

SPELL Language Reference

SPELL version 2.4.4

Distribution list

Full Report:

SES-SSO-SOE

For Information Purposes:

Open Source

Reference:

SES-SSO-SOE-SPELL-2015/06

Date issued:

February 2015

Acronyms

CV	Command Verification
GCS	Ground Control System
GDB	Ground Database
GUI	Graphical User Interface
HMI	Human Machine Interface (equivalent to GUI)
IDE	Integrated Development Environment
MMD	Manoeuvre Message Database
OOL	Out-of-limits
PDF	Portable Document Format
PROC	Automated SPELL procedure
RCP	Rich Client Platform
S/C	Spacecraft
SCDB	Spacecraft Database
SDE	SPELL Development Environment
SEE	SPELL Execution Environment
SES	Société Européenne des Satellites
SPELL	Satellite Procedure Execution Language and Library
TC	Telecommand
TM	Telemetry
URI	Uniform Resource Identifier
USL	Unified Scripting Language
UTC	Coordinated Universal Time

Table of Contents

1	Introduction	12
1.1	Purpose of this document.....	12
1.2	Unified Scripting Language	12
1.3	A first on SPELL features	12
1.4	Framework architecture	12
2	The Python language.....	14
2.1	Python constructs	14
3	The SPELL language	26
3.1	SPELL drivers	26
3.2	Configuration and modifiers	26
3.3	Operation error handling.....	27
3.4	Reacting on operation result	28
3.5	Special error handling	30
3.6	Silent execution	31
4	SPELL language services	32
4.1	Time management.....	32
4.2	Telemetry values acquisition	36
4.3	Verifying telemetry values.....	39
4.4	Building commands.....	47
4.5	Sending commands	48
4.6	Holding execution.....	55

SPELL version 2.4.4

4.7	Injecting TM parameters.....	59
4.8	Getting and setting TM parameters limits.....	59
4.9	Displaying messages.....	68
4.10	Injecting events.....	68
4.11	Ground control system configuration and variables	69
4.12	Requesting Information.....	70
4.13	Managing GCS system displays	72
4.14	Procedure execution flow control	75
4.15	Databases	78
1.2	Data containers.....	84
4.16	Sub-procedures	86
4.17	File manipulation	90
4.18	Ranging operations	93
4.19	Sharing data between procedures	95
4.20	Memory management functions	100
4.21	Ground control system TM/TC database	102
5	Appendix A: table of functions	104
6	Appendix B: table of modifiers	108
7	Appendix C: modifier default values	114
8	Appendix D: modifiers applicable to all functions	116

Table of Examples

Example 1: indentation	14
Example 2: splitting character strings.....	15
Example 3: splitting with comma	15
Example 4: splitting code with backslash	16
Example 5: single line comments in python	16
Example 6: multi-line comments in python.....	16
Example 7: variable types in Python	17
Example 8: lists and dictionaries	18
Example 9: arithmetic expressions.....	20
Example 10: boolean expressions	21
Example 11: string expressions	21
Example 12: substrings	21
Example 13: conditional statements.....	22
Example 14: for loop statements	23
Example 15: while loop statements.....	23
Example 16: user defined function example	24
Example 17: arguments by reference	24
Example 18: module (procedure) import.....	25
Example 19: SPELL function with keyword parameters	27
Example 20: changing defaults for a function in a procedure	27
Example 21: failure action configuration	28
Example 22: automatic failure processing.....	28
Example 23: reacting on function result	30
Example 24: special error handling	31
Example 25: silent execution.....	31
Example 26: using NOW in a loop	33
Example 27: using NOW in a loop (2)	33
Example 28: modifying the output time format	34
Example 29: time management.....	35
Example 30: obtain engineering TM values	36


Example 31: waiting for updates	37
Example 32: waiting for updates with timeout	37
Example 33: acquisition of raw value	37
Example 34: acquisition extended information	38
Example 35: verify a single TM value	39
Example 36: verify a single TM value in raw format	40
Example 37: verify using next update and timeout	40
Example 38: verify with tolerance	40
Example 39: using Verify in a test statement	41
Example 40: getting failure information	41
Example 41: getting failure information (output)	41
Example 42: do nothing on False	42
Example 43: do not prompt the user and take the <code>SKIP</code> action	42
Example 44: repeat verification if it fails, 2 times	42
Example 45: delay the verification	43
Example 46: ignoring string value case	43
Example 47: multiple verifications	44
Example 48: multiple verifications and configurations	44
Example 49: multiple verifications and configurations (2)	44
Example 50: verifications with multiple values	45
Example 51: comparing telemetry parameters	45
Example 52: comparing telemetry parameters (wrong code)	45
Example 53: comparing telemetry parameters with items	46
Example 54: comparing telemetry parameters with lists of items	46
Example 55: building a simple command	47
Example 56: building a simple command	47
Example 57: building a simple command	47
Example 58: command arguments	48
Example 59: building a command with arguments	48
Example 60: sending a simple command	49
Example 61: time-tagged commands	49
Example 62: commands with a release time	50
Example 63: load only command	50


Example 64: confirm command execution	50
Example 65: confirm critical command execution	51
Example 66: commands with arguments	51
Example 67: sending sequence	51
Example 68: sending a group of commands	52
Example 69: sending a group of commands with arguments	52
Example 70: sending a list of commands grouped	52
Example 71: sending a list of commands blocked	52
Example 72: sending with timeout.....	53
Example 73: additional information	53
Example 74: sending with delayed release	53
Example 75: sending and verifying	54
Example 76: sending, verifying and adjusting limits.....	54
Example 77: complex send	55
Example 78: wait relative time condition	56
Example 79: wait absolute time condition	56
Example 80: wait telemetry condition	57
Example 81: wait telemetry condition with maximum delay	57
Example 82: simple interval.....	58
Example 83: complex interval	59
Example 84: injecting parameters	59
Example 85: reading all limit definitions	61
Example 86: complex limit definitions for HARDSOFT limit.....	62
Example 87: complex limit definitions for STATUS limit	62
Example 88: reading applicable limit definitions.....	62
Example 89: reading limit definitions by identifier	62
Example 90: reading limit definitions from status parameters.....	63
Example 91: reading single data from limit definitions	63
Example 92: reading single data from limit definitions (2).....	63
Example 93: reading single data from limit definitions (3).....	63
Example 94: modifying limit definitions	64
Example 95: modifying limit definitions (2)	64
Example 96: modifying limit definitions (3)	64

Example 97: modifying limit definitions (4)	65
Example 98: modifying limit data for Status parameter	65
Example 99: modifying limit definitions (5)	65
Example 100: modifying limit definitions for SPIKE limit	65
Example 101: modifying limit definitions for STEP limit	66
Example 102: enabling and disabling alarms	66
Example 103: automatic limit adjustment for HARDSOFT parameter	66
Example 104: automatic limit adjustment for STATUS parameter	67
Example 105: loading limits from file	67
Example 106: restoring limits	67
Example 107: checking alarm status	68
Example 108: display messages	68
Example 109: sending notifications	68
Example 110: event messages	69
Example 111: change configuration value	69
Example 112: retrieve configuration value	69
Example 113: variable mappings with ground database	69
Example 114: types of prompt	70
Example 115: default prompt list	71
Example 116: custom list with index	71
Example 117: custom list with values	71
Example 118: default value for prompt	72
Example 119: opening displays	72
Example 120: opening a workspace	73
Example 121: specifying the host	73
Example 122: specifying the monitor	73
Example 123: specifying time span	73
Example 124: printing displays	74
Example 125: display format	74
Example 126: closing displays	74
Example 127: closing workspaces	74
Example 128: step definitions	75
Example 129: step and go-to	75

Example 130: displaying step information	76
Example 131: pausing, aborting and finishing.....	77
Example 132: setting user actions	77
Example 133: enabling/disabling user actions	77
Example 134: removing user actions	78
Example 135: spacecraft database	78
Example 136: spacecraft database keys.....	78
Example 137: checking spacecraft database keys	79
Example 138: loading a manoeuvre file	79
Example 139: ground database mappings.....	80
Example 140: using database mappings	80
Example 141: using database mappings (2)	80
Example 142: procedure database	81
Example 143: loading an user database file	81
Example 144: creating an user database file	82
Example 145: saving an user database file.....	82
Example 146: modifying a database value.....	83
Example 147: database data example	83
Example 148: database data example manipulation	84
Example 149: creating a data container	85
Example 150: declare a variable in a data container	85
Example 151: starting a sub-procedure	86
Example 152: procedure library example.....	87
Example 153: folder priorities	87
Example 154: starting procedure using priorities	87
Example 155: starting procedure overriding priorities	87
Example 156: starting a sub-procedure in non-blocking mode	88
Example 157: starting a sub-procedure in hidden mode.....	88
Example 158: starting a sub-procedure in manual mode.....	88
Example 159: starting a sub-procedure (complex)	89
Example 160: starting a sub-procedure with arguments	89
Example 161: accessing procedure arguments	89
Example 162: opening a file	90

Example 163: opening a file	90
Example 164: writing to a file.....	91
Example 165: reading file lines	91
Example 166: reading directory contents	91
Example 167: create a File instance	92
Example 168: deleting files and directories.....	92
Example 169: enabling or disabling the ranging system	93
Example 170: starting ranging activities	94
Example 171: starting dual ranging activities	94
Example 172: aborting ranging activities.....	94
Example 173: setting and getting baseband configurations.....	94
Example 174: getting BBE/Antennae information	95
Example 175: calibrate the ranging system	95
Example 176: status of the ranging system	95
Example 177: setting shared data values	96
Example 178: setting shared data values (2)	96
Example 179: setting shared data values (3)	96
Example 180: setting shared data values in a scope	97
Example 181: creating a scope	97
Example 182: test and set shared data values	97
Example 183: test and set shared data values (2).....	97
Example 184: getting shared data values	98
Example 185: getting shared data values (2).....	98
Example 186: getting shared data values (3).....	98
Example 187: getting shared data information.....	98
Example 188: clearing shared data values	99
Example 189: clearing shared data scopes	99
Example 190: clearing shared data scopes	99
Example 191: generate memory reports	100
Example 192: compare memory images.....	101
Example 193: compare memory images on limited data ranges	101
Example 194: extract memory values	102
Example 195: extract TM/TC database values	103

03 February, 2015	SPELL Language Reference	
Page 11 of 118	File: SPELL - Language Reference - 2.4.4.docx	

03 February, 2015	SPELL Language Reference	
Page 12 of 118	File: SPELL - Language Reference - 2.4.4.docx	

1 Introduction

1.1 Purpose of this document

This document is the software user manual for the SPELL language and library. It is intended to be used mainly by SPELL procedure developers but it can be of use as well for S/C controllers & engineers.

1.2 Unified Scripting Language

USL stands for Unified Scripting Language. The USL concept includes all aspects from automated procedure creation to execution. The goal of USL is to provide automated procedures for all S/C platforms that may work with a variety of ground satellite control systems and, in general, with any ground segment element. The software part of USL is SPELL, the Satellite Procedure Execution Language and Library. SPELL is a software framework developed by SES (www.ses.com) and GMV (www.gmv.com) whose main component is a scripting language based on the well-known, widely used language *python* (www.python.org).

1.3 A first on SPELL features


Some of the strong points of the SPELL framework are:

- Language based on python: the language is flexible, readable and powerful
- GCS independent: it may be used with a variety of ground control systems
- S/C independent: the same procedures may be used on several spacecraft of the same type thanks to the usage of databases
- Extensible language: the language can be extended by adding new functions
- Extensible drivers: SPELL may be extended to interact with new ground segment components

1.4 Framework architecture

The SPELL architecture can be divided into two parts, the *SPELL execution environment (SEE)* and the *SPELL development environment (SDE)*. They can be also seen as the on-line and the off-line part of the SPELL software.

- The execution environment includes all the elements needed for executing procedures. The main components of this environment are: (a) the SPELL server, core of the environment, where the procedure execution is performed. It is responsible of coordinating all tasks, interfacing with ground control systems, etc. (b) the SPELL clients, graphical interfaces through which the procedure executions are controlled and supervised by S/C controllers and engineers.
- The development environment corresponds to an IDE (Integrated Development Environment), based on Eclipse IDE, which provides all the tools and features required for coding SPELL procedures.

03 February, 2015	SPELL Language Reference	
Page 13 of 118	File: SPELL - Language Reference - 2.4.4.docx	

A third part may be considered, the language library. This library is used in both environments and defines the SPELL language syntax and functionalities. It is extensible: new functions and language constructs can be added to the base language in order to append new functionalities to the framework, for example, the ability to interface a new ground segment component, or a new set of functions to connect SPELL with another piece of software.

2 The Python language

In this chapter the Python language is described. Since SPELL is a scripting language based on python, a basic description of this language is provided in first place; the SPELL language description will follow in the next chapter.

2.1 Python constructs

All the rules applicable to Python scripts are applicable to SPELL procedures. Therefore, it should be recommended to read the official Python manuals and library references (<http://www.python.org>). However, a brief description of the most relevant Python elements is given in this section.

2.1.1 Indentation

Indentation is a key element in any Python script, since all code blocks (functions, `if-then-else` statements, `for/while` loops) are defined with indented lines. In the following example, the `if` expression (assign 2 to variable B) is executed whenever the `if` condition (A equals 1) is evaluated to `True`; otherwise, the `else` expression is executed (assign 4 to variable B). Notice that any *indented* code placed between the '`if`' and '`else`' clauses will be taken as part of the '`if`' expression. The same applies for the '`else`' expression.

Python interpreter identifies the end of the `else` block when the indented block finishes. In the example, the statement '`C=4`' is outside the `else` expression.

Example 1: indentation

```
if (A == 1):  
    B=2  
else:  
    B=4  
C=4
```

2.1.2 Multi-line code blocks

It is recommended that when coding SPELL procedures, the source code should not be more than 80-100 characters wide. To achieve this, it is quite common that Python calls or code blocks shall be splitted in several lines. For that matter Python provides three mechanisms: the comma, the symbol '+' and the symbol '\n' (backslash).

Python character strings can be splitted along several lines using the '+' symbol plus the carriage return:

Example 2: splitting character strings

```
if (A == 'This is a very long string and ' +  
    'I have to split it in several ' +  
    'lines so that it is not too long'):  
    ...  
else:  
    ...
```

As a matter of fact, the '+' symbol is not mandatory but it is strongly recommended to use it for code clarity and readability.

Any Python call containing other items different from strings can be splitted in a similar way by using the backslash ('\') and the comma. Function calls can be splitted in several lines of code by introducing carriage returns after the comma symbol. The same applies to the construction of lists and dictionaries:

Example 3: splitting with comma

```
result = Long_Function_Call( argument1 = A,  
                             argument2 = B,  
                             argument3 = C )  
  
my_list = [ 43545,  
            21234,  
            32443 ]  
  
my_dict = { 1:'Parameter A',  
            2:'Parameter B',  
            3:'Parameter C' }
```

Other Python code blocks can be splitted with the aid of the slash character. For example, long arithmetic or boolean expressions:

Example 4: splitting code with backslash

```
result = Condition1 + Function(A,B) +\  
        Condition3  
  
boolean_condition = A and B \  
                    or not C  
  
X = 1 + 45 -\  
    23+5.4 + A/45
```

IMPORTANT: Please notice that the backslash ('\') character shall be **the last character of the line**. Any space left after the backslash will result on a syntax error.

2.1.3 Comments

In Python (and SPELL) scripts, any text following a sharp character ('#') is a comment and is ignored by the interpreter. An example is shown below:

Example 5: single line comments in python

```
# This is a comment line  
  
A = 1 # This is an in-line comment
```

Multi-line comments are written by using *docstrings*. Docstrings are comment blocks delimited by three single-quote characters:

Example 6: multi-line comments in python

```
'''  
This is a multi-line comment  
This is a multi-line comment  
'''
```

2.1.4 Variable types

Python variables have no data type associated. That is, the same variable may be used to store any data type (integers, character strings, objects, files and so on). The following example code is correct in Python:

Example 7: variable types in Python

```
A=1
A='This is a string'
A=['1', 1.0, {'KEY':'VALUE'}]
```

When executing the example, the interpreter would store the different data in the variable A: first an integer, then a string, and then a text file object.

In Python, variable declaration and initialization occur at the same time. Variables do not have to be declared nor defined before actually assigning values to them. As soon as a value is assigned to a variable, the variable exists in the script, but not before.

2.1.5 Basic data types

The basic data types in Python are:

- Integer: arbitrary large signed integers
- Float: double precision, like 1.23 or 7.8e-28
- Boolean: True or False (notice capital T and F)
- Strings: using single or double quotes indifferently
- Lists: using square brackets; e.g. A=[0,1,2,3]
- Dictionaries: using curly braces; e.g. A={ 'KEY1':'VALUE', 'KEY2':'VALUE' }

Data stored in lists can be accessed as shown in the example below. For dictionaries, a key is used instead of an index. In Python dictionaries, keys and values can be of any type. The same occurs with Python lists: list elements can be other lists or dictionaries, classes or functions. Moreover, the elements of a list or dictionary can be of different type.

Example 8: lists and dictionaries

```

mylist = [111,222,333,444]
A      = mylist[2]           # A stores 333

mydict = {1:'FOO', 2:'BAR'}
B      = mydict[2]          # B stores 'BAR'

# This list contains several elements:
#   - another list
#   - a character string
#   - a dictionary
biglist = [
    [ 1, 2, 3, 4 ],
    'My string',
    { 'A':'Option 1' }
]

biglist[0] → [1,2,3,4]
biglist[1] → 'My string'
biglist[2] → {'A':'Option 1'}

# Brackets can be combined:
biglist[0][3] → 4
biglist[2]['A'] → 'Option 1'

```

Notice that Python lists are indexed starting at 0. See the Python reference for details.

2.1.6 Python is case-sensitive

The Python language is case-sensitive. That means that capital letters are taken into account when evaluating expressions. For example, the variable name "Value" is completely different from the variable name "value" in Python.

