` field with an HTML page, and send the UDR back to the collector.

```
final string welcome = "<html >
                    <title >Welcome</title >
                    <body bgcolor=#3b7d73 >
                    <font face=Verdana,Arial >
                    <h2 >HTTPD Doc Example</h2 >
                    Login<p ><form action=/login method=post >
                    <table ><tr ><td align=right >
                    Username</td ><td ><input name=username ></td ></tr >
                    <tr ><td align=right >Password</td >
                    <td ><input name=passwd type=password ></td ></tr >
                    <tr ><td > </td >
                    <td ><input name=Login type=submit ></td ></tr>
                    </table></form ></body ></html >";

consume
{
  input.responseType="text/html";
  input.response = welcome;
  udrRoute(input);
}
```

Login

Verifies that the user is authenticated, by performing a lookup against a database table.

```
table tmp_tab;

initialize {
  tmp_tab = tableCreate("select user_Name, user_PW from users");
}

consume
{
  if (input.requestMethod != "POST") {
    input.response = "Wrong method.";
    udrRoute(input, "No");
    return;
  }

  // Find username.

  string p = input.content;

int i = strIndexOf(p, "username=");
  p = strSubstring(p, i + 9, strLength(p));
  i = strIndexOf(p, "&");
  string u = strSubstring(p, 0, i);
  p = strSubstring(p, i + 1, strLength(p));
  i = strIndexOf(p, "passwd=");
  p = strSubstring(p, i + 7, strLength(p));
  i = strIndexOf(p, "&");
  p = strSubstring(p, 0, i);

  // Verify the login.

  table rowFound_u = tableLookup(tmp_tab, "user_Name", "=", u);
  int hit_u = tableRowCount(rowFound_u);

 if (hit_u == 1) {
      if (p == (string)tableGet(rowFound_u,0,1)) {
          input.username = (string)tableGet(rowFound_u,0,0);
          udrRoute( input, "Yes" );
      } else {
          input.response = "Wrong password.";
          udrRoute(input, "No");
      }
  } else {
      input.response = "Wrong username.";
      udrRoute(input, "No");;
  }
}
```

Success

The `response` field is populated with an HTML page.

```
final string success1 = " <html > <title >Welcome </title >
                    <body bgcolor=#3b7d73 > <font face=Verdana,Arial >
                    <h2 >HTTPD Doc Example </h2 >Welcome ";

final string success2 = "! <p > <a href=/ >Back to start page </a >
                    </body > </html >";

consume
{
  input.responseType = "text/html";
  input.response = success1 + input.username + success2;
  udrRoute(input);
}
```

Failure

The `response` field is populated with an HTML page.

```
final string failure1 = "<html><title>Welcome</title>
                    <body bgcolor=#3b7d73><font face=Verdana,Arial>
                    <h2>HTTPD Doc Example</h2>Failed to log in; ";

final string failure2 = "<p><a href=/>Back to start page</a>
                    </body></html>";

consume
{
    input.responseType = "text/html";
    input.response = failure1 + input.response + failure2;
    udrRoute(input);
}
```

20. HTTP/2 Functions

The HTTP/2 functions are used to exchange data over HTTP/2 as a client. However, the functions can also be used for HTTP/1.

The client functions are used to exchange data with a HTTP server. There are specific functions for GET and POST as well as functions for general HTTP requests. Either plain text or encrypted communication can be used. Basic authentication is supported, as well as the use of a keystore, and if required a truststore, for the functions with an encrypted communication channel.

> **Note!**
>
> In all parameter descriptions below, "HTTP" may refer to both HTTP and HTTPS, and both HTTP/1 and HTTP/2.

httpGet

This function uses the GET method to retrieve content from an HTTP server.

```
string httpGet
    ( string host,
      string path,
      string protocol,                          //Optional
      int port,                    //Optional
      boolean secure,              //Optional
      int requestTimeout,          //Optional
      int connectionTimeout,       //Optional
      string username,             //Optional
      string password,             //Optional
      map<string, string> headers ) //Optional
```

Parameter	Description
host	The name or IP address of the HTTP server.
path	The path. **Example - path** `/api/v2/doc`
protocol	The protocol used: HTTP/1 or HTTP/2. The default value is HTTP/1.
port	The port number to contact the HTTP server on. Port 80 is used for HTTP connection and 443 is used for HTTPS connection by default.
secure	Indicates whether the data should be sent in secure mode or not.
requestTimeout	The number of milliseconds to wait for a response. If the value is not specifically specified, the default timeout is used. The default value is 15000 milliseconds.
connectionTimeout	The number of milliseconds to wait for a connection to be established. If the value is not specifically specified, the default timeout is used. The default value is 3000 milliseconds.
username	A username for an account on the HTTP server.
password	Password associated with the username.
headers	Custom HTTP request headers.
Returns	Content from the HTTP server. It will be `null` if any part of the communication fails.

httpPost

This function uses the POST method to send content to an HTTP/2 server and receives the response.

```
string httpPost
    ( string host,
      string path,
      bytearray content,
      string contentType,
      string protocol,                              //Optional
      int port,                     //Optional
           boolean secure,               //Optional
      int requestTimeout,           //Optional
      int connectionTimeout,        //Optional
      string username,              //Optional
      string password,              //Optional
      map<string, string> headers ) //Optional
```

Parameter	Description
host	The name or IP address of the HTTP server.
path	The path **Example - path** `/api/v2/doc`
bytarraycontent	The body of the request in bytearray format.
contentType	The MIME type of the content. **Example - contentType** `application/json`
protocol	The protocol used: HTTP/1 or HTTP/2. The default value is HTTP/1.
port	The port number to contact the HTTP server on. Port 80 is used for HTTP connection and 443 is used for HTTPS connection by default.
secure	Indicates whether the data should be sent in secure mode or not.
requestTimeout	The number of milliseconds to wait for a response. If the value is not specifically specified, the default timeout is used. The default value is 15000 milliseconds.
connectionTimeout	The number of milliseconds to wait for a connection to be established. If the value is not specifically specified, the default timeout is used. The default value is 3000 milliseconds.
username	A username for an account on the HTTP server.
password	Password associated with the username.
headers	Custom HTTP request headers.
Returns	Content from the HTTP server. It will be `null` if any part of the communication fails.

httpReq

This function makes an HTTP request and uses the specified method, for example GET, POST, and PUT.

```
string httpReq
    ( string method,
          string host,
      string path,
      bytearray content,
      string contentType,
          string protocol,                          //Optional
          int port,                                          //Optional
          boolean secure,                           //Optional
      int requestTimeout,                    //Optional
          int connectionTimeout,            //Optional
          string username,                           //Optional
          string password,                            //Optional
      map<string,string> headers )//Optional
```

