CompatibleOne

Cordscript Reference
Version 1.0

Iain James Marshall

12/12/2012

This document provides the complete reference to the instructions and pseudo machine code of Cordscript.

Table of Contents

Table of Contents 2
Introduction 4
Overview 5
Operation 5
Language 6
- Comments 9
- Values 9
- Variables 10
- Instructions 11
  - INCLUDE 11
  - IF 11
  - ELSE 11
  - WHILE 12
  - FOR 12
  - TRY 13
  - CATCH 13
  - FOREACH 13
  - FORBOTH 14
  - SWITCH 14
  - CASE 14
  - DEFAULT 14
  - BREAK 14
  - CONTINUE 15
  - RETURN 15
  - THROW 15
- Functions 15
- Categories 16
- Instances 17
- Methods 18
  - POST, CREATE, NEW 18
  - GET, RETRIEVE 18
  - PUT, UPDATE 19
  - DELETE 19

LINK 19
DATE 19
DISPLAY 19
COUNT 20
LENGTH 20
DEBUG 20
ROUND 20
WAIT 20
Actions 20
Examples 22
Category List 22
Restart Services 22
Transaction Processing 23
Machine Code 24
General 24
NOP 24
PUSH value 24
POP 24
DUP 24
EVAL value, value, value 24
Arithmetical 24
CAT value 24
AND value 25
OR value 25
XOR value 25
ADD value 25
SUB value 26
MUL value 26
DIV value 26
MOD value 27
Control Transfer 27
JMP instruction 27
ENTER instruction 27
LEAVE instruction 27

CALL instruction 27
RET 27
RETVAL 27
EACH array, variable, instruction 28
BOTH array, name, value, instruction 28
JEQ value, instruction 28
JLS value, instruction 28
JLE value, instruction 28
JGR value, instruction 28
JGE value, instruction 28
JNE value, instruction 29
Conclusions 30
References 31
Annex 1 32
IF ELSE 32
FOR 32
FOREACH 33
FORBOTH 33
WHILE 33
SWITCH CASE DEFAULT 34
TRY CATCH 34
Annex 2 36
STACK 36
DATA 36
CODE 36

Introduction

The Accords Platform, the cloud management and brokerage platform resulting from the CompatibleOne project, comprises a very large number of discrete components each operating behind their own individual OCCI interface and communicating as OCCI/REST/HTTP clients and servers using Transport Layer Security over the Transport Control Protocol. The operation of the Accords Platform is described in detail in the “project deliverable” document, “Accords Platform Version 1.3”, completed in December 2012. The details of the API of the various components, comprising the platform, are described in the “project deliverable” document, “CORDS Technical Reference Version 2.12”, also completed in December 2012. This current document describes the

Cordscript language which may be used to describe additional behavior for the articulation of the standard components when tailored for use in real world use case scenarios.

Overview

Many attributes of the component categories of the CORDS model already make provision for the definition and interpretation of Cordscript instructions. Examples of use of Cordscript can be clearly seen in the case of the “action” expressions of the “configuration”, “release” and “interface” categories. Default values for attributes defined in manifest schema documents also make use of Cordscript describing post processing behavior to be performed by the Accords Platform Parser tools during the parsing and “compilation” operations. Other categories allow use of Cordscript for the description of behavior that must occur when certain conditions arise. A case of this can be seen in the management of service level objectives, by instances of the monitoring “control” category, which makes use of Cordscript for the description of behavior, required for the business values of SLA guarantee terms, and interpreted when penalties and rewards are to be incurred in response to real operational conditions that are subsequently encountered.

The subset of Cordscript instructions that were originally defined to satisfy for the particular needs of the post configuration operations performed during deployment of resources has been extended, as described in this document, to provide a fully blown computing scripted language that is to be used for the control and animation of platform components in operational scenarios.

Operation

The cordscript compiler interpreter has been integrated in the Accords Platform Command Line tool known as “co-command”. This allows for ease of development of cordscript programs allowing real time compilation and testing to be performed.

```
co-command [ options ] run <scriptfilename>
```

For more information relating to the standard options of this tool please refer to the Accords Platform reference [2].

Script prepared in this way can then be established as the values of the attributes of the required category instances to be executed when required during standard platform operation. Cordscript sequences can also be defined in the default attribute value definitions of manifest description schema for the specification of post processing to be performed by the standard Accords Platform Parser during manifest processing.

Language

The Cordscript language provides constructions for the description of the following fundamental programing concepts:

- **Comments**

  Comments are an important part of any program since they provide the means of conveying important background information concerning particular sections and instructions.

- **Values**

  Values may be specified to represent either strings of numerical digits and special symbols, strings of alpha numerical characters or sequences conforming to the Cordscript Object Notation syntax for the description of complex objects and arrays.

- **Variables**

  Variables may be considered as persistent, modifiable, named values. They are active for the duration of the context within they are declared.

- **Operators**

  Cordscript offers a comprehensive range of arithmetic operators allowing for addition, subtraction, multiplication, division and modulus arithmetic to be performed primarily on integral and floating point numerical types.

  The standard logical bitwise operators are also provided for use with integral types allowing AND, NOT, OR and XOR operations to be performed.