2.1.7 Arithmetic expressions

Arithmetic expressions can be created using the following arithmetic operators:

Operator	Meaning
**	exponent
*	multiply
/	divide
%	modulus

Operator	Meaning
+	add
-	subtract
&	bitwise AND
	bitwise OR

The following table shows the basic set of mathematical functions available in Python:

Function	Meaning
pow(x,n)	value raised to the 'n'th power
sqrt(x)	square root
exp(x)	'e' raised to the given exponent
log(x)	natural Logarithm
log(x,b)	logarithm in base 'b'
log10(x)	logarithm in base 10
max(x)	maximum value within the supplied (comma separated) values
min(x)	minimum value within the supplied (comma separated) values
abs(x)	absolute value
ceil(x)	round up
floor(x)	round down
round(x,n)	round to 'n' decimals
sin(x)	sine (in radians)
asin(x)	arc sine
cos(x)	cosine
acos(x)	arc cosine
tan(x)	tangent
atan(x)	arc tangent
degrees(x)	convert to degrees
radians(x)	convert to radians

Find below some examples of arithmetic expressions:

Example 9: arithmetic expressions

```

a = abs(-7.5)      # Absolute Value
b = asin(0.5)      # Arc sine, returns in rads
c = pow(b,3)       # c = b^3
# Compound expression
d = ((max (a, b) + 2.0) % 10) * a

```

2.1.8 Boolean expressions

Boolean expressions evaluate to `True` or `False` and can be created using the following comparison and logical operators:

Comparison operator	Meaning
<code>==</code>	equal
<code>!=</code>	not equal
<code><</code>	smaller
<code>></code>	greater
<code><=</code>	smaller or equal
<code>>=</code>	greater or equal

Logical operator	Meaning
<code>and</code>	and
<code>or</code>	or
<code>not</code>	not

Boolean expressions can contain arithmetic expressions (please refer to the 'arithmetic expressions' section for more details).

Example 10: boolean expressions

```
a = 1
b = 2
c = 3

# This variable evaluates to True
boolean_variable = (a != b) and (not (b > c))

# This variable evaluates to True
x = True or (b==c)
```

IMPORTANT: the usage of the '<>' operator is strongly discouraged since it disappears in the Python 3.0 language specification.

2.1.9 String expressions

Python allows to concatenate strings using the '+' operator. Data types can be converted to strings using the `str` function.

Integers can be converted to strings containing their hexadecimal, octal, and binary values using functions `hex`, `oct`, and `bin` respectively.

Example 11: string expressions

```
i= 5
myvar = 'Integer:' + str(i) + ', Binary: ' + bin(i)
```

Strings can be manipulated as if they were arrays (lists). The square brackets ("[" "]") allow extracting substrings. An index or an index interval can be used:

Example 12: substrings

```
A='This is a string'
A[0] → 'T'                # Get the first character
A[0:3] → 'This'           # Get substring 0 to 3
A[:3] → 'This'            # Equivalent to previous
A[4:] → ' is a string'    # Substring 4 to end
A[-1] → 'g'              # Get the last character
```

Python also provides some string manipulation functions as part of the string data type:

- `str.center(width, [fillchar])` : returns the current string centered in a new string of length 'width' and doing the padding with a space character, or the 'fillchar' character if given.
- `str.capitalize()` : returns the current string with the first character capitalized.
- `str.startswith(sub), str.endswith(sub)` : checks if the current string starts or ends with the given string 'sub' respectively.
- `str.find(sub, [start], [end])` : tries to find the substring 'sub' in the current string, optionally searching within a given range specified by start and end.
- `str.isalpha(), str.isdigit(), str.islower(), str.isspace(), str.isupper()` : checks for the contents of the current string. For example, `isdigit()` returns True if the current string contains numbers only.
- `str.lower(), str.upper()` : converts the current string to lower or upper case.
- `str.replace(old, new)` : replaces the occurrences of 'old' by 'new' in the current string.

All these functions can be invoked on string objects. For further information, please refer to the Python documentation online.

2.1.10 Conditional statements

The clauses `if`, `elif` and `else` are used for these constructs. Boolean conditions are placed using either boolean variables or boolean expressions (please refer to previous sections for more details) as shown in the following example.

Example 13: conditional statements

```

if (<boolean expression>):
    ...
elif (<boolean expression>):
    ...
else:
    ...

```

Parentheses are optional but it is recommended to use them for code readability.

2.1.11 Loop statements

The following constructs are available to create control loops:

Construct	Meaning
for ... in ...	Loop a number of times Loop over the elements of an array

Construct	Meaning
while	Loop while a given expression evaluates to True
break	End execution of the current loop
continue	Skip the remaining statements in the loop and go to the next iteration

The following example shows different ways of using the `for` loop:

Example 14: for loop statements

```
# Loop iterating over a list (1,2,3,...)
for count in [1,2,3,4,5]:
    ...

# Loop iterating over dictionary keys ('A','B',...)
for dict_key in {'A':1, 'B':2, 'C':2}:
    ...

# Use range function for a 'counter' loop
for count in range(0,100):
    ...

# List elements don't need to be integers
for element in MyListOfElements:
    ...
```

The `while` loop is used with a Boolean condition:

Example 15: while loop statements

```
while (<boolean expression>):
    ...
```

2.1.12 User defined functions

The user may define functions as shown in the following examples. Please note the following:

- All arguments to any function are input arguments.
- The function may return one or several values using the `return` construct.
- All variables initialized within the function are only visible within the scope of the function.

The following example shows a function with two arguments and returning a list of three elements:

Example 16: user defined function example

```
# Function definition
def function(x,y):
    a = x+y
    b = x-y
    c = x*y
    return [a,b,c]

# Function call
x,y,z = function(1,2) # x, y and z store 3,-1 and 2
```

In Python, all function arguments are passed by reference. That means that the objects passed within the function arguments can be modified, with the *exception of Python primitive types*. That is, arguments containing strings, numbers or boolean expressions cannot be modified from within the function. On the other hand, a Python object (class instance), a list or a dictionary can be modified without restrictions.

Example 17: arguments by reference

```
# Adds 1 to the given argument
def functionA(x):
    x = x +1
    return

# Changes the first element of a passed list
def functionB(lst):
    lst[0] = 99
    return

x = 1
functionA(x)
x → 1 # x unmodified outside the function

x = [1,2,3]
functionB(x)
x → [99,2,3] # Lists can be modified
```

IMPORTANT: All functions defined in SPELL procedures shall finish with a `'return'` clause, even if nothing is returned at the end of the function.

2.1.13 Modules

Python is organized in *packages* and *modules*. Basically, a package is a directory, and a module is a Python script file where a particular set of data and functions are defined. In SPELL, a module (a Python script file) is called *procedure*.

It is quite common to organize the code in several modules (files), each one containing a set of functions and variables. Functions defined in a module can be used from within another module by *importing* the former in the latter:

Example 18: module (procedure) import

```
# FILE: moduleA.py


def functionA():          # Define a function here
    ...
```

```
# FILE: moduleB.py

import moduleA            # Import the module

moduleA.functionA()       # Use the external function
```

A module is imported by using the Python statement `import` and the name of the file to be imported. Please notice that, in order to use any entity defined inside a given module, the module name plus a *dot* shall be used right before the entity name. In the example, to call `functionA` it is necessary to qualify it with the name of the module and the dot, `moduleA`.

03 February, 2015	SPELL Language Reference	
Page 26 of 118	File: SPELL - Language Reference - 2.4.4.docx	

3 The SPELL language

As it has been said, all the rules applicable to Python are applicable to SPELL. Any SPELL procedure may contain and use any library or built-in types/functions available already in the default distribution of the Python interpreter.

Any additional Python module can be added and used as well; SPELL does not limit the extensibility of Python. This ability makes the SPELL language powerful, since the developer can install and use a wide variety of Python modules for complex mathematical calculations, data processing, version control, and much more.

The SPELL language introduces new functions that allow the procedure to interact with a ground control system (or another entity). All SPELL functions are used as regular Python functions. Nevertheless, some enhancements have been added to them, as it will be explained in the following sections.

Python scripts containing SPELL language constructs are called SPELL procedures. These cannot be executed as a regular Python script with the official Python interpreter: they need to be run within a SPELL execution environment.

The SPELL language includes a variety of functions for injecting, retrieving and manipulating telemetry data or telecommands, for user management, event handling and other services.

3.1 SPELL drivers

The SPELL functions interact with external entities like a GCS by means of an internal layer or middleware called SPELL driver. A driver is the intermediary between the procedure code and the system that the procedure is interacting with.

Drivers are the abstraction layer that makes SPELL procedures independent from the concrete system that is being controlled. This means that the same procedure can be used against different ground control systems. Although the way to actually interact with the GCS may change from one system to another, the SPELL driver take care of these differences internally and transparently to the procedure code and the user.

Depending on the driver being used, some of the SPELL language features may not be available. When a given GCS does not provide an interface for performing a particular task, the SPELL driver cannot provide support for such a task and. As a result, the SPELL functions in relation to that task will have no effect when used. The user will be aware of these cases, since a message will be shown warning the user about them.

3.2 Configuration and modifiers

All SPELL drivers and SPELL functions have a set of configurations for each service they provide. These configurations determine how the driver behaves when the corresponding service is used. That is, the configuration affects how all operations are carried out and how the results, and failures, are handled.

Configurations are based on sets of key/value pairs. Each key/value pair is called a *configuration parameter*, where the key is named *modifier*. Each SPELL function has a default or predefined behaviour, which is, a predefined set of SPELL modifiers and values. The procedure developer may customize the configuration of the functions for particular operations.

For example, it may be required to obtain the raw (non-calibrated) value of a telemetry parameter instead of using the engineering (calibrated) one. The default behaviour of SPELL function providing this service is to provide the engineering value, but using modifiers can change this.

To do so, a modifier with the desired value is passed as an additional argument to the SPELL function. That modifier value will override the default function configuration and will change its behaviour accordingly.

Please refer to the Python language reference for details about "keyword parameters". All SPELL functions use this mechanism which consists on passing the modifiers and values as keyword parameters. The following code example shows a function call where configuration is overridden by using keyword parameters.

Example 19: SPELL function with keyword parameters

```
Function( arg1, arg2,  Mod1=val1, Mod2=val2 )
```

Please refer to the section "Appendix B: table of modifiers" for more details about modifiers and their default values.

3.2.1 Change configuration defaults from procedures

All default configuration values for the SPELL functions are specified in the language configuration file. Nevertheless, these defaults can be changed inside a given procedure by using the `ChangeLanguageConfig` function. This function allows setting default values for a specific SPELL function:

Example 20: changing defaults for a function in a procedure

```
ChangeLanguageConfig( GetTM, Wait=True )
```

In the example above, the default value of the `Wait` modifier for the function `GetTM` will be `True`, no matter what is specified in the language configuration file. Notice that this operation will have effect on the current procedure only, and will not affect others.

It is possible to provide more than one modifier name to the function, to change several defaults at the same time.

3.3 Operation error handling

SPELL functions can handle any possible error coming from the controlled system or the SPELL driver when carrying out an operation. When such a failure happens, the SPELL function captures the error and prompts the user to decide which should be done.

There are different options that the operator may choose, depending on the nature of the action being performed. Some examples are:

- a) ABORT: abort the procedure execution
- b) REPEAT: repeat an operation
- c) RESEND: resend a command
- d) RECHECK: repeat a telemetry verification
- e) SKIP: skip the function call, acting as if no error happened, and return `True` if possible
- f) CANCEL: cancel the function call and continue, returning `False` if possible
- g) etc.

There are predefined options for each SPELL function, but they may be overridden by using SPELL modifier `OnFailure`. The value of this modifier shall be a set of one or more of the actions listed above (taking into account that not all of them are applicable always, depending on the function). The combination is made using the pipe '|' character. For example, passing the modifier `OnFailure = ABORT | REPEAT` will show only these two options to the user in case of failure.

Example 21: failure action configuration

```
Function( arg, OnFailure=ABORT|SKIP )
```

In the example, if the function call fails, the user will be able to choose between **(a)** aborting the procedure execution and **(b)** skipping the function call and continue.

It is also possible to program an automatic behavior for failures. The `PromptFailure` modifier may be used to indicate that the user shall not be prompted in case of failure, but rather perform automatically the action specified in `OnFailure`. Notice that `OnFailure` value shall be a single action in this case.

Example 22: automatic failure processing

```
Function( arg, PromptFailure=False, OnFailure=ABORT )
```

3.4 Reacting on operation result

When a SPELL function returns a boolean value (usually indicating whether the operation was successful or not), it can be configured to interact with the user depending on the result value.

This configuration is done by means of the modifiers `PromptUser`, `OnTrue` and `OnFalse`. Default values for these, and the corresponding meaning are:

- `OnTrue`: action(s) to be carried out, or prompt options to be used, when the SPELL function result is `True`. Usually the default value of this modifier is `NOACTION`, meaning that nothing will be done when the function result is `True`.

- **OnFalse**: action(s) to be carried out, or prompt options to be used, when the SPELL function result is False. Usually the default value of this modifier is **NOACTION**, meaning that nothing will be done when the function result is False.
- **PromptUser**: when at least one of the two modifiers above specifies a set of actions, this modifier indicates whether the user should be prompted (using the options indicated by the modifier **OnXXX**), or an automatic action should be carried out (the one indicated by the modifier **OnXXX**). Notice that, when **OnXXX** modifiers are used to specify prompt options, several actions can be combined with the pipe (|) character. On the other hand, if the **OnXXX** modifier specifies an automatic action, only one action code shall be given.

The following table clarifies the behavior of the SPELL functions depending on the values of these three modifiers:

Function result	PromptUser	OnTrue	OnFalse	Behavior
True	True	NOACTION	NOACTION	The function returns True and nothing else is done.
True	False	NOACTION	NOACTION	The function returns True and nothing else is done.
False	True	NOACTION	NOACTION	The function returns False and nothing else is done.
False	False	NOACTION	NOACTION	The function returns False and nothing else is done.
True	True	ABORT SKIP CANCEL	NOACTION	The user is prompted to select one of the 3 actions specified by OnTrue .
False	True	ABORT SKIP CANCEL	NOACTION	The function returns False and nothing else is done.
True	False	SKIP	NOACTION	The function takes the action SKIP automatically.
False	False	SKIP	NOACTION	The function returns False and nothing else is done.
True	True	NOACTION	ABORT SKIP CANCEL	The function returns True and nothing else is done.

Function result	PromptUser	OnTrue	OnFalse	Behavior
False	True	NOACTION	ABORT SKIP CANCEL	The user is prompted to select one of the 3 actions specified by OnFalse.
True	False	NOACTION	SKIP	The function returns True and nothing else is done.
False	False	NOACTION	SKIP	The function takes the action SKIP automatically.

The two cases highlighted above correspond to the default behavior of the most SPELL functions.

Please notice that, for any SPELL function returning a boolean value, SKIP and CANCEL actions have effect over the returned value (SKIP means True, whereas CANCEL means False). This has to be taken into account when automatic actions are programmed: it is possible to force the function to return True due to the automatic action SKIP, even though the original function result value was False. This kind of constructs can lead to very confusing code and should be used carefully.

The OnTrue and OnFalse modifiers are not to be confused with OnFailure. Meanwhile the formers **can** cause the function to prompt the user if configured to do so, OnFailure **will always prompt** the user since it is indicating an internal failure.

The following example shows the combination of OnFailure and OnFalse modifiers:

Example 23: reacting on function result

```
Function( arg,
        PromptUser=False,
        OnFalse=SKIP,
        OnFailure=ABORT|REPEAT|SKIP|CANCEL )
```

In the example, the function will take automatically the action SKIP if the result of the operation is False. If there is an internal failure, the user will be prompted to choose one of the options indicated by OnFailure, no matter what the value of PromptUser modifier is.

3.5 Special error handling

In some very special cases, the procedure developer may want to capture the operation error and process it 'manually' in the procedure code, instead of letting SPELL to prompt the procedure controller. In such cases, a special SPELL modifier may be used. This modifier is HandleError, and it accepts a boolean value. The default value for all SPELL function calls is True, which means that in case of a failure, the SPELL function will handle the error and will prompt the user to decide what to do next.

If this modifier is passed with value `False` and there is a failure in the SPELL function call, the error will be thrown outside the SPELL function and the procedure developer Python code shall capture it.

Example 24: special error handling

```
try:
    Function( arg, HandleError=False )
except DriverException,ex:
    <code to manage the failure>
```

IMPORTANT: notice that if error handling is disabled in a SPELL function, it shall be ensured that the error will be captured at the procedure level in any case. Otherwise, the error may reach the SPELL execution environment level leading the procedure execution to the aborted state immediately. This is an advanced aspect of the language and should not be used widely.

3.6 Silent execution

All SPELL functions send feedback to the procedure controller in the form of messages and item status notifications. These can be disabled for convenience by using the `Verbosity` and `Notify` modifier.

When `Notify` is set to `False`, the functions will not send any item information feedback to SPELL clients. The value of `Verbosity` is an integer. When given a value higher than the maximum verbosity configured in the system, the text messages will not appear in the SPELL clients. For example, a value of `999` will typically avoid text messages from being sent.

Example 25: silent execution

```
Function( args, Verbosity=999 Notify=False )
```

This feature may be handy when creating custom utility functions.

4 SPELL language services

4.1 Time management

The **TIME** data type is supplied to use and manage time variables supporting both absolute and relative times. **TIME** objects are initialized using strings. The following formats are accepted for absolute time input strings:

Time string	Examples
dd-mmm-yyyy [hh:mm[:ss]]	'23-May-2009' '10-Jan-2009 10:30' '15-Jun-2009 12:00:30'
yyyy-mm-dd [hh:mm[:ss]]	'2007-01-30' '2008-12-23 10:30' '2008-06-01 12:30:30'
dd/mm/yyyy [hh:mm[:ss]]	'21/10/2008' '21/10/2008 10:30' '21/10/2008 10:30:25'
dd-mmm-yyyy:hh:mm[:ss]	'21-Jan-2008:22:30' '21-Jan-2008:22:30:23'
yyyy-mm-dd:hh:mm[:ss]	'2008-01-23' '2008-01-23:10:30' '2008-01-23:10:30:25'
dd/mm/yyyy:hh:mm[:ss]	'21/05/2008' '21/05/2008:10:30' '21/05/2008:10:30:25'

And for relative times (notice the +/- symbols):

Time string	Examples
+ss.nnn -ss.nnn	+23.500 -23.500
+ddd hh:mm[:ss]	+001 23:10

Time string	Examples
-ddd hh:mm[:ss]	+001 23:10:30 -001 23:10 -001 23:10:30

Times are evaluated in UTC time.