Parameter	Description
method	The HTTP method.
host	The name or IP address of the HTTP server.
path	The path **Example - path** `/api/v2/doc`
content	The body of the request in bytearray format.
contentType	The MIME type of the content. **Example - contentType** `application/json`
protocol	The protocol used: HTTP/1 or HTTP/2. The default value is HTTP/1.
port	The port number to contact the HTTP server on. Port 80 is used for HTTP connection and 443 is used for HTTPS connection by default.
secure	Indicates whether the data should be sent in secure mode or not.
requestTimeout	The number of milliseconds to wait for a response. If the value is not specifically specified, the default timeout is used. The default value is 15000 milliseconds.
connectionTimeout	The number of milliseconds to wait for a connection to be established. If the value is not specifically specified, the default timeout is used. The default value is 3000 milliseconds.
username	A username for an account on the HTTP server.
password	Password associated with the username.
headers	Custom HTTP request headers.
Returns	A response from the HTTP server. It will be `null` if any part of the communication fails.

httpMultipartPost

This function uses the POST method to send multipart binary contents to an HTTP server and receives the response.

```
bytearray httpMultipartPost
    ( string host,
          string path,
          list<MultipartSegmentUDR> content,
          string protocol,                            //Optional
             int port,                                     //Optional
          boolean secure,                             //Optional
      int requestTimeout,                    //Optional
          int connectionTimeout,             //Optional
      string username,              //Optional
      string password,              //Optional
      map<string, string> headers ) //Optional
```

Parameter	Description
host	The name or IP address of the HTTP server.
path	The path on the server to which we should do the POST.
content	The body of the the request.
protocol	The protocol used: HTTP/1 or HTTP/2. The default value is HTTP/1.
port	The port to be used for the HTTP server. Port 80 is used for HTTP connection and 443 is used for HTTPS connection by default.
secure	Indicates whether the data should be sent in secure mode or not.
requestTimeout	The number of milliseconds to wait for a response. If the value is not specifically specified, the default timeout is used. The default value is 15000 milliseconds.
connectionTimeout	The number of milliseconds to wait for a connection to be established. If the value is not specifically specified, the default timeout is used. The default value is 3000 milliseconds.
username	Username for the account to be used on the HTTP server.
password	Password associated with the username.
headers	Custom HTTP request headers.
Returns	A bytearray from the HTTP server. It will be null if any part of the communication fails.

Note!

For optional parameters, you need to state null in case you supply subsequent optional parameters. If there are no subsequent parameters, you do not have to state anything.

Example

```
httpMultipartPost( "muHost", "mypath", myMultipartSegmentUDRList, "myProtocol", 8080, false,
null, null, "muUsername");
```

TLS/SSL Encryption

When selecting secure, a keystore is required. In this case, the use of a truststore is supported.

Configure Java Keystore for Secure URL Functions

Keystore is used to store HTTP Client's credential. This certificate is sent to a server for authentication if required.

To specify a Keystore file that you want to use, set the properties `https.apl.keystore_location` and `https.apl.keystore_passphrase` in the relevant ECs. See the example below for how to set these properties using the `mzsh topo` command.

Example - Setting Java keystore properties

```
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
keystore_location <location of the Keystore file>
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
keystore_passphrase <DR encrypted password>
```

If the two properties above are not set in the relevant Execution Context <pico>`.conf`, MZ Default Keystore is used.

The property *`https.apl.keystore_location`* represents the location of the keystore and the property *`https.apl.keystore_passphrase`* represents the passphrase for that keystore.

The following command can be used to create a keystore with the Java keytool program.

```
keytool -keystore /var/opt/mz/HttpdExec.keystore -genkey -keyalg RSA
```

Note!

The Keystore passphrase must be the same as the passphrase used by the certificate.

See the JVM product documentation for more information about how to use the keytool.

Configure Java Truststore for Secure URL Functions

A truststore is a keystore that is used when deciding what to trust - truststore stores certificates from third parties. If you receive data from an entity that you already trust, and if you can verify that the entity is what it claims to be, you can assume that the data does in fact come from that entity.

By Default, MediationZone uses its own truststore, which always trusts any server connection.

If you want to use a specific truststore, use the `mzsh topo` command to add the property `https.apl.userdefined.truststore` to the required ECs and set the value to `true`:

```
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.userdefined.
truststore true
```

The default value of this property is `false`.

After setting the property `https.apl.userdefined.truststore` to `true`, if you want to use a specific truststore, use the `mzsh topo` command to set the following properties in the relevant ECs:

- `https.apl.truststore_location`
- `https.apl.truststore_passphrase`

Example - Setting properties to use a specific truststore

```
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
truststore_location <location of the truststore file>
$ mzsh topo set topo://container:<container>/pico:<pico>/val:config.properties.https.apl.
truststore_passphrase <DR encrypted password>
```

If you do not set these two properties, the Java Default Truststore is used.

21. JSON Functions

The JSON functions are used to to decode and encode JSON formatted strings.

> **Note!**
>
> The JSON parser is designed according to the specification RFC7159, and may accept non-JSON input, as stated in section 9 of RFC7159.

21.1 JSON Decoding Functions

The functions described in this section are used to decode JSON formatted strings.

jsonDecode

This function decodes a JSON formatted string to a `json.JsonObject` UDR or a `json.JsonArray UDR`.

The field `asMap` in a `json.JsonObject` UDR contains key-value pairs where the value is a string, another `json.JsonObject` UDR or a `json.JsonArray` UDR.

The field `asList` in a `json.JsonArray` UDR contains a list of `json.JsonObject` and `json.JsonArray` UDRs.

```
any jsonDecode ( string jsonString)
```

Parameter	Description
jsonString	The JSON formatted string to decode
Returns	A `json.JsonObject` UDR or a `json.JsonArray` UDR

> **Note**
>
> jsonDecode can only decode valid JSON.
>
> If the input is split due to size, the JSON may be invalid. The workaround for this is to use the baAppend function as shown in the example below.