  In addition to these basic operators more complex object operators are also provided allowing concatenation and division for string types and allowing concatenation and member selection for array and structure types.

  Examples:

  ```
  $I = 2;

  $I = $I + 1;

  $I == 3
  ```

  For an integer variable $I with an initial value of 2, the result of the addition of 1 will be 3.

  ```
  $F = 2.0;

  $F = 2 * 3.78;

  $F == 7.56
  ```

  For an float variable $F with an initial value of 2.0, the result of the multiplication by 3.78 will be 7.56

```
$S = "String";

$S = $S # "Example";

$S == "StringExample";
```

For a string variable $S with an initial value of "String", the result of the concatenation with "Example" will be "StringExample".

```
$A = [];

$A = $A + $I;

$A = $A + $F;

$A = $A + $S;

$A == ["3","5.46","StringExample"];
```

For an array variable $A with an initial empty value of [], the result of the successive addition of the preceding results will be ["3","5.46","StringExample"].

```
$S = "a.b.c.d";

$A = $S / ".";

$A == ["a","b","c","d"]
```

For a string variable $S with an initial value of "a.b.c.d", the result of the division of the string at each occurrence of the "." character will result with the array $A containing ["a","b","c","d"].

```
$B = $A[1];

$B == "b";
```

The member operator and index value of 1 applied to the array $A resulting from the previous string division will return the value of "b", the 1st element in the array.

```
$S = {"a";"1","b":"2","c":"3"};

$B = $S["b"];

$B == "2";
```

The member operator and key value of "b" applied to the structure defined for the variable $S will return the value of "2", the value of the tuple named "b".

```
$S = $A * "/";

$S == "a/b/c/d"
```

The above array, $A, when combined using the multiplication operator and the string value of "/" will result in the string variable $S containing the concatenation of the array elements separated by the "/" character.

NB: The combination of certain operators will give unexpected results due to the inbuilt reverse polish notation effect stack depth oriented operation of the cordscript parser and interpreter.

- **Conditional Operators**

  Cordscript provides the standard EQUAL, NOT EQUAL, GREATER THAN, LESS THAN, GREATER OR EQUAL and LESS THAN OR EQUAL comparison operators for the construction of conditional expressions used in IF, FOR and WHILE instructions.

  Examples:

```
If ( $a == 1 ) {       "true if variable $a is equal to 1".display();      }

If ( $a != 1 ) {       "true if variable $a is not equal to 1".display();  }

If ( $a >= 1 ) {       "true if variable $a is greater than or equal to 1".display(); }

If ( $a <= 1 ) {       "true if variable $a is less than or equal to 1".display(); }

If ( $a > 1 ) {        "true if variable $a is greater than 1".display();  }

If ( $a < 1 ) {        "true if variable $a is less than 1".display();     }
```

- **Instructions**

  Instructions are the collection of operations defined for Cordscript and which allow the use of variables and values for the construction of arithmetical and conditional transfer of control statements.

- **Functions**

  Functions are defined, using Cordscript, regrouping named collections of variables and instructions intended to fulfill a particular purpose. Functions may receive invocation parameters and may return results.

- **Categories**

  Categories are the work unit descriptions that will be animated using Cordscript to achieve the required platform behavior. Categories are synonymous with classes of which instances may be created, used and destroyed.

- **Instances**

  Instances of categories are created not only for the storage of descriptive data and but also for the representation and management of both input and results for all operations performed by the platform.

- **Methods**

  Methods refer to the HTTP verbs through which the four fundamental CRUD (create, retrieve, update and delete) may be performed on instances of categories.

- **Actions**

  Actions refer to the OCCI verbs defined for each particular category allowing additional operations to be performed on instances of categories.

Each of the above concepts will now be described in detail showing precise examples of their usage.

Comments

The two standard constructions are offered for delimiting comments.

The slash and star combination signals the start of a comment with the star slash combination indicating the end.

Example:

```
/* enclosed comment using slash start and star and slash */

Instructions;
```

The standard double slash character combination is used to signal a comment that will continue until the next newline sequence.

Example:

```
// comment till end of line using the double slash

Instructions;
```

Values

All data required for use in the construction of Cordscript instructions will be in the form of constant values or expressions. These may be numerical values, integer or float, string values or complex object description values, depending on their initial characters.

Numeric constants will normally start with a digit, or a sign, and will be composed of any number of digits and at most one decimal point. The presence of a decimal point will identify a floating point value otherwise an integer value is assumed and silent clipping may occur. Hexadecimal numeric constants are defined as in the C language by a 0x prefix to the string of hexadecimal digits.

Examples:

```
1

12345

+123

-27

45.67

-2000.55

0xABCD

0xabcd
```

String constants may start with either one of the two quote opening characters, the single quote or the double quote, and must be terminated by the identical quote character as at the start. This allows encapsulation of either of the other quote characters inside a quoted string. String constants may be defined from zero length up to the maximum length of one single contiguous sequence of memory for the machine in question. String constants cannot contain the null (zero) character since this is used for the termination of the sequence of data bytes composing the string.