In addition, the following constructs are available:

- `NOW` Evaluates to the current absolute time
- `TODAY` Evaluates to the current day at 00:00:00
- `TOMORROW` Evaluates to the next day at 00:00:00
- `YESTERDAY` Evaluates to the previous day at 00:00:00
- `HOURL` Relative time of 1 hour
- `MINUTE` Relative time of 1 minute
- `SECOND` Relative time of 1 second

The construct `NOW` is special: each time it is used in the procedure code, it is evaluated to the current time at that very moment. It means that, when used inside a loop, the `NOW` value will change in each iteration (this may lead to confusions).

Example 26: using NOW in a loop

```
x = NOW
for count in range(0,100):
    Display( x )
    # x will have a different
    # time on each iteration!
```

If a fixed `NOW` value has to be used inside a loop, it is recommended to store its value in a variable using the following construct:

Example 27: using NOW in a loop (2)

```
x = TIME(NOW)
for count in range(0,100):
    Display(x)
    # x will keep the same time on all iterations
```

All time objects provide a set of utility methods to access the time data:

- `abs` Obtain the absolute time in seconds
- `rel` Obtain the relative time in seconds
- `isRel` Check if the time object contains relative time
- `julianDay` Convert to Julian day
- `year` Obtain the year
- `month` Obtain the month
- `hour` Obtain the hour
- `minute` Obtain the minutes
- `second` Obtain the seconds

Simple arithmetic operations can be done with `TIME` objects:

- Add one time to another
- Subtract times
- Multiply relative times by integers

Finally, `TIME` objects can be converted to string representation by means of the `str` Python function.


4.1.1 Output format

When `TIME` objects are converted to string using the `str()` and `repr()` functions, the default format used is "dd-mmm-yyyy hh:mm:ss". Nevertheless this format can be modified by using the static function `fmt()`:

Example 28: modifying the output time format

```
TIME.fmt( '%d-%b-%Y %H:%M:%S' )
```

Note that this does not affect in any way the formats accepted as *input* to create new time objects. It is only applicable for the *output* of time string representations.

03 February, 2015	SPELL Language Reference	
Page 35 of 118	File: SPELL - Language Reference - 2.4.4.docx	

The string given to change the output format shall be in the syntax expected by the Python datetime infrastructure, say:

%d for days

%b for months in 3-letter representation

%m for months in 2-digit representation

%Y year in 4-digit representation

%H hours in 2-digit representation

%M minutes in 2-digit representation

%S seconds in 2-digit representation

The following example shows how to initialize and use time variables.

Example 29: time management

```
# Will create a time object with the given time
mytime1 = TIME('2006/04/02 20:32:34')
mytime2 = TIME('+00:00:30')

# Access the time in seconds
A1= mytime1.abs()           # Evaluates to 1144002754.0
A2= mytime2.rel()           # Evaluates to 30

# Using time info fields
B1= mytime1.julianDay()     # Evaluates to 92
B2= mytime1.year()          # Evaluates to 2006
B3= mytime1.month()         # Evaluates to 4
B4= mytime1.hour()          # Evaluates to 20
B5= mytime1.minute()        # Evaluates to 32
B6= mytime1.second()        # Evaluates to 34

# Arithmetic operations
t2 = NOW + 3*HOUR + 20*SECOND
t1 = TODAY + TIME('+00:30:00')
t = t2 - t1
mytime3=mytime1 + mytime2

# Converting time to string,
# Evaluates to the string '2006/04/02 20:32:34'
str(mytime1)
```

4.2 Telemetry values acquisition

When working with telemetry parameters, the usual way is to provide one of the following to the SPELL functions:

- The telemetry *parameter mnemonic*:

```
'PARAM123'  
'FORMAT COUNTER'
```

Notice that spaces are ignored. In the second example, the mnemonic is not 'FORMAT' but 'FORMAT COUNTER'.

- The telemetry *parameter mnemonic plus the parameter description*:

```
'T PARAM123 Param description'
```

Notice the 'T' at the beginning of the string. This is used to indicate SPELL that the parameter mnemonic corresponds to the next *word* after the 'T'. That is, PARAM123 is the mnemonic. Anything after it is taken as the parameter description, i.e. 'Param description'. The 'T' symbol is required to differentiate this parameter naming form from the one in the first example.

To acquire telemetry parameter values, the `GetTM` function is provided.

4.2.1 Acquiring engineering values

The function may be used as *is*, without using any modifier.

Example 30: obtain engineering TM values

```
variable = GetTM( 'TMparam' )
```

The call shown in the example would store in 'variable' the current TM value of the TM parameter 'TMparam'.

`GetTM` function can provide the current value of the parameter right away, or it can be configured to wait until the parameter is updated and then return that new value. This behaviour can be changed with the modifier `Wait`.

If `Wait=True` is used, the function will wait until an update for the parameter arrives. If `Wait=False`, the function will return the last recorded value right away. The following example shows how to use the `Wait` modifier:

Example 31: waiting for updates

```
GetTM( 'TMparam', Wait=True )
```

Whenever the `Wait=True` modifier is used, it is possible to specify a timeout for the operation. This time will be taken as the maximum amount of time to wait until the next parameter update arrives. If the time limit is reached and no update has arrived, the function will fail:

Example 32: waiting for updates with timeout

```
GetTM( 'TMparam', Wait=True, Timeout=1*MINUTE )
```

Notice that the `Timeout` shall be used in combination with `Wait=True`. It does not make sense to give a timeout for a non-blocking operation. If only the `Timeout` modifier is provided, it will be ignored. `Timeout` accepts integers/floats (as seconds), `TIME` objects and time strings.

4.2.2 Acquiring raw values

The usage is the same but including the modifier `ValueFormat` with the value `RAW`:

Example 33: acquisition of raw value

```
GetTM( 'TMparam', ValueFormat=RAW )
```

4.2.3 Extended information

By using the `Extended` modifier, the `GetTM` function will return a telemetry item object instead of the parameter value. This object is a Python class that provides more information about the telemetry parameter through its methods:

- `name()`: returns the parameter mnemonic.
- `description()`: returns the parameter description.
- `fullName()`: returns a combination of mnemonic and description.
- `raw()`: returns the raw value.
- `eng()`: returns the engineering or calibrated value.

- f) `status()`: returns the validity of the parameter.
- g) `time()`: returns the update time.
- h) `refresh()`: used to reacquire and update the information contained in the item.

Notice that the telemetry item provided by `GetTM` will NOT update automatically its values, validity, update time, etc. The `refresh()` method shall be called explicitly for this.

Example 34: acquisition extended information

```
item = GetTM( 'TMparam', Extended=True )
value = item.raw()
updateTime = item.time()
```

4.2.4 Possible failures

There are several cases where the `GetTM` function will fail and the user will be prompted for action:

- a) The requested TM parameter does not exist.
- b) There is no TM link established in the GCS.
- c) TM flow quality is not OK.
- d) TM parameter is invalid.
- e) When waiting for updates, the parameter sample does not arrive before the timeout.

4.2.5 Failure actions

By default the user will be prompted to select one of the following actions:

- a) `REPEAT`: repeat the TM parameter acquisition.
- b) `SKIP`: skip the acquisition and continue.
- c) `ABORT`: abort the procedure execution.

The actual default set of choices may vary depending on the language configuration files.

IMPORTANT: Notice that the `GetTM` function will return no value in this case! Use `SKIP` carefully since this may lead to a crash further in the procedure.

4.3 Verifying telemetry values

Although `GetTM` may be used for retrieving TM parameter values and then performing comparisons with them, SPELL language provides the `Verify` function to carry out this kind of operations in a more elaborated way.

The `Verify` function can compare TM parameter values against constants or other TM parameter values. As the `GetTM` function, it will handle any possible failure during TM value acquisitions.

Lists of several TM parameters may be passed to the function so that many comparisons can be carried out at the same time (in parallel).

4.3.1 Verifying a single parameter value

To verify the value of a single parameter, the following statement can be used:

Example 35: verify a single TM value

```
Verify( [ 'TMparam', eq, 'VALUE' ] )
```

The previous example will verify whether the current value of the TM parameter `'TMparam'` equals `'VALUE'` or not. The available comparison operators are the following:

- `eq` = equal to
- `ge` = greater than or equal to
- `gt` = greater than
- `lt` = less than
- `le` = less than or equal to
- `neq` = not equal to
- `bw` = between (ternary operator)
- `nbw` = not between (ternary operator)

Ternary operators require two values on the right side of the verification definition instead of one.

The `ValueFormat` modifier may be used in order to use the engineering or the raw value of the TM parameter for the comparisons (using `RAW` or `ENG` values for the modifier):

Example 36: verify a single TM value in raw format

```
Verify( [ 'TMparam', eq, 'VALUE' ], ValueFormat=RAW )
```

4.3.2 Last recorded values and timeouts

The modifiers `Wait` and `Timeout` are also applicable for the `Verify` function, having similar effect during the TM parameter value acquisition. By default, the `Verify` function will wait until the next parameter sample arrives before performing the comparison. If `Wait=False` is used in `Verify`, the last recorded values will be used for the comparison.

Example 37: verify using next update and timeout

```
Verify( [ 'TMparam', eq, 'VALUE' ],  
        Wait      = True  
        Timeout = 10 )
```

In the example above, the `Verify` function will wait at most 10 seconds for the next TM sample. If this sample cannot be acquired within that time, a failure will happen. As always, `Timeout` accepts integers, floats, `TIME` objects and time strings.

4.3.3 Tolerance

A tolerance value for the comparisons may be provided using the `Tolerance` modifier, in order to perform more flexible comparisons. `Tolerance` is applicable to numerical values only.

Example 38: verify with tolerance

```
Verify( [ 'TMparam', eq, 20.3 ], Tolerance=0.1 )
```

This comparison will be valid if the value of the TM parameter is between 20.2 and 20.4 inclusive.

4.3.4 Failed comparisons

`Verify` function returns a composite object which can be evaluated to `True/False`. That is, this object can be directly used in an `'if'` statement:

Example 39: using Verify in a test statement

```

if Verify( [...] ):
    ...
else:
    ...

```

Whenever one or more TM parameter verifications is evaluated to `False`, the object returned by `Verify` will evaluate to `False`. In this case, information about which TM conditions evaluated to false can be obtained through the composite object given by the function. This object can be manipulated as a Python dictionary to some extent. The following example shows the usage:

Example 40: getting failure information

```

result = Verify( [...] )

for tmParam in result.keys():
    Display( tmParam + ':' + str(result[tmParam]) )

```

The code above could produce the following output:

Example 41: getting failure information (output)

```

Param_1: True
Param_2: True
Param_3: False

```

Meaning that the verification failed because the third TM condition was evaluated to `False` (i.e. the TM parameter had not the expected value).

4.3.5 OnFalse for Verify function

When the result of a `Verify` call is `False` (i.e. the composite object evaluates to `False`), the function will prompt the user for an action since this is interpreted as a problem. `Verify` function is one of the SPELL functions whose `OnFalse` modifier value is not `NOACTION` but `ABORT` | `SKIP` | `RECHECK` | `CANCEL`.

There two ways of modifying this behavior. The first one is to assign `NOACTION` to `OnFalse` modifier in the `Verify` function call:

Example 42: do nothing on False

```
Verify( [ 'TMparam', eq, 'VALUE' ],
        OnFalse=NOACTION )
```

This will make the `Verify` function to return the composite object normally, but without prompting the user for actions. Another approach is to use the `PromptUser` modifier with value `False` to carry out an automatic action as it was explained in section 3.4. If this modifier is given to the function, the user will not be prompted and the execution will continue. Notice that `OnFalse` shall have a single action code:

Example 43: do not prompt the user and take the `SKIP` action

```
Verify( [ 'TMparam', eq, 'VALUE' ],
        PromptUser=False,
        OnFalse=SKIP )
```

If the verification fails in this example (which means that the condition is evaluated to false), the `SKIP` action is automatically taken. As a result, the `Verify` function will return `True` (since this is the value associated to `SKIP` action). To return `False` in case of verification failure, `CANCEL` action could have been chosen.

Notice that the composite object is no longer returned in case of carrying out automatic actions.

Function failures caused by an error in the parameter acquisition or by a failure in the GCS connection will be handled normally (the user will be prompted in any case with the `OnFailure` options).

4.3.6 Retrying verifications

The modifier `Retries` can be used as well to make SPELL repeat the TM value acquisition a given number of times if the comparison is `False`. In every retry, the *next parameter sample* is used, no matter what the value of the `Wait` modifier is.

Example 44: repeat verification if it fails, 2 times

```
Verify( [ 'TMparam', eq, 'VALUE' ],
        Retries=2 )
```

In the previous example, if the first comparison results on `False`, the operation will be repeated at most two times using the next TM sample (i.e. using `Wait=True`, no matter if the `Wait` modifier is passed to the function by the developer) until the comparison becomes `True`. If there is no successful comparison after the second repetition, the function fails. Giving a value of zero to `Retries` will result on no retries being done.

4.3.7 Delaying the verification

In some cases it may be interesting to wait a certain amount of time before actually performing the verification. This is done by means of the modifier `Delay`, which accepts an integer, float or time instance specifying this amount of time.

Example 45: delay the verification

```
Verify( [ 'TMparam', eq, 'VALUE' ],  
        Delay=2*MINUTE )
```

4.3.8 Case-insensitive verification

When verifying string values, it is possible to perform the checks ignoring the case of the values. To do so, the modifier `IgnoreCase` shall be given with a value of `True`.

Example 46: ignoring string value case

```
Verify( [ 'TMparam', eq, 'MyString' ],  
        IgnoreCase=True )
```

4.3.9 Function result

The return value of the `Verify` function is `True` when the comparison is successful. When the condition is evaluated to `False`, and no special modifier is provided, the user will be prompted for action, being able to choose one of the following:

- d) `SKIP`: ignore the `False` comparison and continue, giving `True` as return value.
- e) `RECHECK`: repeat the TM verification using the next TM parameter sample.
- f) `CANCEL`: ignore the false comparison and continue, giving `False` as return value.
- g) `ABORT`: abort the procedure execution.

This set of actions can be changed with the modifier `OnFailure`.

4.3.10 Verifying multiple conditions

Several TM conditions can be verified in parallel. To do so, the list of conditions has to be passed to the function `Verify`:

Example 47: multiple verifications

```
Verify( [ [ 'TMparam1', eq, 'VALUE1' ],
          [ 'TMparam2', neq, 'VALUE2' ],
          [ 'TMparam3', lt, 10.5 ] ] )
```

Notice the usage of a 'list of lists'. All mentioned modifiers are applicable for parallel verifications.

If using a modifier is required for one of the TM conditions and not for all of them, a configuration dictionary can be included *inside that condition*. In this way, the modifier will affect the associated condition only, not the rest.

Example 48: multiple verifications and configurations

```
Verify( [ [ 'TMparam1', eq, 'VALUE1' ],
          [ 'TMparam2', neq, 4, {ValueFormat:RAW} ],
          [ 'TMparam3', lt, 10.6 ] ] )
```

In the example, the second TM condition will be tested using the raw value of the parameter, and the engineering value will be used for the other two conditions. Notice that modifiers cannot be passed as keyword parameters in this case, but a Python dictionary shall be used.

On the other hand, global modifiers (that is, modifiers affecting the whole function call) can still be passed using keyword arguments as explained before:

Example 49: multiple verifications and configurations (2)

```
Verify( [ [ 'TMparam1', eq, 'VALUE1' ],
          [ 'TMparam2', neq, 4, {ValueFormat:RAW} ],
          [ 'TMparam3', lt, 10.5 ] ],
Timeout = 10 )
```

In the example, a timeout of 10 seconds will be used for ALL condition verifications. Notice that if the same modifier is used as a global modifier and inside a TM condition, the latter will override the global value for the corresponding condition evaluation.

4.3.11 Verifying against multiple parameter values

It is also possible, on each telemetry condition, to provide a list of several values to perform the comparison against. The evaluation of the condition will be carried out using each of the given values in sequence, continuing until the condition evaluates to True. If none of the given values satisfies the condition, it will evaluate to False.

To define a condition with multiple parameter values a list shall be used:

Example 50: verifications with multiple values

```
Verify( [ [ 'TMparam1', eq, ['VALUE1', 'VALUE2'] ],  
          [ 'TMparam2', neq, 4, {ValueFormat:RAW} ],  
          [ 'TMparam3', lt, 10.5 ] ] )
```

Note that the operators btw and nbw do NOT accept lists of values.

4.3.12 Comparing parameters against other parameters

The telemetry conditions can be defined using telemetry parameters in the value field as well. This allows the user to compare the values of two telemetry parameters.

Example 51: comparing telemetry parameters

```
TMparam2 = GetTM( 'TMparam2', Extended = True )  
Verify( [ [ 'TMparam1', eq, TMparam2 ] ] )
```

The telemetry conditions may be constructed using **strings** as expected values for telemetry parameters. Due to this, the SPELL telemetry conditions cannot be symmetrical. That is, the first element in a telemetry condition must be always a telemetry parameter item or a telemetry parameter name, whereas the last element in a condition can be a number, a string literal, or a telemetry parameter item.

Therefore it should be noted that the following example would NOT have the same effect as the previous one:

Example 52: comparing telemetry parameters (wrong code)

```
Verify( [ [ 'TMparam1', eq, 'TMparam2' ] ] )
```

In this case, we would be comparing the value of the telemetry parameter 'TMparam1' against the string literal 'TMparam2', and not the value of the telemetry parameter 'TMparam2'.

Nevertheless it would be possible to provide telemetry items to BOTH condition fields:

Example 53: comparing telemetry parameters with items

```
tm1 = GetTM( 'TMparam1', Extended=True )
tm2 = GetTM( 'TMparam2', Extended=True )

Verify( [ [ tm1, eq, tm2 ] ] )
```

this code would have the desired effect.

Finally, it should be noted that telemetry items can be also used within lists of expected values:

Example 54: comparing telemetry parameters with lists of items

```
tm1 = GetTM( 'TMparam1', Extended=True )
tm2 = GetTM( 'TMparam2', Extended=True )
tm3 = GetTM( 'TMparam3', Extended=True )

Verify( [ [ tm1, eq, [tm2,tm3] ] ] )
```

In this case the condition would be True if the telemetry parameter 'TMparam1' had the same value as the parameter 'TMparam2' or as the parameter 'TMparam3'.