Example - Example - Decoding JSON formatted string to json.JsonObject and json.JsonArray

```
import ultra.json;

bytearray ba;

consume {
    ba = baAppend(ba, input);
}

drain {
        if (ba != null) {
            string s = baToStr(ba);
            any js = jsonDecode(s);

            if (isJsonObject(js)) {
                debugJsonObject((JsonObject) js, "");
            } else if (isJsonArray(js)) {
                debugJsonArray((JsonArray) js, "");
            }
        }
}

void debugJsonObject(JsonObject js, string parent) {
    map<string, any> mapObj = js.asMap;
    list<string> ikeys = mapKeys(mapObj);
    string path;

    if (strLength(parent) > 0) {
        path = path + "/" + parent;
    }

    for (string i : ikeys) {
        any value = mapGet(mapObj, i);

        if (isJsonObject(value)) {
            debugJsonObject((JsonObject) value, i);
        } else if (isJsonArray(value)) {
            debugJsonArray((JsonArray) value, i);
        } else {
            debug(path + "/" + i + ":" + value);
        }
    }
}

void debugJsonArray(JsonArray ja, string parent) {
    list<any> array = ja.asList;

    for (any i : array) {
        if (isJsonObject(i)) {
            debugJsonObject((JsonObject) i, parent);
        } else if (isJsonArray(i)) {
            debugJsonArray((JsonArray) i, parent);
        }
    }
}

boolean isJsonObject(any value) {
    return instanceOf(value, JsonObject);
}

boolean isJsonArray(any value) {
    return instanceOf(value, JsonArray);
}
```

jsonDecodeUdr

This function decodes a JSON formatted string to a DRUDR.

```
void jsonDecodeUDR ( string jsonString,
                     DRUDR udr,
                     boolean requireExactMapping (optional) )
```

Parameter	Description
jsonString	The JSON string to decode
udr	The UDR to store the decoded data
requireExactMapping	Set to true if there must be matching UDR fields for all fields in the JSON string. The default value is `false`. When `requireExactMapping` is set to `true` and the function fails to match a field, it will throw an exception. If the function is used in batch workflow, the exception will cause it to abort. In a real-time workflow, the function does not decode the string, but the workflow does not abort. When `requireExactMapping` is set to false, the function will decode all JSON fields that can be mapped to a UDR field, the rest of the string will be ignored.
Returns	Nothing

Example - Encoding UDR to JSON and decoding the result

Ultra:

```
internal itemUDR {
    int itId;
    string itDescription;
    string itMisc1 : optional;
    list<string> itMisc2 : optional;
};

internal transactionUDR {
    int trId ;
    string trDescription;
    float amount;
    long trDate;
    map<string, itemUDR> trItems;
    boolean trProcessed;
    ipaddress trSourceIP;

};
```

APL:

```
import ultra.JSON_EXAMPLE.ULTRA_JSON_EXAMPLE;

consume {
    //Item 1
    itemUDR item1 = udrCreate(itemUDR);
    item1.itId = 1;
    item1.itDescription = "Item1";
    item1.itMisc2 = listCreate(string);
    listAdd(item1.itMisc2, "abc");
    listAdd(item1.itMisc2, "def");
    listAdd(item1.itMisc2, "ghi");

    //Item2
    itemUDR item2 = udrCreate(itemUDR);
    item2.itId = 1;
    item2.itDescription = "Item2";
    item2.itMisc1 = "abc";

    //Transaction1
    transactionUDR transaction1  = udrCreate(transactionUDR);
    transaction1.trId = 1;
    transaction1.trDescription = "Transaction1";
    transaction1.amount = 999.99;
    transaction1.trDate = dateCreateNowMilliseconds();
    transaction1.trItems = mapCreate(string, itemUDR);
    mapSet(transaction1.trItems, "item1Key",item1);
    mapSet(transaction1.trItems, "item2Key",item1);
    transaction1.trProcessed = true;
    transaction1.trSourceIP = ipLocalHost();

    //Encode with JSON
    string json;
    json = jsonEncodeUdr(transaction1);
    bytearray ba;
    strToBA(ba, json);
    debug("Encoded JSON (" + baSize(ba) + "bytes):\n" + json + "\n" );

    //Decode from JSON
    transactionUDR transactionIn = udrCreate(transactionUDR);
    jsonDecodeUdr(json, transactionIn);
    debug("Decoded JSON:\n" + transactionIn);
```

21.2 JSON Encoding Functions

The functions described in this section are used to to encode lists, maps, or UDRs to JSON formatted strings.

jsonEncodeList

This function encodes a list to a JSON formatted string.

```
string jsonEncodeList ( list<any>list)
```

Parameter	Description
list	The list to encode
Returns	A JSON formatted string

jsonEncodeMap

This function encodes a map to a JSON formatted string.

```
string jsonEncodeMap ( map<string,any> map)
```

Parameter	Description
map	The map to encode
Returns	A JSON formatted string

jsonEncodeUdr

This function encodes a UDR to a JSON formatted string.

> **Info!**
>
> For cases where you would not want Big Integer or Big Decimal values to be converted into string, you can enable `mz.apl.jsonencodeudr.ignore.bigint.convert` in 2.6.3 Execution Context Properties.

```
string jsonEncodeUdr ( DRUDR udr)
```

Parameter	Description
udr	The UDR to encode.
Returns	A JSON formatted string.

22. LDAP Functions

The LDAP functions allow you to connect to an LDAP, using anonymous or simple authentication, modify and delete entries or perform searches based on names, and filters, and explicitly define what attributes to be returned. The LDAP functions use connection pooling for best performance.

> **Note!**
>
> All functions except ldapCreate are asynchronous in nature. The ldapCreate function is recommended to be called from the initialized block and the returned value to be stored in a global variable.

LDAP Related UDR Type

The UDR type created by default in the LDAP agent can be viewed in the UDR Internal Format Browser. To open the browser open an APL Editor, in the editing area right-click and select UDR Assistance...; the browser opens.

Error Handling

The default error handling for the functions `ldapCreate`, `ldapSearch`, and `ldapScopeSearch` is to handle all errors as exceptions, which means that the workflow will abort in the batch case and the request will typically be discarded in the real-time case.

If this is not desired behavior, it is possible to set these LDAP functions to suppress all communication errors and instead return `null` in error situations. In this case, the error will be made available through the `ldapGetLastError` function.

Network Timeout Property

If there are network problems when communicating with the LDAP server the LDAP plugin commands use a network timeout of 5 minutes. This default timeout can be modified by setting the property `mz.ldap.network.timeout` to a timeout in milliseconds.

Set the property in the `cell.conf` or the relevant Execution Context <pico>`.conf`, depending on where the workflow is executed.

Idle Connection Timeout

By default, a connection remains within a pool in an idle state for five minutes before it is closed. To set the amount of time in milliseconds, set the property `com.sun.jndi.ldap.connect.pool.timeout` in the `cell.conf` or the relevant Execution Context <pico>`.conf`, depending on where the workflow is executed.

ldapCreate

Creates a connection towards an LDAP server, using either anonymous or simple authentication. This function is usually invoked in the `initialize` block.

```
any ldapCreate
   ( string  host ,
     int  port ,
     string  name ,
     string  principal ,    //Optional
     string  credentials  ) //Optional
```

Parameter	Description
host	The host name of the LDAP server
port	The port number of the LDAP server
name	Name which identifies the context in which the search will be performed. The value can span multiple naming systems and must be fully qualified and identify entries from the root of the LDAP server. If specified as `null` this means that the search will be performed from the LDAP root.
principal	Optional argument that defines the user to connect as. If omitted, the function will connect to the LDAP server using anonymous authentication. This argument requires that the credentials argument is supplied.
credentials	Optional argument that defines the user password.
Returns	An identifier used when invoking the search function. If `ldapSuppressErrors` has been called, then this function returns null if an error is detected during communication with the LDAP server. `ldapGetLastError` should be used to get the error message in this case.