Examples:

```
“hello world”

‘hello world’

“<element name=’value’/>”

‘<element name=”value”/>’
```

Complex objects are defined using Cordscript Object Notation, with an identical syntax as Javascript Object Notation, where two special characters are used to delimit the description of the two basic value types, the complex object and the array. These types may be freely intermingled to create arrays of objects and objects containing arrays, as required for the description of the data structure.

Example Object or Structure:

```
$v = { “name” : “test”, “status” : “ok” }
```

Here we can see that the complex object description string is delimited by the left and right curly braces, between which all other displayable characters are permitted and collected to create the complex string object.

Example Array:

```
$v = [ “red”, “green”, “blue” ];
```

Here we can see a simple string array containing the names of the three primary colors.

Combined Example:

```
$v = [ { “name” : “true”, “value” : 1 }, { “name” : “false”, “value” : 0 } ];
```

In this more complex example we can see an array where each element is a complex data structure each defining values for their own name and value members.

This technique, used for the definition of complex objects, is of particular importance for the description of category data structures that will be input to and output from the various category methods and instance actions of the standard OCCI client interface of the Cordscript execution environment.

Variables

Variables are values for which a name has been defined and which may be modified during the course of execution of the Cordscript instructions. Variable names are identified by the preceding ‘$’ character followed by any combination and number of alphanumeric characters and the underscore character. Command line parameters passed to the script at run time will be available through the special variable names of “$1”, “$2”, “$3” and so on, and may be length tested to check for their validity.

Variables are defined when first used and will remain in scope until the block in which they were defined goes out of scope.

Variables are always passed by reference to instructions and their content can therefore be modified as if they were pointers to the original data source.

Examples:

```
$v = 1;

$name = "john";

$complex_name = "another variable";
```

Instructions

The following instructions are defined for the Cordscript interpreter allowing conditional transfer of control and iterations to be specified.

INCLUDE

This instruction includes the contents of a Cordscript file or stream at the current location. The file will be processed as if it were an integral part of the current file or stream and global functions and variables encountered will be added to the enclosing parent execution context.

IF

This instruction allows the selection of conditional sections of code depending on the result of the evaluation of conditional expressions. When the conditional expression evaluates to false then control will continue after the closing brace of the conditional block. These expressions may be fully nested to any degree allowing complex conditions to be expressed.

Example:

```
If ( $v > 1 )

{

        Statements of the conditional block;

        If ( $v > 5 ) { $v.display(); }

}
```

The first line of the above shows a first conditional expression when the variable "v" is tested to be greater than 1. When the condition is true then the statements of the conditional block will be evaluated including the second nested conditional expression.

ELSE

The else instruction can only be used directly after the conditional block of an IF instruction and defines an alternative conditional block for processing as a result of the conditional expression evaluating to false. These IF and ELSE instructions can be freely nested within each other as necessary for the expression of the required conditional logic.

Example:

```
If ( $v > 1 )
```

```
        {
                Statements of the conditional block;
                If ( $v > 5 ) { $v.display(); }
        }
        Else
        {
                Alternative conditional block;
        }
```

In the above example, building from the preceding example of the IF instruction, here we can see the definition of an alternative block of conditional instructions that will be executed when the value of the variable “v” is not greater than 1 as described by the controlling conditional expression of the IF instruction.

WHILE

This instruction allows evaluation of the conditional block to be repeated while the conditional control expression evaluates to true. When the conditional expression is false then control will be transferred to after the conditional block. These instructions can be fully nested to any degree allowing complex iterations to be defined.

Example:

```
        While ( $v > 1 )
        {
                Statements of the conditional block;
                If ( $v > 5 ) { $v.display(); }
                $v = $v - 1;
        }
```

In this example we see a conditional control expression. The condition will be evaluated at the start of each cycle. When the condition is true then the statements of the conditional block will then be executed otherwise control will be passed to the instructions that follow the conditional block. The final line of the conditional block shows the iterator responsible for changing the conditions during each cycle.

FOR

This instruction provides a construction allowing a complex iterator to be defined using specific instructions for the initialization, conditions and iteration control.

```
        For (   initialization;
                Conditional;
                Iteration )
        {       conditional block;      }
```

First the initialization instruction will be evaluated. Then, for each cycle, the conditional expression will be evaluated. When this evaluates to true, then the conditional block will be executed. The iterator will be executed after the conditional block and control will pass back to the conditional expression. If the condition evaluates to false then control will be transferred to the next instruction in sequence after the iterative conditional expression.

Example:

```
For (   $v=1;

        $v < 10;

        $v = $v + 1     )

{       $v.display();  }
```

The preceding example will iterate ten times and will display the value of the variable "v" at each step until it reaches the value of "10". The above example is shown in expanded form for clarity and may be condensed and expressed on one line as can be seen below.

Example:

```
For (   $v=1; $v < 10; $v = $v + 1 ) { $v.display();}
```

TRY

This instruction marks the start of a construction for the management of run tie exceptions and allows for the centralization of exception processing subsequently required for the handling of fault conditions encountered during the execution of the enclosed block. This construction will be complimented by one "catch" instruction responsible for the actual processing of the data delivered by the exception.

Example:

```
Try {   instructions;  }
```

CATCH

This instruction compliments the preceding "try" instruction for the processing of the data delivered by the exception management mechanism. This instruction cannot be used on its own. A variable must be declared for the reception of the exception data. The body of the catch must be provided for the analysis of the exception data.