4.3.13 Boolean expressions

A set of telemetry conditions in SPELL are evaluated as an "AND" combination by default. Therefore, in order to combine a set of conditions with AND only, just provide the conditions inside a Python list using the syntax described in section 4.3.10. As of SPELL 2.4, in order to create more complex expressions, the special constructs AND() and OR() can be used. These constructs accept telemetry conditions and themselves as arguments.

- The AND() construct combines a set of conditions that will be evaluated in parallel, and will evaluate to True if all the contained conditions evaluate to True.
- The OR() construct combines a set of conditions that will be evaluated in parallel, and will evaluate to True if ANY of the contained conditions evaluate to True.

The general syntax to create a boolean telemetry expression follows:

Example 55: building a simple command

```
expression = AND( <condition>, <condition>, ... )
expression = OR( <condition>, <condition>, ... )
```

The resulting `expression` object can be passed to a `Verify` function instead of the classical list of conditions. In this syntax, `<condition>` can be:

- Another `AND()` or `OR()` construct
- A classical telemetry condition [`<param>`, `<operator>`, `<value>`]

Some examples follow:

Example 56: building a simple command

```
[1] Verify( AND( ['tm1',eq,0], ['tm2',eq,0] ) )
[2] Verify( OR( ['tm1',eq,0], ['tm2',eq,0] ) )

[3] Verify( AND( ['tm1',eq,0] ,
                  OR( ['tm2',eq,-1],
                      ['tm2',eq,0 ]
                  )
            )
        )
```

In the code above,

- Example [1] is equivalent to a classical `Verify` call where the Python list is substituted by the `AND()` construct;
- Example [2] will evaluate to True when `'tm1'` has the value 0 **OR** `'tm2'` has the value 0.
- Example [3] will evaluate when `'tm1'` has the value 0 **AND** `'tm2'` has the value 0 **OR** -1.

4.4 Building commands

SPELL stores telecommand definitions as TC items. TC items are Python objects that contain all the parameters of a telecommand definition (name, argument names, argument values, etc). To build TC items, the `BuildTC` function is provided:

Example 57: building a simple command

```
tc_item = BuildTC( 'CMDNAME' )
```

In the example above, 'tc' will store the definition of the telecommand 'CMDNAME'. If the given command name does not exist or there is a problem when building the command definition, the function will fail and prompt the user to choose an action.

4.4.1 Command arguments

The `args` keyword is used for this matter. TC arguments are defined in a Python list of lists where argument names, values and modifiers are specified.

Example 58: command arguments

```
args= [ [ 'ARG1', VALUE1 ],
        [ 'ARG2', 0xFF, {Radix:HEX} ] ]
```

The following modifiers can be used:

- `ValueType`: LONG, STRING, BOOLEAN, TIME, FLOAT
- `ValueFormat`: ENG/RAW
- `Radix`: DEC/HEX/OCT/BIN

`ValueType` modifier can be used to explicitly cast the argument value to a given type. For example, if an integer is given as an argument value, SPELL will interpret it directly as a long. But, if the argument should be considered as a string, `ValueType=STRING` shall be used for that argument. `ValueFormat` is used to specify if the argument is given in engineering (default) or raw format. The `Radix` modifier indicates the radix of the value (default is decimal). The telecommand item is obtained then:

Example 59: building a command with arguments

```
tc_item = BuildTC( 'CMDNAME', args=[...] )
```

4.5 Sending commands

The `Send` function is provided for sending, executing and monitoring telecommands. The same function can be used for sending single commands, command sequences, or command lists (grouped or not, depending on the driver features).

4.5.1 Sending a command without arguments

This is the simplest case. To send the command, the following statement is used:

Example 60: sending a simple command

```
Send( command = 'CMDNAME' )  
  
Send( command = tc_item )
```

The keyword `command` is mandatory. It accepts a command name (string) or a command item. The given command will be sent, its execution will be monitored and the status will be reported to the user. If ever the command execution fails, the user will be prompted for choosing an action as usual.

The `Send` function will monitor command execution and will prompt the user in case of failure.

4.5.2 Sending a time-tagged command

A time-tagged command is a telecommand whose execution is **delayed on board** until a certain time arrives. To time-tag a command, the `Time` modifier shall be used. The value for this modifier can be a date/time string or a `TIME` object representing an absolute date.

Example 61: time-tagged commands

```
Send( command= 'CMDNAME', Time=NOW + 30*MINUTE )  
  
Send( command= 'CMDNAME', Time='2008/04/10 10:30:00' )
```

The passed time shall be absolute.

4.5.3 Release time for commands

A release time can be specified for telecommands. The GCS shall process the release time appropriately and release the command when it is desired, SPELL will only provide the release time via the driver. The release time implies that the command shall not be released from the GCS until a certain time arrives (**delayed on ground**). To specify the release time, the `ReleaseTime` modifier shall be used. The value for this modifier can be a date/time string or a `TIME` object representing an absolute date.

Example 62: commands with a release time

```
Send( command= 'CMDNAME', ReleaseTime=NOW+30*MINUTE )  
  
Send( command= 'CMDNAME',  
      ReleaseTime='2008/04/10 10:30:00' )
```

The passed time shall be absolute.

4.5.4 Loading but not executing

In some cases a TC has to be loaded on the S/C but the procedure developer does not want to execute it yet. For this matter, the `LoadOnly` modifier can be used:

Example 63: load only command

```
Send( command= 'CMDNAME', LoadOnly=True )
```

In this case SPELL considers the `Send` call successful as soon as the command is loaded on board. This feature is available for those GCS where dual-step commanding is possible.

4.5.5 Confirm execution of all commands

The procedure may force the user to confirm explicitly whether a given telecommand should be sent or not. To do so, the modifier `Confirm` shall be used with a value `True`. By default, command injections are not confirmed.

Example 64: confirm command execution

```
Send( command= 'CMDNAME', Confirm=True )
```

4.5.6 Confirm execution of critical commands

The procedure may force the user to confirm explicitly whether a given **critical** telecommand should be sent or not. Critical commands are identified as such in the GCS TM/TC database. To enable or disable the confirmation of critical commands injection, the modifier `ConfirmCritical` is used.

Example 65: confirm critical command execution

```
Send( command= 'CMDNAME', ConfirmCritical=True )
```

4.5.7 Sending a command with arguments

Arguments can be passed in two ways:

- Define a TC item including the argument definitions, and send the TC item
- Use the `args` keyword shall be used in the `Send` function, similarly as for `BuildTC` function

Example 66: commands with arguments

```
Send( command= tc_item ) # The item contains args  
  
Send( command= 'CMDNAME', args=[...] )
```

4.5.8 Sending sequences

Sequences are lists of commands that are defined at the level of the GCS TM/TC database. From the SPELL point of view, a sequence is just a single commanding entity that takes more time to verify, since it is composed of several sub-commands that shall be executed and verified.

To differentiate a command from a GCS sequence, the keyword argument `sequence` shall be used.

Example 67: sending sequence

```
Send( sequence = 'SEQNAME' )
```

4.5.9 Sending command groups

To send a command list or group, a list of TC names or items shall be passed and the `group` keyword argument is used.

Example 68: sending a group of commands

```
Send( group = [ 'CMD1', 'CMD2', 'CMD3' ] )
```

In the example above, the three commands will be sent and verified one by one, sequentially. If one of the commands fails, the sequence will be interrupted and the user will be prompted to choose an action.

When sending lists, it is not possible to pass all the command arguments using the `args` keyword. For this matter, TC items shall be used.

Example 69: sending a group of commands with arguments

```
Send( group = [ tc_item1,tc_item2,tc_item3 ] )
```

In the example, `'tc_item1'`, `'tc_item2'` and `'tc_item3'` are objects that contain the command definitions, including arguments and values. To build such objects, the `BuildTC` function is used.

4.5.10 Grouping and blocking command groups

If the feature is available on the SPELL driver (that is, depending on the GCS being used), it is possible to indicate that all the commands in the group shall be sent as a 'critical section'. This means that all commands of the group shall be sent one after the other and no other command coming from another source shall be able to be uplinked between two commands of the group. To specify that the list of commands should be taken as a group, the `Group` modifier is used with value `True`:

Example 70: sending a list of commands grouped

```
Send( group = [tc_item1, tc_item2, tc_item3],  
      Group=True )
```

A list commands can be sent as a *block* as well. When several commands are in a block, they are **(1)** grouped as explained above, and **(2)** uplinked to the spacecraft all together in the same encoded frame. To block a list of commands, the `Block` modifier is used:

Example 71: sending a list of commands blocked

```
Send( group = [tc_item1, tc_item2, tc_item3],  
      Block=True )
```

4.5.11 Timeouts for execution verification

The `Timeout` modifier may be used in order to specify the time window for the command execution. That is, the command execution shall be confirmed before the specified time:

Example 72: sending with timeout

```
Send( command = tc_item, Timeout=1*MINUTE )
```

When sending command lists, the timeout affects to each command in the list.

4.5.12 Platform specific parameters

The `addInfo` keyword argument can be used to pass additional information required by the GCS to inject a given command. This parameter accepts a Python dictionary containing any kind of data. The contents of this dictionary depend on the SPELL driver being used.

Example 73: additional information

```
Send( command = tc_item, addInfo={"any key":"any data"} )
```

4.5.13 Delaying command release time

The `SendDelay` modifier can be used to indicate the amount of time to wait before actually sending a command(s). This time delay is introduced by the GCS and not by SPELL: the command is actually injected into the GCS immediately, but it is the GCS who holds the command release for the specified amount of time.

Example 74: sending with delayed release

```
Send( command = tc_item, SendDelay=1*MINUTE )

Send( group = [tc_item1,tc_item2,tc_item3],
      SendDelay=1*MINUTE )
```

If a group of commands is used, the delay is applied to each command individually. Notice that the amount of time specified with `SendDelay` is automatically added to the command execution verification time window (for the given command only).

4.5.14 Sending And Verifying

The `Send` function allows creating a closed-loop operation consisting on sending one or more commands and then checking TM parameters to verify that the command executions have been correctly performed.

All that was said for `Send` and `Verify` functions applies here: all modifiers may be combined to create complex operations.

The simplest case is to send a simple command and then to check a TM parameter value. The `Send` function accepts all the arguments applicable to the `Verify` function. The only difference is that the TM verification list shall be identified with the `verify` keyword:

Example 75: sending and verifying

```
Send( command = tc_item, verify=[['TMparam',eq,10]] )
```

This statement will send the command, confirm its execution, and then check the given TM parameter condition. If the command execution fails, the user will be able to choose to resend the command or to skip/abort. If the TM verification fails, the user will be able to choose between repeating the whole operation (that is, resend the command and retry the verification), recheck the TM condition only, skip the whole operation, etc. To check more than one TM condition, a list of lists shall be used with the `verify` keyword.

4.5.15 Automatic limit adjustment

If the telemetry verification feature is used in a `Send` function call, an automatic limit adjustment can be done as well. This feature is activated when the `AdjLimits` modifier is used and set to `True`. Please refer to section 4.8.7 for details about limit adjustment using TM conditions (`AdjustLimits` function).

Example 76: sending, verifying and adjusting limits

```
Send( command = tc_item,  
      verify=[['TMparam',eq,10]],  
      AdjLimits=True )
```

4.5.16 A complex example

The following example shows a `Send` statement including several features in the same call.

Example 77: complex send

```
Send( command = 'TCNAME',

      args = [ [ 'ARG1', 1.0 ],
                [ 'ARG2', 0xFF, {Radix:HEX} ] ],

      verify=[ ['TM1',eq,10.0,{Tolerance:0.1} ],
                ['TM2',gt,0,{Timeout:20} ] ],

      Delay=10*SECOND,
      Tolerance=0.5,
      OnFailure=CANCEL,
      PromptUser=False
    )
```

This example code will:

- 1) Send the telecommand 'TCNAME' with the arguments ARG1 and ARG2 set to 1.0 and 0xFF
- 2) Once the telecommand execution is verified,
- 3) Wait 10 seconds (Delay modifier)
- 4) Then verify the two given telemetry conditions, the first one with tolerance 0.1, the second with tolerance 0.5 (notice the global Tolerance modifier)
- 5) The first TM condition will be verified with the default timeout, but the second one will use a time window of 20 seconds.
- 6) If ever the operation fails, the user will not be prompted (PromptUser=False) and the function will return False (OnFailure=CANCEL)

4.5.17 Return value

The `Send` function returns `True` unless there is a command failure.

4.6 Holding execution

There are cases where the procedure execution shall be paused for a certain amount of time. The `WaitFor` function allows stopping the procedure execution until a condition is satisfied. This condition may be specified in terms of time, or using telemetry conditions.

4.6.1 Stop procedure execution for a given amount of time

To stop procedure execution for some time:

Example 78: wait relative time condition

```
WaitFor( 2 )  
  
WaitFor( 2*SECOND )  
  
WaitFor( '+00:00:02' )
```

The previous example will wait for two seconds in all the cases; the execution will be resumed afterwards. There are some assumptions regarding the function arguments:

- If an integer or float number is given, the value will be interpreted as an amount of seconds.
- If a string is given, the value will be interpreted as a date/time string
- If a `TIME` object is given, the corresponding absolute time in seconds will be used.

Notice that date/time strings and time objects should represent relative times and not absolute times. If the given time is absolute, the behavior of the `WaitFor` function is different. If the given parameter cannot be interpreted as a time value, the function will fail.

4.6.2 Stop procedure execution until a given absolute time arrives

`WaitFor` function may be used for stopping execution until a given time is reached. To do so, absolute times are used:

Example 79: wait absolute time condition

```
WaitFor( '2008/10/03 10:45:34' )
```

4.6.3 Stop procedure execution until a telemetry condition is fulfilled

Execution may be put on hold until a given telemetry parameter satisfies a given condition. That is, `WaitFor` function may accept the same arguments as `Verify` function. The behavior will be to wait until all the passed verification conditions are fulfilled.

Example 80: wait telemetry condition

```
WaitFor( ['TMparam', eq, 23 ] )
```

The example above will wait until the given condition is `True`. All modifiers applicable to the `Verify` function are applicable here. For example, `ValueFormat` modifier could be used to perform the verification check using the raw value of the parameter.

The `Delay` modifier, in this function, allows specifying that the function should fail if the TM condition is not fulfilled before a particular time limit:

Example 81: wait telemetry condition with maximum delay

```
WaitFor( ['TMparam', eq, 23 ], Delay=20 )
```

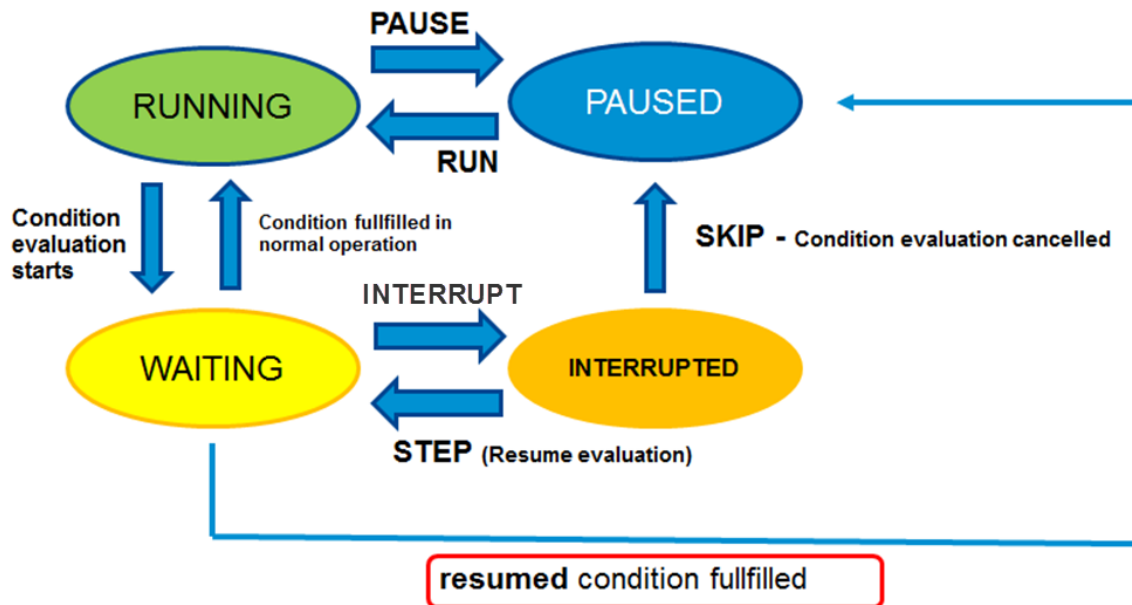
In the example above, the function will fail if `TM1` does not have the value 23 before 20 seconds. The `Delay` modifier should not be confused with `Timeout` modifier: the latter is used to specify the maximum time to *acquire* the TM value from the GCS, not the maximum time to reach the desired value (23).

4.6.4 Aborting the wait

The execution of the `WaitFor` statement can be interrupted by using the `INTERRUPT` command button meanwhile the system is waiting for the condition to be fulfilled. This can be done regardless the type of `WaitFor` statement (based on time or telemetry). When the statement gets interrupted, the procedure reaches the `INTERRUPTED` state. The telemetry checks or the time countdown are then put on hold. From this state, the following operations can be performed:

- Resume the countdown or telemetry check: use the `STEP` command.
- Abort the countdown or telemetry check: use the `SKIP` command.

Note that when a `WaitFor` statement is interrupted, whatever the following actions taken by the user are, the procedure will end at the end in `PAUSE` state.



If the procedure is put to **WAITING** state again the condition check will be resumed. In the case of a time condition, note that the original target time (regardless the condition being based on absolute or relative time) will not change.

4.6.5 Showing progress in time conditions

Interval and **Message** modifiers can be used to set update intervals for the remaining time when time conditions are used. The **Interval** modifier specifies the pattern of the updates, and **Message** defines the message to be shown to the user on each update.