Example - Using ldapCreate

```
any ctx = ldapCreate("10.0.0.1", 389, "o=users");
...
any ctx = ldapCreate("10.0.0.1", 389, "o=users",
            "cn=Administrator, o=users", "secret");
```

ldapAdd

The command is used to add an entry.

```
string ldapAdd
   ( any  identifier ,
     string  name ,
     list<string>  attributes  )
```

Parameters:

Parameter	Description
identifier	The connection identifier returned from `ldapCreate`.
name	The name of the entry to add. This is matched relative to the context specified with the *name* argument in `ldapCreate`. If no object matches the name, an error will be thrown stating there is no such name.
attributes	The new attribute value. If a delete operation is requested, the matching value is deleted.
Returns	In the event of an error, a message from LDAP server will be returned otherwise null.

ldapCloseCtx

The command is used to close a context that has previously been returned by `ldapCreate`.

```
int ldapCloseCtx
 ( object context )
```

Parameter	Description
context	The context returned from `ldapCreate`.
Returns	If the function is successful, it will return 0.

ldapDelete

An APL command used to delete an entry.

```
string ldapDelete
   ( any  identifier ,
     string  name )
```

Parameter	Description
identifier	The connection identifier returned from `ldapCreate`.
name	The name of the entry to be deleted.
Returns	In the event of an error, a message from LDAP server will be returned otherwise null.

ldapGetLastError

Retrieves the error message of the last error reported for an LDAP command.

```
string ldapGetLastError()
```

Parameter	Description
Returns	The last error message that an LDAP function registered. This information is kept per thread and agent so it is only guaranteed to be present in the same APL code block as the function call causing the error.

ldapModify

An APL command used for modifying an entry by adding, replacing or deleting its attributes.

```
string ldapModify
   ( any  identifier ,
     string  name ,
     string  operation ,
     list<string>  attributeValue  ) //Optional
```

Parameter	Description
identifier	The connection identifier returned from `ldapCreate`
name	The name of the entry to modify
operation	The operation can be either *ADD* , *REMOVE* or *REPLACE* .
attributeValue	The list of new attribute values. If a *REMOVE* operation is requested, the matching values are deleted.
Returns	In the event of an error, a message from LDAP server will be returned otherwise null.

ldapSearch

Performs a search in a LDAP server based on a number of arguments.

```
list<ldapResult> ldapSearch
   ( any  identifier ,
     string  name ,
     string  filter ,          //Optional
     list<string>  retAttrs  ) //Optional
```

Parameter	Description
identifier	The connection identifier returned from `ldapCreate`
name	The name of the object to be searched for. This is matched relative to the context specified with the *name* argument in `ldapCreate`. If no object matches the name, an error will be thrown stating there is no such name.
filter	Optional value, specifying the attributes of the requested LDAP objects. This value conforms to RFC 2254. The specified filter will match objects, relative to the *name* parameter.
retAttrs	Optional list of strings that defines what attributes to be returned
Returns	A list of `ldapResult` UDRs, where every entry contains a name, an attribute and a list of values. The name value is the name relative to the target context of the search. If the target context matches the search filter, then the returned name will be an empty string. The attribute is the attribute name and the values are all values for the given attribute. If `ldapSuppressErrors` has been called, then this function returns null if an error is detected during communication with the LDAP server. `ldapGetLastError` should be used to get the error message in this case.

ldapScopeSearch

Performs a search in a specified part of the LDAP server based on a number of arguments.

```
list<ldapResult> ldapScopeSearch
   ( any  identifier ,
     string  scope ,
     string  name ,
     string  filter ,          //Optional
     list<string>  retAttrs  ) //Optional
```

Parameter	Description
identifier	The connection identifier returned from `ldapCreate`
scope	The search scope, can be one of *OBJECT_SCOPE* , *ONELEVEL_SCOPE* or *SUBTREE_SCOPE* . For further information, see your LDAP manual.
name	The name of the object to be searched for. This is matched relative to the context specified with the *name* argument in `ldapCreate`. If no object matches the name, an error will be thrown stating there is no such name.
filter	Optional value, specifying the attributes of the requested LDAP objects. This value conforms to RFC 2254. The specified filter will match objects, relative to the *name* parameter.
retAttrs	Optional list of strings that defines what attributes to be returned
Returns	A list of `ldapResult` UDRs, where every entry contains a name, an attribute and a list of values. The name value is the name relative to the target context of the search. If the target context matches the search filter, then the returned name will be an empty string. The attribute is the attribute name and the values are all values for the given attribute. If `ldapSuppressErrors` has been called, then this function returns null if an error is detected during communication with the LDAP server. `ldapGetLastError` should be used to get the error message in this case.

ldapSetPooling

The command is used to enable/disable connection pooling. Pooling is enabled by default.

```
void ldapSetPooling ( boolean value )
```

Parameter	Description
value	The value should be set to either `true` or `false`.
Returns	The function returns nothing.

ldapSuppressErrors

An APL command used to change the error behavior of the other LDAP functions.

This function only changes the error behavior of the functions `ldapCreate`, `ldapSearch`, and `ldapScopeSearch`. It does not suppress errors for configuration errors (such as the user using invalid input for the function calls).

```
void ldapSuppressErrors
   ( boolean  value  )
```

Parameter	Descriptionn
value	Whether or not errors should be suppressed.
Returns	Nothing