Example:

```
Catch ( $variable ) { instructions;  }
```

FOREACH

This instruction allows for matrix, complex object and list processing such that each individual element of the source operand will be presented as an output operand on each cycle of the iteration. The iteration will terminate when no more values are available from the source. The composition of the instruction is shown in the following examples.

Example:

```
$source = compute.list();
```

```
Foreach ( $source as $output )       { $output.display();  }
```

In the above case the list of compute category instance identifiers will be retrieved by the first instruction as CSON array element to be displayed one line at a time by the FOREACH instruction.

FORBOTH

This instruction allows for structured, complex object processing such that each individual name and value element pairs of the source operand will be presented as in the two output operands on each cycle of the iteration. The iteration will terminate when no more name and value pairs are available from the source. The composition of the instruction is shown in the following examples.

Example:

```
$source = instance.get();

Forboth ( $source as $name and $value )      { $name.display();    }
```

In the above case the list of category instance attribute values will be retrieved by the first instruction as CSON array element to be displayed one line at a time by the FORBOTH instruction.

SWITCH

This instruction allows construction of a multiple case selection in conjunction with the CASE and DEFAULT instructions that will be described in subsequent sections.

Example:

```
Switch $v

{

Case    1       :

Case    $x      :

Default         :

}
```

This example shows a simple switch where the variable “v” will be tested against an explicit case of “1” and an indirect value through the variable “x”.

CASE

This instruction is used in conjunction with the preceding SWITCH instruction which provides an example of the different types of use that are permitted, either by constant value or by indirect variable values.

DEFAULT

This keyword provides the default conditions for use with a SELECT and CASE construction. As described above.

BREAK

During iterations such as FOR, WHILE and the SELECT instruction, this instruction performs control of transfer out of the iteration loop or selection group.

CONTINUE

During iterations, such as provided by the FOR and WHILE instructions, this instruction allows transfer of control directly to the iteration control condition.

RETURN

This instruction allows explicit return from a Cordscript function, or termination of the current Cordscript execution context. An optional return value may be specified that will be returned for use by the calling context.

Example 1:

```
Function       main() { return("hello world"); }

$v = Main();
```

In this first example the return instruction performs exit from a Cordscript function and returns a value to the calling instruction.

THROW

This instruction raises an explicit exception that delivers exception data that will be caught by the "catch" clause of an enclosing "try" instruction.

Example:

```
Throw( penalty.new() );
```

Functions

Cordscript allows the definition of functions, regrouping variables and instructions, as named entities, that may be invoked from either within a Cordscript context or from an external cordscript enabled environment. Functions may define parameters for the reception of invocation values and may return result values upon completion.

Example 1:

```
Function example( $a, $b ) { $a.display(); $b.display(); return( result"); }
```

Here we see an example function which defines two parameters and returns a string result. The following example of function invocation:

```
$v = example( "hello", "world" ); $v.display();
```

Would produce the following output to the standard output stream:

```
"hello"

"world"

"result"
```

The access scope of functions can be defined as "public" or "private". In the former case invocation of the function is possible from outside of the execution context in which it is defined. When declared private, invocation is only possible from within the same execution context.

```
Public Function NewNode( $name, $access, $scope, $type )
```

```
{

$n = node.create();

$n.update( {"name":$name,"access":$access,"scope":$scope,"type":$type} );

Return ( $n );

}
```

This example shows a public interface function which creates a new instance of the "node" category using the values provided by the invocation parameters and returning the resulting universal and unique category instance identifier.

Categories

The fundamental descriptive element of the Open Cloud Computing Interface OCCI classification system is the "category" often referred to as the "kind". Categories are defined, in OCCI, in terms of collections of attributes and actions. This is synonymous with a standard classification system whereby classes are described in terms of attribute members and method functions. OCCI provides in fact the transposition of a remote object description and method invocation system for use over REST and HTTP. The Accords Platform is entirely described and built on the same principles where by all concepts are implemented as OCCI Category Instances and their management components. The Cordscript language provides a mean by which these underlying OCCI instances and management components may be used in a very simple yet powerful programmatic way.

Methods

Category names may be used directly, in conjunction with the following OCCI methods:

- **POST**

  This method, or one of the synonyms NEW or CREATE, will request the creation of a new instance of the named category. The resulting universal unique identifier of the newly created category instance will be returned to indicate success otherwise an error structure will be returned.

- **GET**

  This method, or its synonym LIST, will request the list of category instances corresponding to the OCCI filters provided as parameters. When no filters are provided the complete collection of category instance identifiers will be returned. An array value will be returned containing the collection of instance identifiers.

- **DELETE**

  This method requests the deletion of all category instances matching the provided OCCI filter. If no filter is provided then all instances of the corresponding category will be deleted.

  No result will be returned by the DELETE method.

OCCI Filters

In the preceding method description, reference was made to the term OCCI Filter. This refers to a collection of OCCI name value pairs that may be used for the selection of records for retrieval or deletion. The same construction is also used for the transmission of attribute names and values for instance creation.