The following example will show the specified message once per minute until the time condition is fulfilled:

Example 82: simple interval

```

WaitFor( 1*HOUR,
        Interval=1*MINUTE,
        Message='Hi there')
  
```

The remaining time is also shown alongside each update message. To specify a more complex interval, a list of times can be used:

Example 83: complex interval

```
WaitFor( 5*HOUR,
        Interval=[1*HOUR, 5*MINUTE, 1*SECOND]
        Message='Hi there')
```

In the previous example, the update will be done (and the message will be shown):

- Once per hour, until one hour remains; then
- Once each 5 minutes, until 5 minutes remain; then
- Once per second until the time limit is reached.

4.7 Injecting TM parameters

The `SetGroundParameter` function allows injecting values for existing ground TM parameters into the GCS:

Example 84: injecting parameters

```
SetGroundParameter( 'PARAM', VALUE )
```

The example above would inject the value `'VALUE'` for the parameter `'PARAM'` on the GCS. To retrieve ground parameter values, `GetTM` function shall be used.


4.8 Getting and setting TM parameters limits

The functions `GetLimits` and `SetLimits` allow getting and setting TM parameter Out-of-limits (OOL) definitions. The functions `IsAlarmed`, `EnableAlarm`, and `DisableAlarm` allow procedures to check the alarm status of TM parameters and to enable or disable the limit checking for a given TM parameter.

4.8.1 Identifying limits

One TM parameter may have several limit definitions, not all of them necessarily applicable at the same time. In order to differentiate them, each definition shall have an associated string identifier; the actual identifier used depends on the SPELL driver implementation, no restriction is imposed by the SPELL language.

To read or manipulate limits, the particular limit definition to be modified shall be specified. This is done by means of the `Select` modifier. Possible values for this modifier are `ALL` (modify/get all available definitions), `ACTIVE` (get/modify the currently active definition) or a limit definition string identifier.

03 February, 2015	SPELL Language Reference	
Page 60 of 118	File: SPELL - Language Reference - 2.4.4.docx	

4.8.2 Types of limits

SPELL recognizes the following limit types:

- **Hard-Soft:** defined by a list of four values. Two values define a hard (outer) limit, and the other two define a soft (inner) limit. The TM parameter value shall remain between these limits in order to be in nominal state.
- **Status:** for digital or status parameters, defined by a list of parameter values and associated status.
- **Step:** defined by two values, detect if there is a step up/down in the TM parameter value bigger than the two deltas.
- **Spike:** defined by a delta value, detect if there is a change in the TM parameter bigger than the delta.

Note that the SPELL drivers are not obliged to implement and recognize all these limit types. As with all other SPELL language features, only a subset of the limit management model can be implemented.

4.8.3 Limit specification

Limits are read and modified in the form of dictionaries with modifiers:

- **Hard-Soft:** the dictionaries are like { `LoRed:x`, `LoYel:x`, `HiYel:x`, `HiRed:x` } where x are numerical values. H-S limits can be also specified by { `Midpoint:x`, `Tolerance:x` } still resulting on a four values definition, with yellow and red limits being equal to each other. If any of the values is not available in the GCS the key will not be available in the dictionary.
- **Status:** the dictionaries are like { `Nominal:['A','B']`, `Warning:['C']`, `Error:['D']`, `Ignore:['E']` } where 'A', 'B', etc. are TM parameter values. The modifiers `Nominal`, `Warning`, `Error` and `Ignore` represent all the status recognized by SPELL. Any other status supported by GCS shall be mapped by SPELL drivers to one of these, or just ignored. If any of the states has no values the state will be returned with an empty list.
- **Step:** the dictionary includes a modifier with a list with two numeric values { `Delta:[x1, x2]` }.
- **Spike:** the dictionary has the modifier { `Delta:x` } where x is a numeric value.

4.8.4 Retrieving OOL definitions

To retrieve all the existing limit definitions of a TM parameter:

Example 85: reading all limit definitions

```
GetLimits( 'PARAM', Select = ALL )

→{'ID1': {LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4,
          Hysteresis:2, Active:True, Name:'ID 1 Def',
          Algorithm: 'TM1<TM2+TM3', Type=HARDSOFT },
   'ID2': {LoRed:y1, LoYel:y2, HiYel:y3, HiRed:y4,
          Hysteresis:2, Active:True, Name:'ID 2 Def',
          Algorithm: 'TM1<TM2+TM3', Type=HARDSOFT }}
```

In the example above, two definitions (with names "ID1" and "ID2") of an analog parameter are provided. In the dictionary shown, only the numerical parameters of the limit definition are displayed; nevertheless SPELL provides support for all the following limit characteristics (by means of the modifiers indicated):

- **LoRed, HiRed:** used for lower and upper, alarm-type or hard limits for an analog parameter. Accept numerical values (integer or float)
- **LoYel, HiYel:** used for lower and upper, warning-type or soft limits for an analog parameter. Accept numerical values (integer or float)
- **Hysteresis:** used to indicate the maximum number of limit violations that need to occur before raising a warning or alarm. Accepts integer numbers.
- **Active:** indicates if the given limit definition is active or not. Accepts True/False.
- **Name:** indicates a description or code name for the limit definition, if there is any. Accepts strings.
- **Nominal, Warning, Error, Ignore:** used for status parameters only, these modifiers indicate which of the possible values of the telemetry parameter are to be considered nominal values, warning values, etc. Accept strings or list of strings.
- **Algorithm:** used to provide a string representation of the activation algorithm of the limit definition, if there is any.
- **Type:** indicates the type of limit: HARDSOFT, STATUS, SPIKE, STEP.

Note that a dictionary with all the limit information is retrieved in the above case, including all the possible definitions available. Each dictionary key corresponds to the limit definition identifier. The actual identifiers used are free to choose and depend on the SPELL driver implementation (can be strings or numbers). Nevertheless it is assumed that the keys used in these dictionaries are consistent with the values provided to the Select modifier, as seen in the next paragraphs.

Note as well that the dictionary value for a given definition is another dictionary, whose keys are the different limit values (LoRed, LoHigh, etc).

In case that the parameter has not limit definitions, an exception will be raised and the user will receive a prompt.

Examples of limit definition dictionaries with all the possible modifiers explained are shown below:

Example 86: complex limit definitions for HARDSOFT limit

```
'ID1': { LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4,
        Hysteresis:2, Active:True, Name:'ID 1 Def',
        Algorithm: 'TM1<TM2+TM3', Type=HARDSOFT }
```

Example 87: complex limit definitions for STATUS limit

```
'ID1': { Nominal:['ENABLED'], Warning:['DISABLED'],
        Error:['FAILED', 'UNKNOWN'], Ignore:[],
        Hysteresis:1, Active:True, Name:'ID1 Status check',
        Type=STATUS}
```

Instead of retrieving all existing definitions, it may be preferable to retrieve the currently applicable (active) definition(s) only:

Example 88: reading applicable limit definitions

```
GetLimits( 'PARAM', Select = ACTIVE )
→{"ID1": { LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4,
          Hysteresis:2, Active:True, Name:'ID 1 Def',
          Algorithm: 'TM1<TM2+TM3', Type=HARDSOFT }}
```

In the example above "ID1" is the active definition at the moment of the call. In some cases there could be more than one definition active at a time. So, a dictionary of definitions is provided.

Finally, to retrieve a definition in particular:

Example 89: reading limit definitions by identifier

```
GetLimits( 'PARAM', Select = 'ID1' )
→{ LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4, Hysteresis:2,
   Active:True, Name:'ID 1 Def', Algorithm: 'TM1<TM2+TM3',
   Type=HARDSOFT }
```

Notice that since a specific definition is requested, only this definition is provided, not a dictionary of definitions.

As explained above, acquiring limit definitions for status parameters would result on dictionaries containing the Nominal, Error, Warning and/or Ignore modifiers.

Example 90: reading limit definitions from status parameters

```
GetLimits( 'STATUS_PARAM', Select = 'ID1' )
→ { Nominal:[v1,v2], Ignore:[v3], Error:[v5,v6,v7],
    Warning:[v8], Hysteresis:2, Active:True, Name:'ID 1 Def',
    Algorithm: 'TM1<TM2+TM3', Type=STATUS }
```

If only one of the values of a given OOL definition is to be retrieved, this can be indicated with the `Value` modifier:

Example 91: reading single data from limit definitions

```
GetLimits('PARAM', Select = 'ID1', Value=Error )
→ [v5,v6,v7]
```

As well:

Example 92: reading single data from limit definitions (2)

```
GetLimits('ANALOG', Select = 'ID1', Value=LoRed )
→ 0.567
```

Finally, for midpoint/tolerance and deltas can be used:

Example 93: reading single data from limit definitions (3)

```
GetLimits('ANALOG', Select = 'ID1', Value=Delta )
→ 0.0001

GetLimits('ANALOG', Select = 'ID1', Value=Midpoint )
→ 10.0

GetLimits('ANALOG', Select = 'ID1', Value=Tolerance )
→ 0.5
```

The type modifier will only be available when a limit definition ID is provided by the Select modifier. If no limit definition ID is provided an exception will be raised and the user will receive a prompt.

4.8.5 Modifying OOL definitions

The `SetLimits` function is used to modify the OOL definitions. First, the definitions to be modified shall be obtained with `GetLimits` (see previous section). The obtained definitions can be manipulated and then passed to `SetLimits` function:

Example 94: modifying limit definitions

```
LD = GetLimits('PARAM', Select=ACTIVE )
LD['ID1'][LoRed] = 0.5
SetLimits( 'PARAM', Limits=LD )
```

In the example above, the `LoRed` value of the definition "ID1" is modified. Another approach would be:

Example 95: modifying limit definitions (2)

```
LD = GetLimits('PARAM', Select=ACTIVE )
mydef = LD['ID1']
mydef[LoRed] = 0.5
SetLimits( 'PARAM', Definition=mydef, Select='ID1' )
```

Notice that in this second case a definition is passed to the function in spite of a dictionary of definitions. The definition to be modified has to be identified with `Select` modifier. If no definition identifier is provided, the SPELL driver shall modify all the ACTIVE OOL definitions for the TM parameter with the values on the definition dictionary provided:

Example 96: modifying limit definitions (3)

```
mydef={LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4}

SetLimits( 'PARAM', Definition=mydef, Select=ACTIVE )
OR
SetLimits( 'PARAM', Definition=mydef )
```


Notice that the `Select` modifier gets the `ACTIVE` constant as value by default, that is, when the modifier is not provided.

In the case of status parameters the keys of the dictionary are `Nominal`, `Warning`, `Error` and `Ignore`:

Example 97: modifying limit definitions (4)

```
LD = GetLimits('PARAM', Select=ACTIVE )
mydef = LD['ID1']
mydef[Nominal] = ['VALUE']
SetLimits( 'PARAM', Definition=mydef, Select='ID1' )
```

The limit definition will be updated and sent with `SetLimits` to the GCS. In the example above the `Nominal` states will be changed to the state `VALUE`. The states will depend on each parameter.

Modifying single datum in a limit definition

The appropriate modifier is used to indicate which parameter to change:

Example 98: modifying limit data for Status parameter

```
SetLimits( 'PARAM', Nominal=['V1', 'V2'], Select='ID1' )
```

The example above will assign the states `V1` and `V2` to the `Nominal` status.

Note that if no definition name ("ID1" in the example) is given, the change will be applied to all the `ACTIVE` limit definitions for the parameter.

Another example using middle-points:

Example 99: modifying limit definitions (5)

```
SetLimits( 'PARAM', Select='ID1',
          Tolerance=0.5, Midpoint=10.0 )
```

Spike limits will be provided by the `Delta` modifier with a number value, as follow:

Example 100: modifying limit definitions for SPIKE limit

```
SetLimits( 'PARAM', Select=ALL, Delta=0.23 )
```

The `Select=ALL` modifier will change the `Delta` to all the limit definitions for the parameter (active or not).

For a Step limit the `Delta` modifier will have a list of two numbers as a value:

Example 101: modifying limit definitions for STEP limit

```
SetLimits( 'PARAM', Select=ACTIVE, Delta=[0.23, 0.30] )
```

4.8.6 Enabling and disabling limit checking

The functions `EnableAlarm` and `DisableAlarm` are provided to easily enable or disable the out-of-limits alarms for a given parameter:

Example 102: enabling and disabling alarms

```
EnableAlarm( 'PARAM' )  
  
DisableAlarm( 'PARAM' )
```

4.8.7 Adjusting limits

The function `SetLimits` can change the OOL definitions of a given parameter with a set of TM conditions as the ones used for `Verify` function. The limit values will be adjusted to match the given set of values in order to prevent the GCS from raising alarms.

`SetLimits` will use the value of each TM condition and the tolerance (the one given or the default one) as the *midpoint and tolerance* of the limit definition.

Only the TM conditions using the equality operator are taken into account.

Example 103: automatic limit adjustment for HARDSOFT parameter

```
AdjustLimits( [['PARAM', eq, 3.4, {Tolerance:0.1}]] )
```

The example above will adjust the limits of the TM parameter `'PARAM'` to the values `Midpoint=3.4`, `Tolerance=0.1` which will result in the already described Hard-Soft limit definition.

Example 104: automatic limit adjustment for STATUS parameter

```
AdjustLimits ( [['PARAM2', eq, ['A']] ] )
```

For a Status parameter the Nominal Status will be updated adding the provided state value to the current state list in Nominal and will remove the new state from the previous Status, i.e. `Error`, `Warning` or `Ignore`.

4.8.8 Loading limits from files

If the GCS supports it, a OOL definitions file name can be specified via the `SetLimits` function. This will make the SPELL driver to request the GCS to load that file:

Example 105: loading limits from file

```
LoadLimits ( 'limits://name_of_file' )
```

4.8.9 Restoring limits

If the GCS supports it, all the modifications of OOL definitions performed by the procedures can be reset, bringin the OOL database back to its original state. This is achieved by using the following function:

Example 106: restoring limits

```
RestoreNormalLimits()
```

4.8.10 Checking alarm status

The function `IsAlarmed` can be used to check if a TM parameter is out of limits:

Example 107: checking alarm status

```
True/False = IsAlarmed( 'PARAM' )
```

4.9 Displaying messages

The `Display` function may be used to show messages to the user. `Display` will send messages to the SPELL clients with different format depending on the `Severity` modifier, which may take the values `INFORMATION`, `WARNING` and `ERROR`. The default value is `INFORMATION`.

Example 108: display messages

```
Display( 'Message' )

Display( 'Message', WARNING )

Display( 'Message', Severity = ERROR )
```

The `Severity` modifier is not mandatory but it may be recommendable to use it for clarity in some situations. `Display` will accept any string and does not return values.

The `Notify` function can be used to send NAME/VALUE notifications to the GUI, that will be shown in the Code or Tabular view. These notifications are similar to regular messages, but are typically linked to a concrete piece of procedure information (one function call, one telemetry item, etc.) and are only shown in the aforementioned view.

Example 109: sending notifications

```
Notify( 'Name', 'Value', <status> )

<status> = ITEM_SUCCESS, ITEM_PROGRESS, ITEM_FAILED
```

4.10 Injecting events

The function `Event` is quite similar to `Display` function, although the messages injected with it are displayed on the GCS event subsystem as well as in the SPELL clients.

Example 110: event messages

```
Event ( 'Message' )  
  
Event ( 'Message', WARNING )  
  
Event ( 'Message', Severity = ERROR )
```

4.11 Ground control system configuration and variables

The functions `GetResource` and `SetResource` allow getting and setting GCS system variables and configuration parameters. The actual scope of these modifications and which parameters can be utilized depend on the SPELL driver and the GCS being used. To get or set a configuration parameter value:

Example 111: change configuration value

```
SetResource ( 'Variable', Value )
```

Example 112: retrieve configuration value

```
value = GetResource ( 'Variable' )
```

Both functions will fail if the given variable does not exist or cannot be written or read. The Ground Database (see section 4.15) is normally used to map GCS-specific variable names to unified/standard variable names, which are the ones used in the SPELL procedures. This way, SPELL procedures are kept independent from the GCS.

Example 113: variable mappings with ground database

```
SetResource ( GDB['DECODER'], GDB['DECODER 1'] )
```

In the previous example, the GCS variable storing the current S/C decoder to be used is modified to store the value corresponding to the Decoder 1. Notice that, thanks to the GDB database the variable name and value are mapped to the generic names `'DECODER'` and `'DECODER 1'`.

4.12 Requesting Information

The `Prompt` function is used to display a message and request some input from the user. There are several types of prompt available:

- `OK`: the user may choose 'Ok' only as an answer
- `CANCEL`: the user may choose 'Cancel' only as an answer
- `OK_CANCEL`: the user may choose 'Ok' or 'Cancel' as an answer.
- `YES`: the user may choose 'Yes' only.
- `NO`: the user may choose 'No' only.
- `YES_NO`: the user may choose 'Yes' or 'No'
- `ALPHA`: the user may give any alphanumeric answer
- `NUM`: the user may give any numeric answer
- `DATE`: the user may give a date as an answer
- `LIST`: a custom list of options is provided to the user for choosing

These types are specified with the modifier `Type`. If only the message and the prompt type are given, the `Type` modifier is implicit for the second argument:

Example 114: types of prompt

```
Prompt( 'Message', OK )
```

A custom list of options can be used with the type `LIST`.

There are some possible combinations for this type of prompt:

- Give a set of key:value pairs. The user chooses one of the values, the corresponding key is returned as a result. This is the default behavior.

Example 115: default prompt list

```
# Returns 'A' or 'B':  
Prompt( 'Message',  
        ['A:Option 1', 'B:Option 2'],  
        Type=LIST )  
  
# Returns '1' or '2':  
Prompt( 'Message',  
        ['1:Option 1', '2:Option 2'],  
        Type=LIST )
```

- Give a set of values, and use the type `LIST|NUM`. The user chooses one of the values, and the corresponding option *index* is returned.

Example 116: custom list with index

```
#Returns 0 or 1  
Prompt( 'Message',  
        ['Option 1', 'Option 2'],  
        Type=LIST|NUM )
```

- Given a set of key:value pairs, and use the type `LIST|ALPHA`. The user chooses one of the values, and the same value is returned.

Example 117: custom list with values

```
#Returns 'Option 1' or 'Option 2'  
Prompt( 'Message',  
        ['Option 1', 'Option 2'],  
        Type=LIST|ALPHA )
```

A default value may be set for a prompt function with the `Default` modifier. This modifier is taken into account only if a `Timeout` for the prompt is also specified. The timeout gives the maximum time for the user to give an answer: if the time limit is reached, the default value is returned.