Example

Example - LDAP APL Plugins

```
// Demonstration of LDAP APL plugin functions

// The following code shows the following:
//
// 1) How to open a LDAP connection
// 2) How to add/delete an object to/from LDAP
// 3) How to change objects properties
// 4) How to perform different type of searches

void runDemoCode() {

  any ctx;
  string name = "DEMOUSER";
  string distinguisedName = "cn="+name;
  string error;
  list<string> attributes;
  list<ldapResult> result;

  // ================================================
  // Opens a connection to a LDAP Server
  // ================================================
  ctx = ldapCreate("127.0.0.1", 389, "dc=nodomain",
                   "cn=admin,dc=nodomain", "demo");

  // ================================================
  // Adding/Deleting/Modifying an LDAP Object
  // ================================================

  //
  //Adds a new person object to the LDAP Server
  //
  attributes = listCreate(string);
```

```
listAdd(attributes, "cn: "+name);
listAdd(attributes, "objectClass: top");
listAdd(attributes, "objectClass: person");
listAdd(attributes, "sn: Test");
listAdd(attributes, "description: Test Record");
listAdd(attributes, "userPassword: {crypt}rkM1sTbxrERtE");

error = ldapAdd(ctx, distinguisedName, attributes);
if( error != null ) {
debug("Error creating user: "+error);
}

//
//Changes description for the created person object
//
attributes = listCreate(string);
listAdd(attributes, "description: A demo user");

error = ldapModify(ctx, distinguisedName, "REPLACE",
                   attributes);
if( error != null ) {
      debug("Error changing user description: "+error);
  }

//
//Removes user description from our person object
//

attributes = listCreate(string);
listAdd(attributes, "description: A demo user");

error = ldapModify(ctx, distinguisedName, "REMOVE",
                   attributes);
if( error != null ) {
debug("Error removing user description: "+error);
}

//
//Adds user description to our person object
//
attributes = listCreate(string);
listAdd(attributes, "description: a description line");

error = ldapModify(ctx, distinguisedName, "ADD",
                   attributes);
if( error != null ) {
debug("Error adding user description: "+error);
}

//
// Deletes our person object from the LDAP Server
//
error = ldapDelete(ctx, distinguisedName);
if( error != null ) {
debug("Error deleting user: "+error);
}

// ================================================
// Executing different searches against the LDAP
// ================================================

// For demo purposes the LDAP tree should contain
// two admin objects (a tree containing a backup
// admin object).

// Setup a filter for the search tests.
// We are interested only in the attribute 'cn'.
//
list<string> attrFilter = listCreate(string);
listAdd(attrFilter, "cn");

//
//Executes a normal search, should return size=1
//
result = ldapSearch(ctx, "dc=test", "cn=admin", attrFilter);
debug("* Normal search");
```

```
  debug("result size="+listSize(result));
  printResultEntries(result);

  //
  //Scoped Search Types
  //
  //This search uses OBJECT scope, and should return size=1
  // Explanation:
  // The search only searches for a specific object, and the
  // 'cn' attribute is returned for that object.
  //
  result = ldapScopeSearch(ctx, "OBJECT_SCOPE", "dc=test",
                           "cn=admin", attrFilter);
  debug("* Object Scope");
  debug("result size="+listSize(result));
  printResultEntries(result);

  //
  //This search uses ONELEVEL scope, and should return size=1.
  //
  // Explanation:
  // The search executes a search on a unique level. Two objects
  // with the same name can not exists on the same level.
  //
  result = ldapScopeSearch(ctx, "ONELEVEL_SCOPE", "dc=test",
                           "cn=admin", attrFilter);
  debug("* Onelevel Scope");
  debug("result size="+listSize(result));
  printResultEntries(result);

  //
  //This search uses SUBTREE scope, and should return size=2.
  //
  // Explanation:
  // The search execute a search on sublevels, and return a
  // match for every admin object found (and we had two).
  //
  result = ldapScopeSearch(ctx, "SUBTREE_SCOPE", "dc=test",
                           "cn=admin", attrFilter);
  debug("* Subtree Scope");
  debug("result size="+listSize(result));
  printResultEntries(result);
}

//
// Help method that prints out the result of a ldapSearch
//

void printResultEntries(list <ldapResult> result)
{
  int resultIdx = 0;
  int resultCount = 0;
  int resultSize = listSize(result);
  while (resultCount < resultSize) {
      ldapResult entry = listGet(result, resultCount);
      debug("Name: " + entry.name);
      debug("Attribute: " + entry.attribute);
      int valueCount = 0;
      int valueSize = listSize(entry.values);
      while (valueCount < valueSize){
           debug("Value: " + listGet(entry.values, valueCount));
           valueCount = valueCount + 1;
      }
      resultCount = resultCount + 1;
    }
}
```

23. Log and Notification Functions

The following functions are used for debugging APL code, or logging user defined messages and events.

debug

Prints the supplied argument to the output target specified in the **Execution** tab of the **Workflow Properties** dialog. Valid options are **Event** or **File** . If **File** is selected, the debug is saved in the temporary directory as stated in the system property `pico.tmpdir`. The file is to be called `debug/<workflow name>`. Each time the workflow is activated resulting in new debug information being written, the existing file is overwritten. If **Event** is selected, the output is shown in the Workflow Monitor.

```
void debug( arg )
```

Parameter	Description
arg	Argument to write to debug output. Could be any type. Note that printing a UDR type will dump all the field values, which may be a large amount of data. Similarly, the debug output for a table or list type may be very large. There is a special case if *arg* is a bytearray. In this case, the output string will be the hex dump returned from the `baHexDump` built-in function. For all other variable types, the output is the direct string conversion, meaning `debug` (*arg*) is the same as `debug`((string) *arg*).
Returns	Nothing

dispatchAlarmValue

The function makes it possible to detect alarm situations based on workflow behavior. It dispatches a user defined `<value>` with a user defined *valueName* from the workflow. The *valueName* used must be defined using Alarm Detection. For further information, see 2.2 Alarm Detection in the Desktop user's guide.

```
void dispatchAlarmValue(string <"valueName">, long value)
```

Parameter	Description
"valueName"	The workflow alarm value name, as defined in the Alarm Detection Editor
value	Any value to be associated with the name
Returns	Nothing

Example - Using dispatchAlarmValue

The following code example displays a situation and syntax useful for the `dispatchAlarmValue`.

```
consume {
 if ( timeToPay ) {
 udrRoute(chargingUdr, "to_billing");
 //Enable for 'amount out of range' Alarm Detection
 dispatchAlarmValue("chargingAmount", chargingUdr.amount);
 }
}
```

dispatchEvent

A user can define a customized event type. This is done using an event UDR, optionally extended with user defined fields. This event UDR can be populated with any information by utilizing APL code, and then be sent on, using the `dispatchEvent` function, to be caught by the Event Notification. For further information about Event Notification, see 4. Event Notifications in the Desktop user's guide.

```
void dispatchEvent( UltraEvent  eventUDR )
```

Parameter	Description
eventUDR	The name of the event UDR
Returns	Nothing

dispatchMessage

This method is used to produce user defined messages associated to predefined Event Categories. For further information about the Event Notification editor, see 4. Event Notifications in the Desktop user's guide. For instance, an Event Category could be named 'DISASTER', and be configured to send an email to the System Administrator. Then an APL agent could send a detailed description with the `dispatchMessage` function whenever this error situation is detected.

```
void dispatchMessage
 ( string  string ,
 string  <Event Category>  )
```

Parameter	Description
string	Value/message to append to the Event Category
<Event Category>	Name of a user defined event as declared in the Event Notification Editor. This event must be defined in the Event Notification Editor in order for the APL code to compile.
Returns	Nothing

log*

Logs a message string to the System Log of type error, warning or information. The entry will fall under the Workflow category where workflow name will be the name of the current workflow and agent name will be the name of the agent logging the message.