Examples

```
$id = Compute.post(occi.filters);
```

This first example shows creation of an instance of the compute category. The unique identifier will be returned and stored in the variable “id”.

```
$list = Compute.get(occi.filters);
```

In this example an array containing the universal identifiers of the list of compute instances will be returned and stored in the variable “list”.

```
$r = Compute.delete(occi.filters);
```

This example shows how ALL instances of the compute category may be deleted in one instruction.

For further information concerning the collection of categories comprising the Accords Platform please refer to the Cords Technical Reference Version2.12.

Instances

As described above the invocation of the POST method for a category identifier will create an instance or resource of that category. The fully qualified universal unique category instance identifier will be returned and may be used to retrieve, update and delete the contents of the instance or resource. The identifier may also be used for the invocation of actions defined for the particular category.

Methods

- **GET**

  This method, or its synonym RETRIEVE, when applied to a fully qualified category instance identifier, allows retrieval of the corresponding category instance attributes and links. The resulting information will be returned in a CSON structure.

- **PUT**

  This method, or its synonym UPDATE, when applied to a fully qualified category instance identifier, allows the attribute values of the category instance to be updated using the attribute name value pairs provided by the OCCI filter matrix.

- **LINK**

  This method, when applied to a fully qualified category instance identifier, and provided with a when applied to a fully qualified category instance identifier as parameter, will create an OCCI LINK between the subject, as the source, and the parameter as the target.

- **DELETE**

  This method when applied to a fully qualified category instance identifier, allows deletion of the category instance and all associated links.

Examples

```
$v = compute.new({"cores":"1","memory":"1G"});
```

Here we see the creation of a compute instance specifying that the 1 core and 1G of RAM are required.

```
$c = $v.get();
```

This instruction performs retrieval of the contents of the Instance identified by the contents of the variable "v". The result will be an array of complex objects describing the attributes and their current values.

```
$c.put({ "cores":"2","memory":"2G"});
```

Here the instance is updated using the values available in the complex variable "c".

```
$v.delete();
```

Finally the instance may be deleted using the universal unique identifier of the category instanced returned by the creation operation and stored in the variable "v".

Methods

As stated above this section defines the methods that may be performed on either OCCI instance identifiers or on OCCI category identifiers.

POST, CREATE, NEW

These three are synonymous and perform the creation of a new instance of the required category.

Examples:

```
Compute.post();
```

This first example will send an OCCI POST message to the published category manager for the "compute" category. The configuration of the publisher address will depend upon the host environment of the Cordscript interpreter.

```
$v = "http://host/compute";

$v.new();
```

This second example show how an alternative or explicit OCCI Category manager can be used for the creation of an instance of the required category.

Both of these preceding examples can be adapted to specify an invocation parameter which provides the description of the attribute values to be used for the creation of the resource instance.

```
$v = "http://host/compute";

$v.new("occi.compute.speed=2G,occi.compute.memory=4G");

$v = compute.post("occi.compute.cores=4,occi.compute.memory=16G");
```

GET, RETRIEVE

These two are synonymous and perform retrieval of the category instance identified by the corresponding universal unique object identifier. When used for a category name they will retrieve the collection of instance identifiers of the instances matching eventual attribute values submitted as research filter criteria.

Examples:

```
$v = compute.post("occi.compute.cores=4,occi.compute.memory=16G");

$r = $v.get();
```

Here we see the retrieval of the resource instance of a newly created "compute" category. This can only be achieved through an instance identifier.

PUT, UPDATE

These two are synonymous and perform update of the category attribute values of the instance identified by the corresponding universal unique object identifier.

```
$v = compute.post("occi.compute.cores=4,occi.compute.memory=16G");

$r = $v.put("occi.compute.speed=4G");
```

Here we see the partial update of a newly created resource instance of the "compute" category.

DELETE

This will perform deletion of the category instance defined by the universal unique object identifier, or the deletion of the collection of category instances defined by the category identifier.

Examples:

```
$v = compute.post("occi.compute.cores=4,occi.compute.memory=16G");

$v.delete();
```

Here we see the deletion of the newly created resource instance of the "compute" category.

```
compute.delete();
```

This second example shows that through the category name it is possible to delete a collection of resource instances of that category. In this particular case ALL resource instances will be deleted. A standard OCCI Attribute filter may be provided to allow precise selection of the instances for deletion.

LINK

This method when applied to a valid OCCI category instance identifier, and when provided with a valid OCCI category instance identifier as parameter, will create an OCCI LINK instance where the subject is the source element and the target is the parameter instance.

DATE

This method when applied to an integer subject will return a standard date and time string corresponding to the provided time value in seconds since the 1st of January 1970.

DISPLAY

This will display the contents of the subject on the output stream. This may be the category name, the object identifier or the list of object identifiers or the attributes of the category instance.

Examples:

$v.display();

"hello world".display();

COUNT

This method operates as for the LIST method but instead of return the array of instance identifiers returns the integer count of matching category instances.

LENGTH

This method returns the length of the subject string.

DEBUG

This method will set the operational debug flag to the value of its subject.

Example:

"1".debug();

This example shows how the debug execution trace can be activated during execution.

ROUND

This method rounds up the real number subject, to the count of decimal places indicated by the invocation parameter. The result is returned via the stack.