Example 118: default value for prompt

```
Prompt( 'Message',
        ['A :Option 1', 'B :Option 2'],
        Type=LIST, Default='A', Timeout=1*MINUTE )
```

If the `Timeout` modifier is used but a `Default` value is not given, the prompt function will make the SPELL clients to play a 'beep' sound if ever the prompt function is waiting for an answer and the operator is not giving it before the timeout. Also if `Timeout` modifier value is 0, the prompt will wait until an answer from the operator arrives.

4.13 Managing GCS system displays

4.13.1 Opening displays and workspaces

The `OpenDisplay` function allows opening TM displays at the target system:

Example 119: opening displays

```
OpenDisplay( 'Display name' )
```

It is assumed that the ground control system supports the telemetry displays through a dedicated application. SPELL also foresees the need to open simple displays, and also sets of displays or "workspaces". The concept of workspace allows a procedure to simultaneously load several displays, instead of using several subsequent calls to `OpenDisplay`. To open a workspace, the function `OpenWorkspace` shall be used:

Example 120: opening a workspace

```
OpenWorkspace ( 'Workspace name' )
```

It is assumed that the target system has defaults for the workstation and monitor where the TM display application should be open, therefore nothing else shall be provided from the procedure. Nevertheless the ability to explicitly select these parameters is also provided by the SPELL language, by mean of the modifiers `Host` and `Monitor`:

Example 121: specifying the host

```
OpenDisplay( 'Display name', Host='hostname' )
OpenWorkspace( 'Workspace name', Host='hostname' )
```

Example 122: specifying the monitor

```
OpenDisplay( 'Display name', Monitor=1 )
OpenWorkspace( 'Workspace name', Monitor=1 )
```

Both modifiers can be used together.

The value given to the `Monitor` modifier is actually free and only driver-dependent. The same happens with the display names, in the sense of that the string given can contain -or not- folder names, or any other structures that help the driver to identify the concrete display to be open. The contents of the first string argument are not constrained by SPELL.

The modifier `Reuse` (which accepts `True` or `False`) can be used to indicate whether the target system should open a new instance of the TM display application (`Reuse=False`) or it should just open the indicated display on the TM display application instance that is already running (if any).

Finally, when opening displays of type TM plot, it is also possible to indicate the initial time span of the plot by using the modifier `Time`:

Example 123: specifying time span

```
OpenDisplay( 'Display name', Time=1*HOUR )
```

4.13.2 Printing displays

The `PrintDisplay` function allows printing TM displays at the target system:

Example 124: printing displays

```
PrintDisplay( 'Display name', Printer='name' )
```

The `Printer` modifier specifies the printer name to be used. The `Format` modifier allows generating either Postscript (with value `'PS'`) or ASCII (value `'VECTOR'`, available only for alphanumeric displays):

Example 125: display format

```
PrintDisplay( 'Display name', Format=VECTOR )
```

The `PrintDisplay` function does not support workspace names.

4.13.3 Closing displays and workspaces

The `CloseDisplay` function allows the procedure closing displays on the TM display application of the target system:

Example 126: closing displays

```
CloseDisplay( 'Display name' )
```

Similarly as for the `OpenDisplay` function, the `Host` modifier can be used to identify the workstation where the TM display application of the target system is running.

To close a set of displays, or workspace:

Example 127: closing workspaces

```
CloseWorkspace( 'Workspace name' )
```

4.14 Procedure execution flow control

As it was explained before SPELL accepts all the typical Python control structures:

- `if / elif / else` clauses
- `for, for ... in` clauses
- `while, while ... in` clauses

In addition to this, SPELL provides the *Goto-Step* mechanism.

4.14.1 Steps

The `Step` function allows indicating the beginning of a procedure operation stage. Once it is executed, the current step title is shown on the SPELL clients. The `Step` function accepts an identifier and a title string:

Example 128: step definitions

```
Step( 'A1', 'Title of the step' )
```

On the previous example, when the `Step` function call is executed, the string 'Title of the step' is shown on the SPELL client applications.

IMPORTANT: the first argument of the `Step` instruction shall be a *string literal*, and not a variable or the result of a function. This is so, because the go-to mechanism relies on these literal values at compilation time. On the other hand, the step description (second argument) is allowed to be a variable.

4.14.2 Go-to mechanism

SPELL provides a special implementation of go-to mechanism based on the `Step` and the `Goto` functions. `Step` function calls can be used as possible target points for a `Goto` call.:

Example 129: step and go-to

```
Step( 'A1', 'Title of the step' )
...
...
...
Goto( 'A1' )
```

There are some rules to take into account when using the go-to mechanism:

- It is not possible to jump to a different scope, e.g. jumping to outside a function or jumping within a function, jumping inside a `for` loop or a `while` loop, etc.
- Caution shall be taken when using go-to, since a wrong usage may lead to
 - o Undefined variable errors
 - o 'Spaghetti code' syndrome, making the procedure hard to read and maintain

The go-to mechanism is provided in order to enable procedure developers to maximize the one-to-one correspondence with the original paper procedures. That does not mean that `Goto` should be widely used all along a procedure; Python provides several flow control clauses that can do the work better than a go-to.

Changing the Step information displayed

As it has been explained, a `Step` instruction can be used as a target for a `Goto` instruction. But when it is desired to change the step information displayed on the client GUI, without establishing a jump point in the procedure that can be used with a `Goto`, the instruction `DisplayStep` can be used:

Example 130: displaying step information

```
DisplayStep('A1', 'Aborting the procedure')
```

This instruction will show the step information in the graphical interface, but it cannot be used as a go-to target. Both arguments of this function can be variables or result of function calls.

4.14.3 Ending, pausing and aborting

The procedure execution can be paused or aborted programmatically by means of the functions `Pause` and `Abort`. It is also possible to successfully finish the procedure execution (giving no errors) with the function `Finish`.

Both `Abort` and `Finish` accept an optional message argument that is shown on the GUI when the procedure is aborted or finished respectively.

Example 131: pausing, aborting and finishing

```
Pause()  
  
Abort('Aborting the procedure')  
  
Finish('Procedure finished successfully')
```

4.14.4 User action function

The system allows the procedure to program a special Python function called "user action". When the user action is set up from the procedure, a new action appears in SPELL clients, allowing the users to trigger the execution of the indicated function at any time during the execution of the procedure.

The `SetUserAction` function is used to enable this mechanism:

Example 132: setting user actions

```
SetUserAction(function, 'Label')  
SetUserAction(function, 'Label', Severity=WARNING)
```

The 'function' argument shall be the name of a Python function already declared in the procedure. The label is the text that SPELL clients will show to users to identify the action.

In the case of the SPELL GUI, when an user action is set up, a new button will appear in the procedure view. The label of the button is the label indicated in the `SetUserAction` argument.

It is also possible to categorize the severity or criticality of the action by using the `Severity` modifier. This severity level will have different effect depending on the SPELL client; in the case of the SPELL GUI, it will determine the colour of the action button (normal, yellow or red).

When an action is first set up with `SetUserAction`, the associated trigger (e.g. a button) on the SPELL client is enabled. Nevertheless, it is possible to enable and disable this trigger on demand from the procedure by using the functions `EnableUserAction` and `DismissUserAction`:

Example 133: enabling/disabling user actions

```
EnableUserAction()  
DisableUserAction()
```

These functions take no arguments since there can be *only one user action configured at a time*. Finally, the procedure can permanently remove a configured user action by using `DismissUserAction` function:

Example 134: removing user actions

```
DismissUserAction()
```

By using this function, the user action trigger will disappear from SPELL clients.

4.15 Databases

SPELL procedures are able to access different configuration data needed for the execution:

- Spacecraft-specific data
- Manoeuvre data
- Ground system data
- User data

These data are stored in databases. A SPELL database is, from the procedure point of view, a normal Python dictionary: a collection of key/value pairs.

4.15.1 Spacecraft database

The spacecraft database provides all configuration parameters that are S/C specific. As it has been said, the same SPELL procedure can be used with several spacecraft of the same bus type. The differences between them (name, location, technological values, optional components and other characteristics) are stored in the spacecraft database.

The procedure code can access the spacecraft database through the global dictionary `SCDB`. This dictionary is used as follows:

Example 135: spacecraft database

```
sat_name = SCDB['SC']
```

It is also possible to iterate over the database keys by using the `keys` method:

Example 136: spacecraft database keys

```
for key in SCDB.keys(): Display( key )
```

Key existence can be checked with the `has_key` function:

Example 137: checking spacecraft database keys

```
if SCDB.has_key('Spacecraft_Name'):
    ...
```

The `SCDB` object is loaded automatically at procedure start-ups and is immediately available for the procedure code. This database is **read only**.

4.15.2 Manoeuvre message databases

Maneuver databases provide support for the execution of station-keeping maneuvers. Procedures may access maneuver data through a mechanism that is equivalent to the spacecraft database. The only difference is that these databases are not automatically loaded by SPELL, but they have to be explicitly loaded by the procedure developer with the `LoadDictionary` function.

In SPELL, database locations are specified using URIs (Uniform Resource Identifier). The URI corresponding to maneuver message databases is `mmd://`

Example 138: loading a manoeuvre file

```
MMD = LoadDictionary( 'mmd://Man01/Part1' )
```

In the previous example, the maneuver file is `'Part1.IMP'`, which is located within the `'Man01'` folder in the maneuver message location. Notice that:

- Real file locations are managed internally by SPELL and are not visible for the procedure: in the example, the parent folder of the `'Man01'` folder is not explicitly set. The URI `mmd` is what SPELL uses to know the real location of the database.
- The base location corresponding to a given URI is specified in the SPELL configuration files.
- The file extension can be omitted; it is `'.imp'` by default.

If the database can be successfully loaded, the `MMD` object will contain the dictionary with all the maneuver information. If there is a failure when loading the database, the function will fail and the user will be prompted. At that moment, the user may choose a different database name or abort the execution. Maneuver databases are **read only**.

4.15.3 Ground system databases

The Ground system database 'GDB' provides a set of GCS configuration parameter mappings that make SPELL procedures independent from the GCS parameter names. Two different control systems will normally have different configuration parameter names for the same feature. Thanks to the GDB mapping, the procedure source code can contain generic names which are the same no matter which GCS (i.e. SPELL driver) is being used. Besides, these generic names are normally more meaningful for the operator than the internal GCS configuration parameter name. For example, let's say the GCS configuration parameter indicating which is the active command decoder is the parameter 'CMD_ACT_DEC'. A typical GDB mapping would be the following:

Example 139: ground database mappings

DECODER	CMD_ACT_DEC
DECODER_CALIBRATION	{0:'DEC1',1:'DEC2'}

Being 'DEC1' and 'DEC2' the allowed values for the 'CMD_ACT_DEC' variable on the GCS. By using this mapping, the decoder parameter can be manipulated on the procedure as follows:

Example 140: using database mappings

```
value = GDB['DECODER_CALIBRATION'][0]

SetResource( GDB['DECODER'], value )
```

The previous example would be equivalent to the GCS dependent procedure code below:

Example 141: using database mappings (2)

```
SetResource( 'CMD_ACT_DEC', 'DEC1' )
```

4.15.4 Procedure database

The procedure database contains data regarding the procedure itself like the current Step number and identifier, the procedure identifier, and so forth. This database can be used as well to store data used during procedure execution in order to share it between functions, instead of using argument passing.

The procedure code can access this database through the global variable `PROC`. It is used as follows:

Example 142: procedure database

```
PROC['VAR'] = value
value = PROC['VAR']
```

The data stored in the procedure database is not persistent: it is lost when the procedure is closed.

There are a number of parameters pre-loaded in the PROC database when a procedure is started:

- "NAME": name of the procedure
- "ARGS": dictionary containing the arguments passed to the procedure
- "STEP": name of the current step (affected by `Step()` and `DisplayStep()` calls)
- "PREV_STEP": name of the previous step if any
- INPUT_DATA: path of the output directory used to store generated files
- OUTPUT_DATA: path of the output directory used to store generated files
- "PARENT": name and instance identifier of the parent procedure if any

4.15.5 User databases

4.15.5.1 Loading user databases

The procedure developer may create custom databases by using the URI `'usr://'`. These databases are created in a dedicated location separate from the others. For security reasons, database files are controlled by SPELL. Databases shall be accessed and manipulated using SPELL functions.

To load an existing user database, `LoadDictionary` function is used as explained:

Example 143: loading an user database file

```
DB = LoadDictionary( 'usr://MyData/data1' )
```

In the previous example, if the `'data1.IMP'` database file does not exist, the function will fail.

4.15.5.2 Creating user databases

To create a new user database from a SPELL procedure, the `CreateDictionary` function shall be used. This function accepts exactly the same arguments as `LoadDictionary`, but instead of loading an existing file will create a new one in the corresponding location.

Example 144: creating an user database file

```
DB = CreateDictionary( 'usr://MyData/data2' )
```

This function will fail if one of the following happens:

- The file name string is syntactically erroneous (spaces in the path names, for example)
- It is not possible to create the file (due to lack of write rights, for example)
- Another file of the same name exists in the same location

The function will prompt the user in case of failure so that it is possible to change the database name or abort the procedure.

4.15.5.3 Saving user databases

For saving database changes the function `SaveDictionary` is provided. This function accepts as an argument the dictionary object being used in the procedure:

Example 145: saving an user database file

```
SaveDictionary( DB )
```

This function will fail if one of the following takes place:

- The persistent storage corresponding to the given dictionary cannot be found (e.g. the file where data is stored, in the case of a flat text file database)
- There is an error while saving data

4.15.5.4 Reverting changes to user databases

Reverting changes done to a user database can be done by reloading it with `LoadDictionary` function. This will substitute the database dictionary with a fresh one containing only the data stored on the last save.

Once a database is loaded on a dictionary object, this dictionary can be manipulated freely without actually affecting the persistent storage of the database. Changes are not committed until `SaveDictionary` function is used. To append or modify a database value:

Example 146: modifying a database value

```
DB['KEY'] = new value
```

4.15.6 Database formats

As it has been said, SPELL databases are key:value pair lists. Keys and values may be any of the following:

- Numbers (integers and floats) in decimal format
- Integers in hexadecimal format: 0xFF
- Integers in octal format: 035
- Integers in binary format: 0b101001
- Date/time strings
- Python lists and dictionaries
- Strings

The example below shows an example of a SPELL database. This example would be valid for any of the database types explained before.

Example 147: database data example

```
KEY1          45.67
KEY2          This is a string
KEY3          'This is another string'
KEY4          2007/01/23 10:30
KEY5          0b011001
KEY6          [0,1,2,3,4]
KEY7          {'A':24, 'B':34, 'C':['X','Y','Z']}
```

The previous example database could be used as follows:

Example 148: database data example manipulation

```
DB['KEY1'] → 45.67
DB['KEY2'] → 'This is a string'
DB['KEY3'] → 'This is another string'
DB['KEY4'] → TIME('2007/01/23 10:30')
DB['KEY5'] → 0b011001
DB['KEY6'] → [0,1,2,3,4]
DB['KEY6'][1] → 1
DB['KEY7'] → {'A':24, 'B':34, 'C':['X','Y','Z']}
DB['KEY7']['C'][2] → 'Z'
```

1.2 Data containers

As of version 2.3 of SPELL, the concept of **data containers** has been introduced. In many situations it is desired to add type-checking to the Python language in order to improve automation safety and to avoid errors caused by wrong user inputs. The SPELL data containers are similar to Python dictionaries, but they include a number of safety mechanisms:

- **Variables are typed using the SPELL type constants:** `LONG`, `STRING`, `DATETIME`, `RETIME`, `FLOAT` and `BOOLEAN`. If the user or the procedure attempts to assign a value that does not match the variable type, an error will take place and the user will be warned. A prompt will be issued to give the chance to the user to analyse and to provide a correct value for the variable.
- **Variables may contain ranges or expected values:** it is possible to define a range of valid values or a list of expected values for the variables. Ranges cannot be defined for `STRING` type, though.
- **Empty variables are monitored:** attempts to use a variable that has not been initialized will cause an error and the user will be prompted in order to provide a correct value to the variable, before continuing the execution.
- **Optional usage checking:** if required, the data container variables will raise a confirmation prompt every time they are used in the procedures.
- **Notifications:** every time a value is assigned to, or read from a data container variable, a notification message will be sent to the graphical interface in order to show the user the actual value being used at a time.
- **Unexistent variables:** if the procedure tries to get the value of a variable that has not been declared, it will provide the chance to define it on-the-fly.

Data container syntax

The syntax to create a data container in a procedure is as follows:

Example 149: creating a data container

```
MY_DATA = DataContainer('Container Name')
```

The variables of a data container must be declared in advance in order to define correctly variable types, ranges and, if desired, default values. The syntax to declare a variable is:

Example 150: declare a variable in a data container

```
MY_DATA['VARNAME'] = Var( Default = <value>,
                          Type={LONG, STRING, BOOLEAN...},
                          Range=[0,10],
                          Expected=['A', 'B', 'C'],
                          ValueFormat=HEX,
                          Confirm=True )
```

The modifiers displayed in the example above are not all mandatory. Some rules apply:

- A variable can be declared without any modifier at all, resulting on a variable without default value, and no explicit type.
- If a data type is not provided, the type will be inferred from (a) the default value if given, (b) the first value assigned to the variable later on, (c) the data type of the range of valid values if any, (d) the data type of the list of expected values if any.
- The `ValueFormat` modifier has effect only for `LONG` types, and accepts the values `HEX`, `BIN`, `DEC` and `OCT`.
- `Range` and `Expected` modifiers cannot be used at the same time, and need to be consistent with the assigned variable type (if any).
- If the `Default` modifier is not used, the variable will be initialized to `None`, and the user will be prompted the next time the variable is going to be used.
- The `Confirm` modifier can be used to enable the "confirm before using value" mechanism for the given variable.

Note that the SPELL data containers can be used as any other regular Python dictionary, since they provide a similar set of methods, and can be iterated in loops.

Predefined data containers

The SPELL execution environment automatically creates two data containers on every procedure being executed. They are named `ARGS` and `IVARS` (for "internal variables"). These are initially empty, and can be used to store custom typed variables. These two special data containers can be directly accessed and edited from the SPELL client graphical interface.