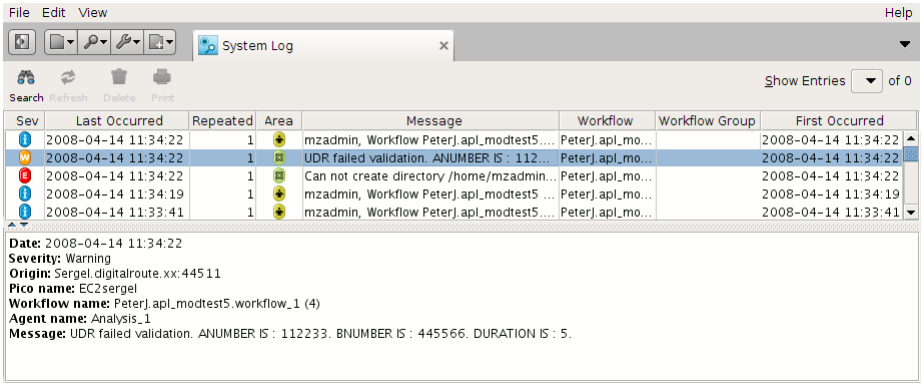
```
void logError
 ( string  message ,
 string  parameterName_n , // Optional
 string|int  parameterValue_n , // Optional
 ... )

void logInformation
 ( string  message ,
 string  parameterName_n , // Optional
 string|int  parameterValue_n , // Optional
 ... )

void logWarning
 ( string  message ,
 string  parameterName_n , // Optional
 string|int  parameterValue_n , // Optional
 ... )
```

Parameter	Description
message	A main message appearing in the log
parameterName_n	Name of an optional parameter. If declared, *parameterValue_n* must be declared as well.
parameterValue_n	Value of an optional parameter. If declared, *parameterName_n* must be declared as well.
Returns	Nothing

Example - Using logWarning

The following code example logs a warning message, which when displayed in the System Log will look like the following figure:

```
logWarning( "UDR failed validation",
 "ANUMBER IS ", input.anum,
 "BNUMBER IS ", input.bnum,
 "DURATION IS ", input.duration);
```



System Log inspection

log.*

These functions invokes logging with log4j. For information about how to configure the logging, e g to set the log level, see the System Administrator's Guide.

```
void log.fatal
(any message,
any tag ) //optional

void log.error
(any message,
any tag ) //optional

void log.warn
(any message,
any tag ) //optional

void log.info
(any message,
any tag ) //optional

void log.debug
(any message,
any tag ) //optional

void log.trace
(any message,
any tag ) //optional
```

Parameter	Description
message	A value that will be appear in the `message` field in the log output This parameter will be ignored if cannot be typecasted to a primitive data type e g string or int.
tag	Objects(s) that will appear in the `tag` field in the log output.
Returns	Nothing

Example - Using log.debug

```
consume {
   log.debug("In consume.");
  list<int> rcDebug =listCreate(int);
 int rc=0;
 listAdd(rcDebug,rc);
 rc=1;
 listAdd(rcDebug,rc);
 log.debug(rc,rcDebug);
 }
```

mailNotify

Sends an email to a configured recipient. In order to operate, the system must have an email remitter and an SMTP mail server defined. For further information,see 2.6.4 Platform Properties in the System Administrator's Guide.

Warning!

If used within the `consume` block, make sure that conditional expressions guarantees that this function does not get called for each UDR.

```
void mailNotify
 ( string  address ,
 string  subject ,
 string  message ,
 string  sender ,  //Optional
 list<string>  attachment  ) //Optional
```

Parameter	Description
`address`	Email address to the recipient on the form: "user@xxx.com"
`subject`	Text ending up as the email subject
`message`	Message string ending up as the body of the email
`sender`	Name or address of the sender of the email **Optional** This field will remain optional only when the attachment field is not populated. Once attachment is populated, the sender field will be a mandatory field.
`attachment`	A list that will contain one or many attachments to be sent with the email. Each list entry will be the full directory path of the attachment. **Example** "/home/admin/attachments/word.txt"
Returns	Nothing

24. MIM Functions

The MIM functions allows the agent to publish its own MIM values or to read MIM values from other agents.

> **Note!**
>
> The MIM functions cannot be used within global APL scripts, that is in code saved in the APL Code Editor.

mimGet

Returns the value of a MIM resource available in the workflow.

```
any mimGet
 ( string  category ,
 string  mimResourceName  );
```

Parameter	Description
category	Name of the agent or workflow owning the MIM resource
mimResourceName	Name of the MIM resource whose value is to be returned
Returns	MIM value as any type. The result *always* needs to be type casted or else compilation fails

Example - Using mimGet

```
mimGet("Workflow", "Batch Count"); // Retrieving a MIM resource owned by the workflow.
mimGet("Disk_1", "Source File Count"); // Retrieving a MIM resource owned by the agent "Disk_1".
```

mimIsNull

Evaluates if a MIM resource is defined. Available to evaluate MIMs of non-object types, such as `int`.

```
boolean mimIsNull
 ( string  category ,
 string  mimResourceName );
```

Parameter	Description
category	Name of the agent or workflow owning the MIM resource.
mimResourceName	Name of the MIM resource to look for.
Returns	`true` or `false`.

mimPublish

Publishes a MIM resource. The method call can only be used outside a function block.

```
void mimPublish
 (  assigned ,
 string  name ,
  mimType );
```

Parameter	Description
assigned	Keyword that identifies when the MIM resource is assigned a value. Can be one of: • batch - Value is assigned from the `consume` block. • header - Value is assigned in the `beginBatch` block. • trailer - Value is assigned in the `endBatch` block. • global - Value is assigned from any block.
name	MIM resource name. Must be a string constant.
mimType	Data type of MIM resource
Returns	Nothing

Note!