Example:

$f = 34.5677889

$x = $f.round(2);

$x == 34.57

WAIT

This will perform a timed wait as defined by the subject of the invocation.

Example:

"5".wait();

This will suspend execution for 5 seconds.

Actions

The above defined collection of actions is complimented with the collection of OCCI Actions available for a particular category or category instance. Actions will be invoked directly, not through the scheduler, through the POST method and the standard OCCI invocation syntax.

Examples:

```
Compute.start();
Compute.stop();
Service.start();
```

```
        Service.stop();
        Service.snapshot();
```

Examples

In this section of the document we shall demonstrate the use of Cordscript for particular purposes.

Category List

Here we shall collect the list of category managers published through the accords platform.

```
"Start of Example".display();
$categories = [];
$list = publication.list();
Foreach ( $list as $url )
{
        $record = $url.get();
        Foreach ( $record as $attribute )
        {
                Forboth ( $attribute as $name and $value )
                {
                        If ( $name == "occi.publicartion.what" )
                        {
                                $categories = $categories + $value;
                        }
                }
        }
}
"List of Published Categories".display();
Foreach ( $categories as $category )
{
        $category.display();
}
"End of Example".display();
```

The above example, using the "foreach" and "forboth" instructions to scan the attribute list of the publication records, could be simplified using the member operator as follows:

```
"Start of Example".display();
$categories = [];
$list = publication.list();
Foreach ( $list as $url )
{
        $record = $url.get();
        $v = $record["occi.publication.what"];
        $categories = $categories + $v;
}
"List of Published Categories".display();
Foreach ( $categories as $category )
{
        $category.display();
}
"End of Example".display();
```

Restart Services

This example shows how the entire collection of service instances can be backed up to create snapshots and then restarted using Cordscript instructions:

```
$list = service.list();
Foreach ( $list as $item )
{
        $item.snapshot();
        $item.restart();
}
```

Here we see the use of the OCCI Actions “snapshot” and “restart”, as defined for the “service” category.

Transaction Processing

In this more complicated example we shall see how transactions may be retrieved by account and collated to provide an overall indication of current commercial performance.

```
// collect the list of accounts
// ----------------------------
$accounts = account.list();
$grandtotal = “0”;
$totals   = []
// retrieve each account and calculate transactions
// ------------------------------------------------
Foreach ( $accounts as $account )
{
         $total = 0;
        $account_information = $account.get();
        // list the transactions for the account
        // -------------------------------------
        $transactions = transaction.list(“occi.transaction.account=”+$account);
        Foreach ( $transactions as $transaction )
        {
                $transaction_details = $transaction.get();
                Foreach ( $transaction_details as $attribute )
                {
                        // scan the transaction record to locate the price
                        // -----------------------------------------------
                        Forboth ( $attribute as $name and $v;alue )
                        {
                                If ( $name == “occi.transaction.price” )
                                {
                                        $prix = $value.get();
                                        $value = $prix.locate(“value”);
                                        Forboth ( $value as $n and $v )
                                        {
                                                $total = $total + $v;
                                        }
                                }
                        }
                }
        }
        $grandtotal = $grandtotal + $total;
        $totals = $totals + $total;
}
```

Machine Code

Equally for extensibility as for ease of implementation and speed of execution, the machine code instructions produced by the Cordscript Compiler target a STACK based architecture rather than a REGISTER based architecture. This signifies that values are pushed to the stack as results of preceding instructions and will be pulled off the stack as operands to subsequent instructions.

General

NOP

This instruction performed no operation. It simply transfers control to the next instruction in sequence. These instructions may be generated as place markers to be targeted in advance by the control transfer instructions of compound statements such as IF, WHILE, FOR, SWITCH etc.

PUSH value

This instruction pushes its operand value onto the stack.

POP

This instruction removes and discards the top most value from the stack.

DUP

This instruction duplicates the top most value of the stack.

EVAL value, value, value

This evaluates a pair of value operands as being either:

- an object identifier and method or action
- a category identifier and method
- An optional count of optional stack based parameters representing the values passed between the calling braces.

Invocation of the resulting construction allows access to the underlying OCCI category management components of the Accords Platform. The method may be any one of the methods defined in the preceding section or any one of the actions described by the capabilities of the corresponding OCCI server.

Arithmetical

CAT value

This instruction unconditionally concatenates two values as string irrespective of their type and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value # Stack Value => New Stack Value
```

Example:

```
CAT     "bonjour"
```

Concatenates “bonjour” with the top of the stack value and pushes the result onto the stack.

AND value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then a logical bitwise AND of the two values and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value & Stack Value => New Stack Value
```

Examples:

```
AND     5
```

Performs a logical bitwise AND operation between 5 and the top of the stack value and pushes the result onto the stack.

OR value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then performs a logical bitwise OR of the two values and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value | Stack Value => New Stack Value
```

Examples:

```
OR      5
```

Performs a logical bitwise OR operation between 5 and the top of the stack value and pushes the result onto the stack.

XOR value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then performs a logical bitwise XOR of the two values and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value ^ Stack Value => New Stack Value
```

Examples:

```
XOR     5
```

Performs a logical bitwise XOR operation between 5 and the top of the stack value and pushes the result onto the stack.