4.16 Sub-procedures

Although SPELL allows importing other procedures through the Python clause `import`, another mechanism is provided to manage procedures programmatically.

4.16.1 Starting procedures

The `StartProc` function allows starting other procedures as if they were started from a SPELL client. Those procedures can be started in different ways:

- Blocking / Non blocking mode (default is blocking)
- Visible / Hidden (default is visible)
- Automatic / Manual (default is automatic)

Example 151: starting a sub-procedure

```
StartProc( 'procedure file name/procedure name' )
```

The child procedure is identified with its file name, without extension, or with its procedure name. The procedure name will match the file name unless otherwise specified in the procedure header.

4.16.1.1 Procedure library and priorities

SPELL procedures are organised in a controlled folder structure on the SPELL server host called SPELL procedure library. The folder structure can be described in the configuration files in such a way that folders are ordered by priority.

Example 152: procedure library example

```
Procedures/
  Bus/                Priority 3
  Payload/            Priority 2
  Validation/          Priority 1
```

In the previous example, procedures may be located in any of the three given subfolders, 'Bus', 'Payload' and 'Validation'. The 'Validation' folder would be the one with higher priority.

Priority is taken into account when starting procedures: if the same procedure name is found in two different folders of the procedure library, the one that is stored in the folder with higher priority will be taken:

Example 153: folder priorities

```
Procedures/
  Bus/
    procedure.py
  Payload/
  Validation/
    procedure.py
```

In the previous example, the SPELL procedure named 'procedure' can be found in both folders 'Bus' and 'Validation'. Since 'Validation' folder has higher priority, the following call would start the procedure found in that folder, that is, 'Validation/procedure':

Example 154: starting procedure using priorities

```
StartProc( 'procedure' )
```

Folder priorities can be overridden by providing the full path to the procedure. The following call would start the procedure located in 'Bus' folder, no matter which priorities are defined:

Example 155: starting procedure overriding priorities

```
StartProc( 'Bus/procedure' )
```

4.16.1.2 Blocking / Non-blocking mode

This mode is set with the `Blocking` modifier. If the value of the modifier is `True` (default), the parent procedure will be *stopped until the child procedure reaches to a finished state* (`FINISHED`, `ABORTED` or `ERROR`). Then, the parent procedure will continue running.

In particular, if the child procedure reaches to an error status (`ABORTED` or `ERROR`) the `StartProc` function call will fail and the user will be prompted as usual.

If the value `False` is used, the `StartProc` function will finish as soon as the sub-procedure is started, but it will not wait until the sub-procedure finishes. Therefore, the function will not fail if the procedure reaches to an error state later on. On the other hand, the function may fail if the sub-procedure fails to start.

Example 156: starting a sub-procedure in non-blocking mode

```
StartProc( 'procedure', Blocking = False )
```

4.16.1.3 Visible/Hidden mode

This mode is set with the `Visible` modifier. If the value of the modifier is `True` (default), the sub-procedure code will be shown to the user in the SPELL client application, alongside the parent procedure. If the value is `False`, the sub-procedure remains hidden to the user.

Example 157: starting a sub-procedure in hidden mode

```
StartProc( 'procedure', Visible = False )
```

4.16.1.4 Automatic/Manual mode

This mode is set with the `Automatic` modifier. If the value of the modifier is `True` (default), the sub-procedure runs as soon as it is started. If the value is `False`, the sub-procedure remains paused until the user explicitly sends the run command to start it.

Example 158: starting a sub-procedure in manual mode

```
StartProc( 'procedure', Automatic=False )
```

All these modifiers can be combined in a single function call:

Example 159: starting a sub-procedure (complex)

```
StartProc( 'procedure',  
          Automatic=False,  
          Visible=True,  
          Blocking=False )
```

4.16.1.5 Passing arguments

Arguments can be given to sub-procedures by using the keyword `args`, and giving a list containing the argument definitions:

Example 160: starting a sub-procedure with arguments

```
StartProc( 'procedure', args=[['A',1],['B',2]])
```

The passed arguments become available in the sub-procedure, in the global dictionary `ARGS`.

Example 161: accessing procedure arguments

```
value = ARGS['A']
```

4.16.1.6 Return value

The function returns `True` when the procedure is successfully executed (if in blocking mode) or successfully loaded (if in non-blocking mode).

4.17 File manipulation

The language provides some lower level functions to manipulate arbitrary text files safely from procedures.

4.17.1 Opening files

The `OpenFile` function allows opening or creating a file for read and/or write access from the procedure. The default mode is `READ` although it can be modified by using the `Mode` modifier, which accepts the values `READ`, `READ_WRITE`, `WRITE` and `APPEND`. The function returns a file handler that can be used in other file manipulation functions.

Example 162: opening a file

```
handle = OpenFile( '<file path>', Mode=READ_WRITE )
```

It is discouraged to use absolute paths for files. At the present version of the language this is not forbidden, but it will in later revisions. The file path shall be relative to one of the two paths provided by SPELL in the `PROC` dictionary: `PROC[INPUT_DATA]` to have the path to procedure input files, and `PROC[OUTPUT_DATA]` for procedure output files.

4.17.2 Closing files

The files shall be closed before end of the procedure with the `CloseFile` function. This function accepts a file handler obtained with `OpenFile` as an argument.

Example 163: opening a file

```
handle = CloseFile( handle )
```

4.17.3 Write data to files

In order to write data to files the function `WriteFile` can be used. It accepts strings or lists of strings as arguments.

Example 164: writing to a file

```
True/False = WriteFile( handle, 'string' )  
  
True/False = WriteFile( handle, ['string1','string2'] )
```

In the first construct, the string is written to the file verbatim. With the second construct, each of the strings of the list are written as separate lines, that is, a '\n' character is added at the end of each.

4.17.4 Read data from files

In order to read data from files the function `ReadFile` can be used.

Example 165: reading file lines

```
[lines] = ReadFile( handle )
```

The function returns a Python list containing all the lines of the file.

4.17.5 Read directory contents

In order to read data from files the function `ReadDirectory` can be used. It accepts either a `File` object (`spell.lib.adapter.file.File` class instance) or a string containing a path.

Example 166: reading directory contents

```
[list of files/directories] = ReadDirectory( handle )  
[list of files/directories] = ReadDirectory( string )
```

The function returns a Python list containing all the directory contents. It will fail if the directory does not exist or it is not readable.

4.17.6 General file system operations

By using the class `File` (automatically imported from `spell.lib.adapter.file`) a number of standard operations can be performed on file system paths. To create a `File` instance:

Example 167: create a File instance

```
f = File( 'path' )
```

With that instance the following operations can be done:

- Get full path string: `f.filename()` → string
- Get containing directory: `f.dirname()` → string
- Get base file name: `f.basename()` → string
- Check file existence: `f.exists()` → True/False
- Check type: `f.isdir()`, `f.isfile()` → True/False
- Check permissions: `f.canWrite()`, `f.canRead()`, `f.isOpen()` → True/False
- Append paths: `f = f + 'extra_path'` → updated File object

4.17.7 Deleting files and directories

The `DeleteFile` function can be used for this purpose:

Example 168: deleting files and directories

```
DeleteFile( 'path' )  
DeleteFile( <file object> )
```

The function will fail if the procedure attempts to delete a directory which is not empty.

4.17.8 Error handling with file operations

As in all other SPELL functions, any error taking place during a file operation will result on a prompt being issued to the user, explaining the reason of the error, and providing a set of possible actions to be carried out in consequence. The set of available actions is as usual configured with the `OnFailure` modifier.

4.18 Ranging operations

The SPELL framework provides also support for interacting with Ranging and Tracking software. The most common operations with ranging systems have been included in the language:

- Ability to start or stop ranging activities.
- Get information about available antennae and baseband stations.
- Configure and calibrate the ranging software and baseband stations.
- Get information about ongoing ranging activities, if any.

Following the usual SPELL philosophy, the SPELL driver is not obliged to implement the ranging services associated to these capabilities. This feature is, as all other in SPELL, optional.

4.18.1 Enabling or disabling ranging activities

Typically the ranging software performs periodic ranging activities. These activities are in one way or the other triggered by a ranging plan or some kind of scheduling system. As the ranging activities may interfere with regular commanding from the ground control system, a common use case is to have the need to disable the ranging software when telecommands need to be sent to the spacecraft.

The functions `EnableRanging` and `DisableRanging` are provided, so that the ranging plan or schedule can be disabled or put on hold from SPELL procedures.

Example 169: enabling or disabling the ranging system

```
True/False = EnableRanging()  
True/False = DisableRanging()
```

Note that these functions do not take arguments like antenna names or baseband station names, as the action to be performed is independent from those.

These two functions are not to be confused with the `StartRanging` or `AbortRanging` functions, which are used to start or stop specific ranging activities on demand.

4.18.2 Starting or stopping ranging activities

As mentioned in the previous section, the procedures can start or stop specific ranging activities on particular antennae and baseband stations on demand. The functions `StartRanging` and `AbortRanging` are provided for this objective.

```
<value> = GetBasebandConfig( 'BBE', 'NAME',  
                               ValueFormat=ENG/RAW )
```

It should be noted that `SetBasebandConfig` can be used with multiple parameters. In all the syntaxes the 'NAME' element identifies a given baseband parameter name, and 'VALUE' the value corresponding to this parameter.

Also note that `GetBasebandConfig` allows specifying an optional calibration of the returned value by means of the `ValueFormat` modifier, in a way similar to the `GetTM` function.

The functions `GetBasebandNames` and `GetAntennaNames` can be used to retrieve information about the available/known baseband stations and antennae.

Example 174: getting BBE/Antennae information

```
[list]= GetBasebandNames ()
[list]= GetAntennaNames ()
```

Finally, the `StartRangingCalibration` function can be used to start calibration activities of the ranging software when needed. The calibration activities are expected to be affected by the `AbortRanging` function as well, therefore there is no `AbortCalibration` function in the language.

Example 175: calibrate the ranging system

```
True/False = StartRangingCalibration ( 'BBE', 'ANTENNA' )
```

4.18.4 Ranging status

The function `GetRangingStatus` can be used by procedures to find out the current status of ranging activities. This function returns one of the following SPELL constants : `IDLE`, `RANGING`, `CALIBRATING`, or `ERROR`.

Example 176: status of the ranging system

```
<status> = GetRangingStatus ()
```

4.19 Sharing data between procedures

As of SPELL 2.4 version, it is possible to share data between procedures running under the same context. To support this feature, the context process holds a "blackboard" mechanism that the procedures may use to write and read data as it was a dictionary or set of key-value pairs.

4.19.1 Setting shared data values

Example 177: setting shared data values

SetSharedData accepts also a syntax based on modifiers:

It is also possible to write several values to the blackboard at the same time by using the following syntax:

[illegible]

In order to organize the data in scopes, the `Scope` modifier shall be used in any of the three previous syntax examples:

Example 180: setting shared data values in a scope

```
True/False=SetSharedData('NAME', <value>, Scope='scope')
```

Note that in order to set or get data in or from a scope different from the global scope, it is required that this scope exists already. The scopes can be created with the function `AddSharedDataScope`:

Example 181: creating a scope

```
True/False=AddSharedDataScope( 'NAME' )
```

4.19.2 Test-and-set shared data values

The `SetSharedData` function also supports a test-and-set mechanism. If an expected value is given in addition to the variable name and the value to be written, the system will set the variable value **only** if the current value matches the expected one. This mechanism can be used to implement synchronization mechanisms between procedures.

Example 182: test and set shared data values

```
SetSharedData( ['NAME', <value>, <expected>] )

SetSharedData( Name='NAME',
                 Value=<value>,
                 Expected=<expected> )
```

Example 183: test and set shared data values (2)

```
SetSharedData( [[ 'NAME', <value>, <expected> ]
                 [ 'NAME', <value>, <expected> ] ] )
```

Note that when using multiple items, the return value of the `SetSharedData` function is a list of boolean flags. If a flag is set to `False`, it means that the variable value could not be updated. This mechanism is only relevant for the test-and-set, as it allows to identify which variables could not be updated since the expected value did not match.

4.19.3 Getting shared data values

The `GetSharedData` function can be used to read values from the blackboard.

Example 184: getting shared data values

```
<value> = GetSharedData ( 'NAME' )
```

If the variable does not exist this function call generates an error and a prompt is issued to the user.

To acquire multiple variables at the same time:

Example 185: getting shared data values (2)

```
[<values>] = GetSharedData ( ['NAME', 'NAME', ...] )
```

Finally, the `Scope` modifier can be used as well on this function in order to use specific variable scopes instead of the global scope:

Example 186: getting shared data values (3)

```
<value> = GetSharedData ( 'NAME', Scope='scope' )
```

In order to know the list of existing scopes and variables from a procedure, the following two functions can be used:

Example 187: getting shared data information

```
[<list>] = GetSharedDataKeys ( Scope='scope' )  
[<list>] = GetSharedDataScopes ( )
```

4.19.4 Clearing shared data values

The `ClearSharedData` function can be used to remove values from the blackboard.

Example 188: clearing shared data values

```
True/False = ClearSharedData()  
  
True/False = ClearSharedData( 'NAME' )  
  
True/False = ClearSharedData( ['NAME', 'NAME', ...] )  
  
True/False = ClearSharedData( 'NAME', Scope='scope' )  
  
True/False = ClearSharedData( Scope='scope' )
```

The first construct would delete all existing variables *in the global scope* in the blackboard. The second construct would delete the variable 'NAME' in the global scope. Similarly the third construct deletes several variables in the global scope. When the Scope modifier is used, only the variables on that scope are affected. In order to delete a scope completely, the `ClearSharedDataScope` function is used:

Example 189: clearing shared data scopes

```
True/False = ClearSharedDataScopes()  
  
True/False = ClearSharedDataScopes( 'scope' )  
  
True/False = ClearSharedDataScopes( Scope='scope' )
```

4.19.5 The GLOBAL scope

The global scope always exists on the blackboard, and cannot be deleted (although its variables can be removed). When no explicit scope name is given in the function calls, the global scope is used unless told otherwise. To make the operations more explicit, it is also possible to use the Scope modifier with the constant GLOBAL:

Example 190: clearing shared data scopes

```
True/False = ClearSharedData( Scope=GLOBAL )
```

4.20 Memory management functions

The memory management functions allow SPELL procedures to perform operations related to the on-board memory, like generating memory reports, comparing memory reports, extracting data from them, and so forth.

4.20.1 Generate memory reports

The function `GenerateMemoryReport` can be used to produce ASCII reports with contents of the on-board memory of the spacecraft. It is understood that these reports are produced using a reference memory image of the on-board memory stored in the ground by the ground control system.

The actual data included in the reports depends on several parameters that can be given to the function. Some of these parameters may not be supported by a ground control system; in that case, and following the usual SPELL approach, these parameters can be just ignored or never used.

The parameters or modifiers accepted by the function are:

- **Image:** used to provide the identifier of a "reference image" of the on-board memory. It is assumed that a reference image is a binary file maintained by the Ground Control System, that reflects the on-board memory of the spacecraft. The `GenerateMemoryReport` function supports having more than one image of the on-board memory to work with. This modifier is used to indicate which of the reference images should be used to generate the report.
- **Type** and **Source:** these optional modifiers can be used to provide filtering parameters to the data to be included in the reports. By using the values given in these modifiers, the SPELL drivers may filter the on-board memory data and generate reports including a subset of those data only.
- **Begin** and **End:** used to delimit on-board memory ranges (e.g. memory addresses) if applicable.
- **Destination:** used to indicate a path where to store the generated report. This path shall use the same URI format as the `LoadDictionary`, `SaveDictionary` and `CreateDictionary` functions; for example, the location "mem://" can be defined to point to a given SPELL data output directory in the XML context configuration file.

Example 191: generate memory reports

```
True/False = GenerateMemoryReport (
    Type           = 'MyDataType',
    Source         = 'MyDataCourse',
    Image          = 'ReferenceImageName',
    Begin          = 'Start address',
    End            = 'End address',
    Destination    = 'mem://My_Report'
)
```

The function returns `True` if the operation is successful. As usual, in case of failures, the user will be prompted with the possible actions to be taken.

When using memory addresses, it is assumed that the values for `Begin` and `End` are valid addresses understandable by the driver. If they are not understandable or are not consistent with the on-board memory or the GCS data, the driver should generate an exception.

4.20.2 Comparing memory images

The function `CompareMemoryImages` allows the procedure to compare two on-board memory images available on the GCS. The actual comparison is to be carried out on the GCS side, transparently to the procedure. The function returns `True` in case of matching images.

Example 192: compare memory images

```
True/False = CompareMemoryImages (
    Image      = ['image1', 'image2']
)
```

The `Image` modifier is used to provide the names of the two reference images to be compared.

The modifiers `Type`, `Source`, `Begin` and `End` are also available in this function and have the same filtering capabilities as for the `GenerateMemoryReport` function. The difference is, that these modifiers limit the comparison of the on-board memory images to the specified ranges of data, instead of performing a comparison of the complete images.

Example 193: compare memory images on limited data ranges

```
True/False = CompareMemoryImages (
    Image      = ['image1', 'image2'],
    Type       = 'MyDataType',
    Source     = 'MyDataSource',
    Begin      = 'Start address',
    End        = 'End address'
)
```

4.20.3 Extract data from memory images

The function `MemoryLookup` allows the procedure to extract data values from a given on-board memory dump or "reference image" maintained by the Ground Control system.

The returned value depends on the ground control system. It would be normally the value of the memory segment indicated by the function parameters.

As in the other memory management functions, the modifiers `Image`, `Type`, `Source`, `Begin` and `End` can be used for filtering and data constraint purposes.

Additionally, the `MemoryLookup` function supports the modifier `Name` to provide an identifier for the concrete datum that needs to be provided, if needed.

The modifier `ValueFormat` can be used to ask for `RAW` or `ENG` (calibrated) values.

Example 194: extract memory values

```
<dict> = MemoryLookup (
    Name      = 'Identifier',
    Image     = 'image1',
    Type      = 'MyDataType',
    Source    = 'MyDataSource',
    Begin     = 'Start address',
    End       = 'End address'
)
```

4.21 Ground control system TM/TC database

In this section, functions related to access and management of the TM/TC databases of the Ground Control systems are described.