Real-time workflows can only publish MIMs of type global.

mimSet

Assigns a value to a user defined MIM resource. The function may be called at any time. Note that it is the responsibility of the user to make sure the MIM value is available in accordance with the specified assigned type (refer to the section above, mimPublish).

```
void mimSet
 ( string  name ,
 any  value );
```

Parameter	Description
name	Name of the MIM resource to assign a value
value	Value to assign - must match the published MIM type
Returns	Nothing

MIM Example

Example - MIM

Note that `mimSet` is made from a `drain` block. This to make sure the MIM exists when called from any other agent. The calls will be made at `endBatch` (for trailer MIMs), which always occurs after `drain`.

```
mimPublish(trailer, "Total duration", int);
int duration;

beginBatch {
 duration = 0;
}

consume {
 duration = duration + input.CallDuration;
}

drain {
 mimSet( "Total duration", duration );
}
```

25. PKCS7 Functions

These functions support the signing of data using CMS signature. See the JDK product documentation for information about using keytool in different scenarios.

loadPKCS7Certificate

Loads the stated PKCS7 certificate.

```
void loadPKCS7Certificate
 ( string keyStorePath,
 string keyAlias,
 string keyStorePassword,
 string signatureAlgorithm )
```

Parameter	Description
keyStorePath	The path to the Java keystore file to be used
keyAlias	The alias for the certificate
keyStorePassword	The password for the keystore
signatureAlgorithm	The algorithm to be used for signing
Returns	Nothing

Hint!

It is recommended that you use this function in the `initialize` function block.

signPKCS7

Signs a bytearray using the previously loaded PKCS7 certificate.

```
bytearray signPKCS7  ( bytearray content)
```

Parameter	Description
content	A byte array of the content to be signed with the certiticate loaded by the `loadPKCS7Certificate` function.
Returns	A bytearray with the signed content.

Example - Using signPKCS7

```
initialize {
        loadPKCS7Certificate("/etc/keystores/keystore.jks",
         "certificateA", "keystoreAndAliasPassword", "SHA1withRSA")
}
consume {
        bytearray baToSign;
        strToBA(baToSign, "Hello World!");
        input.response = signPKCS7(baToSign);
        udrRoute(input)
}
```

26. Random Number Generation Functions

Use the functions below to generate random integer values.

randomInt

Returns a pseudorandom, uniformly distributed int value from a random number generator initiated at workflow start.

The values returned will be pseudorandomly generated and returned. All 2^31 possible int values are produced with (approximately) equal probability.

This function uses the standard Java Random class. Refer to JDK documentation for exact details on the algorithm used for generating pseudorandom values.

```
int randomInt(int bound)
```

Parameter	Description
bound	The upper bound (exclusive). Must be positive.
Returns	A random integer in the range 0 to bound-1

randomLong

Returns a pseudorandom, uniformly distributed long value from a random number generator initiated at workflow start.

The values returned will be pseudorandomly generated and returned. The algorithm uses a seed with only 48 bits, which means that not all possible long values will be returned.

This function uses the standard Java Random class. Refer to JDK documentation for exact details on the algorithm used for generating pseudorandom values.

```
long randomLong()
```

Parameter	Description
Returns	A random long

27. Shell Script Execution Functions

The `scriptExec` function runs a shell script on the EC/ECSA where the workflow runs and then returns a UDR that contains the exit value. The mzadmin user must have execute permissions on the specified script file.

```
ScriptResult scriptExec
 ( string path,
 string argument [,...]) //Optional
```

Parameter	Description
`path`	The absolute path of the script. When the directory is set in the path variable of the mzadmin user on the host, only the filename is required.
`argument`	Command line argument(s). The total size of the arguments must not exceed 50kB.
Returns	A ScriptResult UDR containing the following fields: • ExitValue (int) - The return code from the shell script. • MZError (string) - Error description from MediationZone (if any). • OriginalData (bytearray) - This field is always empty. • StdError (string) - Script output on StdError. • StdOut (string) - Script output on StdOut. • hasMZError (boolean) - True if script failed to execute, otherwise false.

Note!

When the output to stdout or stderr is large, the execution of the script may terminate before the data is fully streamed. This will interrupt the streams and result in partial outputs in the ScriptResult UDR. You may add one more sleep periods in the script to ensure that the output is complete before the execution is terminated.

Example - Using scriptExec

```
initialize {
    ScriptResult rc;
    rc = scriptExec("/home/mzadmin/scripts/script1.sh", "arg1","arg2");
    debug("rc= "+ rc);
}
consume {
    udrRoute(input);
}
```

Substrings separated by spaces are interpreted as separate arguments by the `scriptExec` function. If the arguments contain spaces you may substitute these in your APL code and in the script.

Example - Substituting spaces

Substitute underscores in arguments with spaces in bash script:

```
for ARG in $*
do
        echo ${ARG//_/ }
done
```

Substitute spaces in arguments with underscore in APL:

```
initialize {
    ScriptResult rc;
    rc = scriptExec("/home/mzadmin/scripts/script1.sh", "This_argument_contains_spaces");
    debug(rc.StdOut);
}
```

Output:

```
This argument contains spaces
```

28. Signature Functions

Use the Signature functions to generate digital signatures for data.

signPrepare

This function is called to get a key from a Java keystore file.

```
any signPrepare
    ( string file,
      string keystoreType,
      string password)
```

Parameter	Description
file	The file that contains the keystore dat
keystoreType	The keystore type e g . JKS, PKCS8, PKCS12 or CA_CERTIFICATES_PATH
password	The keystore password
Returns	A private key object

signData

This function calculates the hash value of a bytearray, using the specified algorithm, and signs the result.

```
bytearray signData
    ( any key,
      bytearray data,
      string algorithm)
```

Parameter	Description
key	A private key object
data	The data to be signed
algorithm	The signature algorithm e g SHA1withDSA, SHA1withRSA or SHA256withRSA.
Returns	A bytearray containing the digital signature

signHmac

This function calculates the hash value of a key retrieved with the signPrepare function or a bytearray constituting a key, using the specified algorithm, and signs the result.

```
bytearray signData
    ( any key,
      bytearray data,
      string algorithm)
```

Parameter	Description
key	A private key object
data	The data to be signed
algorithm	The signature algorithm e g Hmac-SHA256 or Hmac-SHA512.
Returns	A bytearray containing the digital signature

signHmacSHA256

This function calculates the hash value of a key retrieved with the signPrepare function or a bytearray constituting a key, using Hmac-SHA256, and signs the result.

```
bytearray signData
    ( any key,
      bytearray data)
```

Parameter	Description
key	A private key object
data	The data to be signed
Returns	A bytearray containing the digital signature

signHmacSHA512

This function calculates the hash value of a key retrieved with the signPrepare function or a bytearray constituting a key, using Hmac-SHA512, and signs the result.

```
bytearray signData
    ( any key,
      bytearray data)
```

Parameter	Description
key	A private key object
data	The data to be signed
Returns	A bytearray containing the digital signature

29. Workflow Functions

Use the functions below to control the workflow state from APL or to retrieve values from the workflow table.