ADD value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then adds or concatenates the two values and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value + Stack Value => New Stack Value
```

Examples:

```
ADD     5
```

Adds 5 to the top of the stack value and pushes the result onto the stack.

```
ADD     “bonjour”
```

Concatenates “bonjour” with the top of the stack value and pushes the result onto the stack.

```
ADD     [“a”,”b”]
```

Adds the top of the stack value to the array operand and pushes the resulting array onto the stack.

SUB value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then subtracts the stack value from the operand value and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value - Stack Value => New Stack Value
```

Example:

```
SUB     10
```

Subtracts the top of the stack value from 10 and pushes the result onto the stack.

MUL value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then multiples the values and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value * Stack Value => New Stack Value
```

Example:

```
MUL     2
```

Multiply the top of the stack value by 2 and push the result onto the stack.

DIV value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then divides the operand value by the stack value and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value / Stack Value => New Stack Value
```

Example:

```
DIV	10
```

Divide 10 by the top of the stack value and push the result onto the stack.

MOD value

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and then performs modulus division of the operand value by the stack value and pushes the result back onto the stack.

This can be represented by the following pseudo code:

```
Operand Value Modulus Stack Value => New Stack Value
```

Example:

```
MOD	12
```

Calculate the modulus division of 12 by the top of the stack value and push the result onto the stack.

Control Transfer

JMP instruction

This instruction performs unconditional transfer of control to the indicated instruction or address.

ENTER instruction

This instruction marks the entry into a fault tolerant section of code and sets the operand instruction as the current exception handler to which all uncaught exceptions will be sent.

LEAVE instruction

This instruction marks the exit from a fault tolerant section of code and restores the previous exception hander. The optional instruction operand may cause unconditional branching to the indicated instruction otherwise processing will continue in sequence.

CALL instruction

This instruction performs unconditional transfer of control to the indicated instruction context of the targeted function. The context will store the current return address for use by the subsequent return instruction with or without a return value.

RET

This instruction performs unconditional transfer of control to the return address of the calling context of through an eventual CALL instruction.

RETVAL

This instruction performs unconditional transfer of control to the return address of the calling context of through an eventual CALL instruction.

The value operand will be popped off the operational stack prior to being pushed to the return stack after recovery of the code segment index pointer and associated execution context.

EACH array, variable, instruction

This instruction expects three parameters, the source array variable, the output variable and the exit label. The source array variable is expected to be a CSON array and the instruction will cut the first element from the array and return it as the output variable. The resulting array will be saved to the source variable and the iteration will terminate with transfer of control to the indicated instruction when the source array is empty.

BOTH array, name, value, instruction

This instruction expects four parameters, the source array variable, the two output variables and the exit label. The source structure variable is expected to be a CSON object and the instruction will cut the first name and value pair element from the structure to be returned as the output variables. The resulting structure will be saved to the source variable and the iteration will terminate with transfer of control to the indicated instruction when the source structure is empty.

JEQ value, instruction

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and performed conditional transfer of control to the indicated instruction or address if the operand value is equal to the stack value. When both values provide invalid, null or not a number values, the instruction will also evaluate to *true*. Transfer of control will be to the next instruction in sequence if the condition evaluates to *false*.

JLS value, instruction

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and performed conditional transfer of control to the indicated instruction or address if the operand value is strictly less than the stack value. When the value operand is invalid, null or not a number value, and when the stack value is also valid, then the instruction will also evaluate to *true*. Transfer of control will be to the next instruction in sequence if the conditions evaluate to *false*.

JLE value, instruction

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and performed conditional transfer of control to the indicated instruction or address if the operand value is less than or equal to the stack value. When the value operand is invalid, null or not a number value, and that the stack value is valid or invalid, then the instruction will also evaluate to *true*. Transfer of control will be to the next instruction in sequence if the condition evaluates to *false*.

JGR value, instruction

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and performed conditional transfer of control to the indicated instruction or address if the operand value is strictly greater than the stack value. When the value operand is valid and the stack value is invalid, null or not a number value, then the instruction will also evaluate to *true*. Transfer of control will be to the next instruction in sequence if the condition evaluates to *false*.

JGE value, instruction

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and performed conditional transfer of control to the indicated instruction or address if the operand value is greater than or equal to the stack value. When the value operand is valid or invalid and the stack value is invalid, null or not a number value, then the instruction will also evaluate to *true*. Transfer of control will be to the next instruction in sequence if the condition evaluates to *false*.

JNE value, instruction

This instruction evaluates the value operand, pops and evaluates the top most value from the stack and performed conditional transfer of control to the indicated instruction or address if the operand value is not equal to the stack value. When both values provide invalid, null or not a number values, then the instruction will evaluate as *false*. Transfer of control will be to the next instruction in sequence if the condition evaluates to *false*.

Conclusions

Cordscript provides all the constructions of a modern scripted programming language and as such allows programmatic control of the various events in the CORDS model and provisioning system life cycle.