4.21.1 Extract data from TM/TC database tables

The function `TMTCLookup` provides a generic way to access the TM/TC database tables of the Ground Control system. The modifiers `Name`, `Type`, `Begin`, `End` and `Source` can be used to build queries that are applied on the GCS database in order to extract any arbitrary data. The actual effect of these modifiers depends on the driver. The proposed approach is nevertheless described next:

- **Name:** provides the identifier or mnemonic of a concrete datum to be obtained.
- **Source, Type:** can be used to identify a TM/TC database table, or a data subset.
- **Begin, End:** can be used to add additional filtering to the database query by providing numerical constraints to the queries like lower/upper limits, time windows, etc.

The returned value depends on the ground control system, but a Python dictionary is proposed as the best approach as it allows containing structured data sets.

Example 195: extract TM/TC database values

```
<dict> = TMTCLookup (  
    Name      = 'Identifier',  
    Type      = 'MyDataType',  
    Source    = 'MyDataSource',  
    Begin     = 'Lower limit',  
    End       = 'Upper limit'  
)
```

5 Appendix A: table of functions

The following table summarizes the set of available SPELL functions:

Function	Short description	Section
Abort	Aborts the procedure execution	4.14.3
AbortRanging	Abort any ongoing ranging activity	4.19
AddSharedDataScope	Add a scope to the blackboard of shared variables	4.20
BuildTC	Build a telecommand item	4.4
CreateDictionary	Create a new user database	4.15.5.2
ChangeLanguageConfig	Change default modifier values	3.2.1
ClearSharedData	Remove shared variables	4.20
ClearSharedDataScopes	Remove shared variable scopes	4.20
CloseFile	Close a generic file	4.18
CloseDisplay	Close a TM display	4.13
CloseWorkspace	Close a TM workspace	4.13
CompareMemoryImage	Compare two on-board memory dumps on the GCS	4.21
DeleteFile	Delete a file or directory	4.18
DisableAlarm	Disable TM parameter OOL alarms	0
DisableRanging	Disable the ranging software	4.19
DisableUserAction	Disable a programmed user action	4.14.4
DismissUserAction	Remove a programmed user action	4.14.4
Display	Display a message	4.9
DisplayStep	Display step/stage information	1.1.1

Function	Short description	Section
EnableAlarm	Enable TM parameter OOL alarms	0
EnableRanging	Enable the ranging software	4.19
EnableUserAction	Enable a programmed user action	4.14.4
Event	Inject an event	4.10
Finish	Finish the execution successfully	4.14.3
GenerateMemoryReport	Generate a report of the on-board memory	4.21
GetAntennaNames	Get list of known antenna names	4.19
GetBasebandConfig	Get baseband configuration parameters	4.19
GetBasebandNames	Get list of known basebands	4.19
GetRangingStatus	Get status of the ranging software	4.19
GetResource	Get a configuration parameter	4.11
GetSharedData	Get shared variables	4.20
GetSharedDataKeys	Get list of existing variables	4.20
GetSharedDataScopes	Get list of existing variable scopes	4.20
GetTM	Retrieve a TM parameter value	4.2
GetLimits	Get TM parameter OOL definitions	4.8
IsAlarmed	Check OOL status	4.8
LoadDictionary	Load a database	4.15
LoadLimits	Load a set of OOL definitions from a file	4.8
MemoryLookup	Extract values from on-board memory	4.21
Notify	Send an item notification	4.9

Function	Short description	Section
OpenFile	Open a generic file	4.18
OpenDisplay	Open a TM display	4.13
OpenWorkspace	Open a TM workspace (set of displays)	4.13
Pause	Pause the execution	4.14.3
PrintDisplay	Print a TM display	4.13
Prompt	Prompts user for information	4.12
ReadDirectory	Read contents of a directory	4.18
ReadFile	Read data from a file	4.18
RestoreNormalLimits	Restore normal OOL definitions in GCS	4.8
SaveDictionary	Save changes done to a dictionary	4.15.5.3
Send	Send telecommands	4.5
SetBasebandConfig	Set baseband configuration parameters	4.19
SetGroundParameter	Inject a TM parameter value	4.7
SetSharedData	Set shared data variables	4.20
SetUserAction	Program an user action function	4.14.4
StartRangingCalibration	Start a ranging calibration activity	4.19
SetResource	Set a configuration parameter	4.11
SetLimits	Set TM parameter OOL definitions	4.8
StartProc	Start a subprocedure	4.16
StartRanging	Start a ranging activity	4.19
TMTCLookup	Extract values from the GCS TM/TC database	4.22

Function	Short description	Section
Verify	Verify a set of TM conditions	4.3
WaitFor	Wait for a time or TM condition	4.6
WriteFile	Write data to a file	4.18

6 Appendix B: table of modifiers

The following table summarizes all the SPELL modifiers and their usage for each function they are applicable to:

Modifier	Function	Value	Description
AdjLimits	Send, Verify	True,False	It is applicable for Send when using a TM verification part. If True, it will make the system adjust the OOL definitions of the given parameters by using the passed midpoints and tolerances.
Automatic	StartProc	True,False	If true, the procedure is set to running state as soon as it is loaded. If false, the procedure remains paused.
Begin	CompareMemoryImages GenerateMemoryReport MemoryLookup TMTCLookup	<STRING>	Establish a lower limit or starting point for a memory or database operation.
Block	Send	True,False	Determines if a group of commands should be considered as a command block. Availability depends on the SPELL driver.
Blocking	StartProc	True,False	If True, the parent procedure execution is blocked until the child procedure finishes. If False, the parent procedure continues as soon as the child procedure is correctly loaded.
Confirm	Send, data containers	True,False	If True, the user shall confirm explicitly that a telecommand should be sent to the S/C. When used in data container variables, it enables the variable usage confirmation mechanism.
Default	Prompt, data containers	<STRING>, True/False, <TIME>	Sets the default value for a prompt. Takes effect when used in combination with the Timeout modifier. In data containers, gives the default value for a variable.
Delay	Send	<TIME> (relative)	Applicable only when the TM verification part is used. It determines the time delay between the TC verification and the start of the TM verification.
Delay	Verify	<TIME>	Determines the time to wait before starting the TM verification

Modifier	Function	Value	Description
		(relative)	
Delay	WaitFor	<TIME> (relative)	Sets the amount of time (relative) to wait before continuing the execution.
End	CompareMemoryImages GenerateMemoryReport MemoryLookup TMTCLookup	<STRING>	Establish an upper limit or ending point for a memory or database operation.
Extended	GetTM	True, False	Retrieve detailed TM item information
Expected	data containers SetSharedData	List of values	Contains the list of valid values for a data container variable. For shared data specifies the expected value of the shared variable required to perform the value update
HandleError	<ALL>	True, False	If False, and there is a failure in a SPELL function call, the error will be raised to the procedure level so that the procedure code can handle it manually.
HiBoth	SetLimits, GetLimits	<NUMBER>	High/Red and High/Yellow out of limit definition value
HiRed	SetLimits, GetLimits	<NUMBER>	High/Red out of limit definition value
HiYel	SetLimits, GetLimits	<NUMBER>	High/Yellow out of limit definition value
Host	OpenDisplay OpenWorkspace CloseDisplay CloseWorkspace	<STRING>	Sets the host name where a TM display or workspace should be open or closed.
IgnoreCase	Verify	True, False	If True, it indicates that string comparisons should be case insensitive when performing a TM verification
Image	GenerateMemoryReport CompareMemoryImages MemoryLookup	<STRING>	Identify an on-board memory reference image or dump
Interval	WaitFor	<TIME> or list of	Sets the update interval when waiting for a given time condition

Modifier	Function	Value	Description
		<TIME> (relative)	
LoadOnly	Send	True,False	If True, the sent telecommand is considered as successfully verified as soon as it is loaded on board. There is no execution verification.
LoBoth	SetLimits, GetLimits	<NUMBER>	Low/Red and Low/Yellow out of limit definition value
LoRed	SetLimits, GetLimits	<NUMBER>	Low/Red out of limit definition value
LoYel	SetLimits, GetLimits	<NUMBER>	Low/Yellow out of limit definition value
Message	WaitFor	<STRING>	Sets the update message when waiting for a given time condition
Midpoint	SetLimits, GetLimits	<NUMBER>	Middle point for an out of limit definition, used in combination with Tolerance modifier
Mode	OpenFile	READ READ_WRITE WRITE APPEND	Mode to be used when opening a file on the file system.
Monitor	OpenDisplay OpenWorkspace	<ANY>	Identifies the monitor where a display or workspace should be open.
Name	SetSharedData GetSharedData ClearSharedData	<STRING>	Indicates the name of shared data variables
Notify	<ALL>	True,False	If True (default) information and notifications regarding the function call are sent to SPELL clients. If False, the function is executed silently and no feedback is sent to the user.
OnFailure	(ALL)	ABORT, SKIP, REPEAT, CANCEL, RESEND, RECHECK	Sets the set of available actions that the user may choose whenever a failure occurs during a function call. Applicable values depend on the particular function being used. The values may be combined with the logical or (' ').

Modifier	Function	Value	Description
OnTrue/OnFalse	(ALL)	NOACTION, ABORT, SKIP, REPEAT, CANCEL, RESEND, RECHECK	Sets the set of available actions that the user may choose whenever a SPELL function returns True/False (If the value is NOACTION, nothing is done). PromptUser value will determine if an automatic action is carried out instead of prompting the user.
Printer	PrintDisplay	<STRING>	Sets the name of the printer to be used.
PromptUser	(ALL)	True,False	If False, the user will not be prompted if there is a failure during the function call, but the action specified by OnTrue/OnFalse modifiers will be directly carried out.
Range	Data containers	List of two values	Range of accepted numeric values for a data container variable
Radix	BuildTC	HEX, DEC, OCT, BIN	Sets the radix of the TC argument value
Retries	Verify	<INT>	Sets the number of repetitions for a TM condition check before declaring it as failed.
SendDelay	Send	<TIME> (relative)	Sets the amount of time the GCS should wait before sending a command. Availability depends on the SPELL driver.
Severity	Display, Event, SetUserAction	INFORMATION, WARNING, ERROR	Determines the severity of the message being shown/injected into the GCS. For user actions, determines the formatting of the action trigger on the client.
Source	GenerateMemoryReport CompareMemoryImages MemoryLookup TMTCLookup	<STRING>	Used to provide selection / filter conditions for memory and database operations
Time	Send	<TIME> (absolute)	Used for time-tagging commands.
Time	OpenDisplay	<TIME> (relative)	Indicates a time-span for a plot

Modifier	Function	Value	Description
Timeout	GetTM	<TIME> (relative)	Sets the maximum time allowed to acquire the TM parameter value. It does make sense when Wait=True only. If the parameter sample does not come before the time limit is reached, the function fails.
Timeout	Prompt	<TIME> (relative)	Maximum time to wait for the user to give an answer. If used in combination with the Default modifier, the default value is returned if the limit time is reached. If no default value is given, the prompt fails.
Tolerance	SetLimits, GetLimits	<NUMBER>	Used for setting out of limit definitions, in combination with Midpoint modifier.
Tolerance	Verify	<NUMBER>	Tolerance to be used in TM verifications
Type	Prompt GenerateMemoryReport CompareMemoryImages MemoryLookup TMTCLookup	OK,CANCEL,YES, NO,YES_NO, OK_CANCEL,LIST, ALPHA,LIST ALPHA, LIST NUM, NUM <STRING>	Determines the type of prompt to be performed. Used to provide selection / filter conditions for memory and database operations
Units	BuildTC	<STRING>	Sets the units name of the TC argument value
Until	WaitFor	<TIME> (absolute)	Used for holding the procedure execution until the given time arrives.
Value	SetSharedData GetSharedData ClearSharedData	<PRIMITIVE TYPE>	Specifies the values for shared data variables
ValueFormat	GetTM, Verify, GetBasebandConfig	ENG,RAW	Determines whether the calibrated parameter value (ENG) or the raw value (RAW) should be used for the operation being done.
ValueType	BuildTC	LONG/STRING/	Forces the type of a telecommand argument value

Modifier	Function	Value	Description
		BOOLEAN/TIME / FLOAT	
Visible	StartProc	True,False	If True, the procedure appears on the SPELL client applications alongside the parent procedure.
Wait	GetTM	True,False	If True, the function will wait for the next parameter update and will return the new TM parameter value. If False, the current value (last recorded value) of the parameter will be returned.

The `<TIME>` tag in the previous table means any of the following:

- When *relative* times are accepted, an integer/float indicating an amount of seconds
- When *relative* times are accepted, a `TIME` object or a date/time string indicating a relative time value.
- When *absolute* times are accepted, a `TIME` object or a date/time string indicating an absolute time value.


7 Appendix C: modifier default values

SPELL modifiers may have a predefined default value. These defaults are specified in the SPELL language configuration files which are described in the SPELL operations manual.

Modifier values configuration is **spacecraft dependent**: different spacecraft may have different modifier default values within the same SPELL server.

The following table shows a typical set of default values. Please notice that the actual values being used by a procedure may be different from these, since they depend on configuration files.

Modifier	Function	Default Value
AdjLimits	Send, Verify	True
Automatic	StartProc	True
Block	Send	False
Blocking	StartProc	True
Confirm	Send	False
HandleError	<ALL>	False
IgnoreCase	Verify	False
Notify	<ALL>	False
OnFailure	(ALL)	ABORT SKIP REPEAT CANCEL
PromptUser	(ALL)	True
OnTrue	(ALL)	NOACTION
OnFalse	(ALL)	NOACTION
OnFalse	Verify	ABORT SKIP REP EAT CANCEL
Retries	Verify	2
Severity	Display, Event, SetUserAction	INFORMATION
Timeout	GetTM	30 (*)
Tolerance	Verify	0.0
Type	Prompt	OK
ValueFormat	GetTM, Verify	ENG
Visible	StartProc	True
Wait	GetTM	False

03 February, 2015	SPELL Language Reference	
Page 115 of 118	File: SPELL - Language Reference - 2.4.4.docx	


Notice that some modifiers do not have a default value. In these cases, the modifier will not be shown in this table.

(*) The Timeout for telemetry values acquisition should be normally configured to match the S/C telemetry format period. 30 seconds is just an example.

8 Appendix D: modifiers applicable to all functions

Modifier	Value	Description
Notify	True/False	<p>Enable or disable the item notifications that can be generated by all functions, which are displayed on the SPELL GUI code view (if used), in the Data and Status columns. Disabling notifications will increase execution speed but less feedback will be given to users.</p>
OnFailure	ABORT SKIP CANCEL REPEAT RECHECK* RESEND* NOACTION RESUME* HANDLE	<p>Indicate the possible actions available in case of operation failure.</p> <ul style="list-style-type: none"> If the function call is interactive for operation failures (PromptFailure=True), this modifier indicates the list of options presented to the user. The value can be a combination (using " ") of the list on the left. If the function call is not interactive for operation failures (PromptFailure=False) this modifier shall have one single value, which is the action that will be automatically performed by the procedure in case of failure. <p>The possible actions are:</p> <ul style="list-style-type: none"> ABORT: abort the execution and the procedure. SKIP: dismiss the current operation but assume it was success. This means, in many functions, that the function will return True. CANCEL: same effect as SKIP, but considering the function call as failed. This means, in many functions, that the function will return False. REPEAT: attempt to do the same operation again. NOACTION: perform no action and just return the value that the function can provide. To be used for non-interactive case (PromptFailure=False) HANDLE: let the procedure handle the failure. To be used in interactive cases (PromptFailure=True) where the option HANDLE is given to the user together with others. If HANDLE is selected, the function will raise a Handle exception object at that moment, and the object shall be captured at procedure level with an except clause. <p>There are some specific values kept for backwards compatibility:</p> <ul style="list-style-type: none"> RECHECK is specific to telemetry verifications (Verify function) and tells the procedure to repeat failed telemetry acquisitions. RESEND is specific to Send function and tells the procedure to retry a command injection. RESUME is not part of the SPELL core but it can be used by drivers to support resuming command group injection from the last successful command, instead of letting SPELL retry the whole command group injection upon failures.
FailureCode	<int>	<p>Can be used to indicate a custom error code in any SPELL function. Shall be used in combination with HANDLE.</p>
PromptFailure	True/False	<p>If True, the user will be prompted through the controlling client and given all possible actions to perform after the failure. The options are indicated in the OnFailure modifier. If False is given to this modifier, the procedure will perform automatically the action indicated by OnFailure.</p>

HandleError	True/False	<p><i>If True, let SPELL handle operation failures and therefore use the above modifiers to configure the reaction of the procedure to them (OnFailure, PromptFailure, etc.).</i></p> <p><i>If False, any operation failures will not be handled by SPELL and the procedure is in charge of capturing all DriverException objects and reacting in consequence. If False value is used, but the function calls are not encapsulated on a try-except clause, any failure on the operations will crash the procedure.</i></p>
OnTrue	ABORT SKIP CANCEL REPEAT RECHECK* RESEND* NOACTION	<p><i>If PromptUser = True, present the combination of actions given by this modifier to the user in order to select which action to perform if the SPELL function returns True.</i></p> <p><i>If PromptUser = False, perform the action given by this modifier directly without prompting the user, if the SPELL function returns True.</i></p> <p><i>If the function returns False, nothing is done (unless OnFalse modifier is used). Note that this option is only valid if there are no operation failures, but the SPELL function returns True as a normal result of the operation.</i></p> <p><i>Refer to OnFailure for description of the actions.</i></p> <p><i>It can be combined with OnFalse.</i></p>
OnFalse	ABORT SKIP CANCEL REPEAT RECHECK* RESEND* NOACTION	<p><i>If PromptUser = True, present the combination of actions given by this modifier to the user in order to select which action to perform if the SPELL function returns False.</i></p> <p><i>If PromptUser = False, perform the action given by this modifier directly without prompting the user, if the SPELL function returns False.</i></p> <p><i>If the function returns True, nothing is done (unless OnTrue modifier is used). Note that this option is only valid if there are no operation failures, but the SPELL function returns False as a normal result of the operation.</i></p> <p><i>Refer to OnFailure for description of the actions.</i></p> <p><i>It can be combined with OnTrue.</i></p>

03 February, 2015	SPELL Language Reference	
Page 118 of 118	File: SPELL - Language Reference - 2.4.4.docx	