abort

Stops the workflow, and logs a user defined message to the System Log.

```
void abort ( string  message )
```

Parameter	Description
message	A message (of type error), which is sent to the System Log when the workflow is aborted
Returns	Nothing

> **Note!**
>
> The only APL code to be executed after an abort call is the `deinitialize` function block.

cancelBatch

Emits a Cancel Batch that aborts the processing of the current batch and possibly continues with the next file (depending on the Workflow Configuration).

```
void cancelBatch
 ( string  message ,
 drudr  errorUDR  ) //Optional
```

Parameter	Description
message	A message (of type error), which is logged to the System Log when the batch is cancelled
errorUDR	It is possible to send an error UDR containing any useful information to the ECS where the batch is sent (if configured to do so). Note that an error UDR may be defined from the Workflow Properties dialog as well. In this case, the APL code overrules the window configuration. This parameter is optional.
Returns	Nothing

dynamicFieldGet

Retrieves the stated dynamic field from the workflow table. The returned value can either be a boolean, an integer, or a string. The fields are configured in the **Dynamic Fields** tab in the Workflow Properties. See 3.1.8.2 Dynamic Fields Tab in the Desktop User's Guide for further information.

> **Note!**
>
> The APL code will not validate unless the dynamic field(s) has been configured in Workflow Properties.

```
boolean/int/string dynamicFieldGet
    ( string category,
      string name )
```

Parameter	Description
`category`	The dynamic field's category
`name`	The name of the dynamic field
`Returns`	The value of the selected dynamic field

hintEndBatch

Large input files can be split into smaller output files using `hintEndBatch`. The function will send an End Batch message to the other agents in the workflow, possibly causing a split of the current batch being processed. How each agent acts upon such a request can be found out in the respective user's guide; in the Transaction Behavior section. If the workflow does not contain any agent capable of acting upon the request, the workflow will abort.

> **Note!**
>
> The `hintEndBatch`() function is only supported for workflows containing one of the following:
>
> - Database Collection agent
> - Disk Collection agent
> - FTP Collection agent

```
void hintEndBatch()
```

Parameter	Description
Returns	Nothing

> **Note!**
>
> A split batch cannot be sent to ECS. Calling `cancelBatch` after `hintEndBatch`, will result in workflow abort. However, the original batch can be sent to ECS to make sure to evaluate if the batch will be canceled before it is split.

sleep

Puts the current thread to sleep a given number of milliseconds before resuming execution. This is mostly used for simulation, for instance traffic shaping or limiting number of calls towards an external system.

```
void sleep
 ( long  milliseconds  )
```

Parameter	Description
`milliseconds`	Number of milliseconds the current thread will sleep before resuming execution.
Returns	Nothing

wfStop

Stops a running workflow. *`wfStop`* produces a single stop signal and does not wait for the workflow to stop. If it succeeds in stopping the workflow, the stop is registered in the System Log.

```
string wfStop ( string  wfName , boolean  immediate  )
```

Parameter	Description
wfName	The name of the workflow that should be stopped. **Note!** This command can be used from within the same workflow that should be stopped.
immediate	True: Stops currently handled batch. False: Runs through currently handled batch and then stops. Note: In a realtime workflow this *immediate* flag has no significance.
Returns	A string. If the command is successful the value is `null`. Otherwise, a text message is returned.

Note!

In ECSAs, the `wfStop` command can only stop workflows running on the same container host where the command is applied. Otherwise, the following message is returned: "A workflow by the name <wfName> has not been found."

30. Azure Functions

The Azure OAuth Token functions include:

- acquireAzureOAuthToken
- acquireAzureOAuthTokenCustomAuthority

acquireAzureOAuthToken

Contacts Azure to set up an OAuth Token based on the stated parameters.

```
AzureOAuthToken acquireAzureOAuthToken(
string tenantID,
string resourceID,
string clientID,
string certificate,
string certifcatePassword )
```

Parameters

Parameter	Description
tenantID	The Tenant ID is a GUID for your instance, or domain.
resourceID	The Resource ID is the App ID uri of the Azure application that you want to receive an OAuth Token for.
clientID	Also known as an Application ID, a GUID that identifies the Azure application that is requesting the OAuth Token.
certificate	The full path and filename to the certificate or truststore. Must be set up on the EC that the workflow is running on. Only a PFX formatted certificate or a PKCS#12 formatted JKS truststore can be used.
certificatePassword	The password to the certificate or truststore.
Returns	An AzureOAuthToken UDR, or an exception if the parameters/credentials were incorrect.

Example

AzureOAuth.AzureOAuthToken = AzureOAuth.acquireAzureOAuthToken("733f2c27-a0ca-45e1-a3b7-014bd4518ea3", "https://storage.azure.com/", "69bca157-2362-4ca7-adf3-6a26f6bd842f", "/opt/azure/certificate/appaccesscert.pfx", "verysecretpass");

acquireAzureOAuthTokenCustomAuthority

Contacts Azure to set up an OAuth Token based on the stated parameters with `authorityHost`.

```
AzureOAuthToken acquireAzureOAuthToken(
string tenantID,
string resourceID,
string clientID,
string certificate,
string certifcatePassword,
string authorityHost )
```

Parameters

Parameter	Description
tenantID	The Tenant ID is a GUID for your instance, or domain.
resourceID	The Resource ID is the App ID uri of the Azure application that you want to receive an OAuth Token for.
clientID	Also known as an Application ID, a GUID that identifies the Azure application that is requesting the OAuth Token.
certificate	The full path and filename to the certificate or truststore. Must be set up on the EC that the workflow is running on. Only a PFX formatted certificate or a PKCS#12 formatted JKS truststore can be used.
certificatePassword	The password to the certificate or truststore.
authorityHost	The URL for the directory that Microsoft Authentication Library will request tokens from.
Returns	An AzureOAuthToken UDR, or an exception if the parameters/credentials were incorrect.

Example

AzureOAuth.AzureOAuthToken = AzureOAuth.acquireAzureOAuthToken("733f2c27-a0ca-45e1-a3b7-014bd4518ea3", "https://storage.azure.com/", "69bca157-2362-4ca7-adf3-6a26f6bd842f", "/opt/azure/certificate/appaccesscert.pfx", "verysecretpass", "https://login.microsoftonline.com/");

30.1 Azure OAuth Token UDR

The Azure OAuth Token APL Plugin produces one type of UDR; **AzureOAuthToken**.

AzureOAuthToken

This UDR is used for describing a specific attachment.

Field	Description
`Expires (date)`	At which time the generated token becomes invalid.
`Token (string)`	The generated token in string format.
`OriginalData (bytearray)`	The original data in bytearray format.