This will be of particular importance in the following fields:

1. Service Level Agreement Guarantees and Business Values

   The business value category instances defined for service level guarantees makes provision for control over penalty and reward behavior through cordscript instructions provided by their “expression” attribute. This currently exhibits a pre-defined default behavior involving the creation of penalty records for all penalties incurred. This is accompanied by the deletion of the packets of monitoring data in order to avoid accumulation that would lead to a big data problem. This is an area where the application of cordscript programs would allow examination of the monitoring data over a particular period would allow finer control over penalty management and reward attribution. This could be then performed on an agreement defined basis rather than occurring in a one size fits all fashion.

2. Placement Algorithms

   The standard placement engine makes provision for the definition of the type of placement algorithm to be employed during the placement operation. The standard hard coded values of “default” and “quota:default” along with the various zone, reputation, security and price related variations are far from satisfactory for the majority of cases. The use of cordscript for the definition of new placement algorithms would increase the flexibility of the placement engine and more especially with respect to provisioning extensions for PAAS, BPAAS or WAAS expected in the future.

3. Customer Invoice Processing

   The transaction processing performed for the production of customer invoices is currently very primitive performing a simple costing report with no attempt at correspondence with account based commercial policy. The definition of commercial policy would benefit greatly from the use of cordscript programming on an account to account and customer to customer basis allowing greater flexibility in this important domain.

4. Security Policies

   As for the preceding topic concerning commercial policy another field which would greatly benefit from the adjunction of cordscript would be that of role based access policy control and enforcement on again an account to account basis.

References

1. CORDS Reference Manual V2.12

2. Accords Platform Reference V1.3

3. OCCI CORE Model

4. OCCI Infrastructure Mode

5. OCCI HTTP Rendering

6. OCCI JSON Rendering

Annex 1

This annex describes the machine code sequences generated for the different iterative and conditional compound instructions of Cordscript. These sequences of instructions are generated during the compilation phase and result in the production of the execution graph.

IF ELSE

The following simple IF and ELSE construction:

```
If ( $v > 0 )
{       $v.display();  }
Else
{       $v.display();  }
```

Will be compiled to produce the following machine code:

```
        PUSH    0

        JLE     $V      False

        EVAL    $V      “display”

        JMP     Exit

False:  NOP

        EVAL    $V      “display”

Exit:   NOP
```

FOR

The following simple FOR construction:

```
For ( $v=1; $v < 10; $v = $v + 1 )
{       $v.display();  }
```

Will be compiled to produce the following machine code:

```
        PUSH    1

        SET     $V

        JMP     Entry

Retry:  NOP

        PUSH    10

        JGE     $V      Exit

        JMP     Entry

Next:   NOP

        PUSH    1

        ADD     $V

        SET     $V

        JMP     Retry

Entry:  EVAL    $V, “display”
```

```
        JMP     Next

Exit:   NOP
```

FOREACH

The following simple FOREACH construction:

```
Foreach ( $list as $output )
{       $output.display();     }
```

Will be compiled to produce the following machine code:

```
Next:   EACH    $list   $output Exit

        EVAL    $output “display”

        JMP     Next
```

FORBOTH

The following simple FORBOTH construction:

```
Forboth ( $complex as $name and $value )
{
        $name.display();
        $value.display();
}
```

Will be compiled to produce the following machine code:

```
Next:   BOTH    $complex       $name   $value  Exit

        EVAL    $name   “display”

        EVAL    $value  “display”

        JMP     Next

Exit:   NOP
```

WHILE

The following simple WHILE construction:

```
While ( $v != 0 )
{
        $v.display();
        $v = v - 1;
}
```

Will be compiled to produce the following machine code:

```
Next:   NOP

        PUSH    0

        JEQ     $V      Exit

        EVAL    $V      “display”

        PUSH    1

        SUB     $V

        SET     $V

        JMP     Next
```

```
Exit:   NOP
```

SWITCH CASE DEFAULT

The following simple SWITCH, CASE and DEFAULT construction:

```
Switch ( $v )
{
Case    1       :
        "one".display();
        Break;
Case    2       :
        "two".display();
Default         :
        "other value".display();
}
```

Will be compiled, with optimization, to produce the following machine code:

```
        PUSH    $V

        DUP

        JNE     1       Next

        EVAL    "one"   "display"

        JMP             Exit

        JMP             Over

Next:   DUP

        JNE     2       Next2

Over:   EVAL    "two"   "display"

        JMP             Next2

Next2:  NOP

        EVAL    "other value" "display"

Exit:   POP
```

TRY CATCH

The following simple TRY and CATCH construction:

```
Try
{
        Throw 1;
}
Catch ( $v )
{
        $v.display();
}
```

Will be compiled to produce the following machine code:

```
        ENTER   Catch

        PUSH    1

        JMP     Catch

        LEAVE   Exit

Catch:  LEAVE
```

```
        SET     $V
        EVAL    $V      “display”
Exit:   NOP
```

Annex 2

This annex describes the execution context used during the execution of the resulting Cordscript execution graph. Each instruction in the graph references a single parent context which provides the following elements:

STACK

This is the central element of the Cordscript execution context and provides a last in first out, LIFO, stack onto which results produced on output by instructions are placed to be used as input to subsequent instructions.

DATA

This element provides the list of variables declared for the context.

CODE

The element contains the list of functions defined for the context.