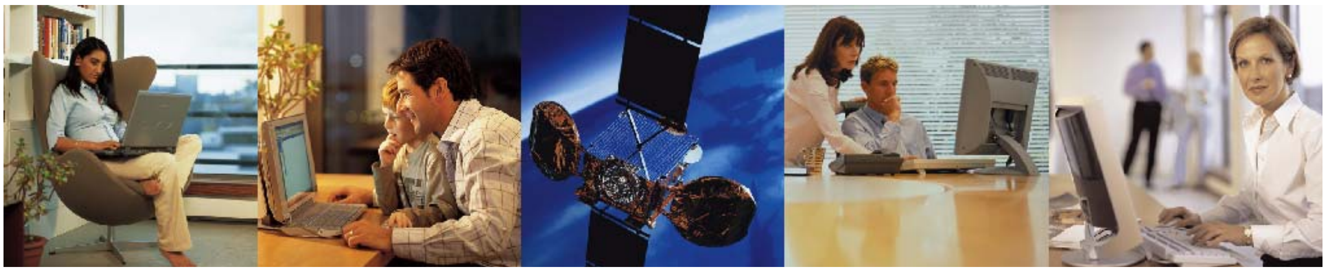

SPELL Language Manual

Software version 2.0.1

Author	Date	Version	Comment
Rafael Chinchilla & Fabien Bouleau	15 July 2008	2.0	Document created
Javier Noguero	12 March 2008	2.1	Added Python introduction Corrections: - Time services - Updated SPELL functions
Rafael Chinchilla	14 March 2008	2.1	Revision for Starting package Corrections: - Acronymns - Example TOC - Spelling - Example code - Extended SPELL intro - Completed function Send - Added subprocedure mgmt
Rafael Chinchilla	19 March 2008	2.1	Added missing functions Removed double quotes
Rafael Chinchilla	22 March 2008	2.1	Minor revision after G. Morelli comments
Rafael Chinchilla	25 March 2008	2.1	Minor revision after A. Pierre comments
Fabien Bouleau	08 July 2009	2.1	Minor revision after J. Boesen comments
J. Andres Pizarro	07 August 2009	2.1	Updated Prompt function description
Rafael Chinchilla	01 September 2009	2.2	Updated OnTrue/OnFalse/PromptUser and Verify return values
Rafael Chinchilla	09 April 2010	2.3	Minor changes
Rafael Chinchilla	14 April 2010	2.3	Added new limits spec
Rafael Chinchilla	19 Nov. 2010	2.4	Updated to 2.0 version

Prepared By	Rafael Chinchilla Javier Noguero Fabien Bouleau	Signature: *signature on file*
Reviewed By	Thomas Nowak	Signature: *signature on file*
Authorized By	Thomas Nowak	Signature: *signature on file*

Table of Contents

Table of Examples

Ref. Documents

Acronyms

CV	Command Verification
GCS	Ground Control System
GDB	Ground Database
GMV	GMV Aerospace & Defence (www.gmv.com)
GUI	Graphical User Interface
HMI	Human Machine Interface (equivalent to GUI)
IDE	Integrated Development Environment
MMD	Manoeuvre Message Database
OOL	Out-of-limits
PROC	Automated SPELL procedure
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

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 and 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.

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 '\' (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
+	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
==	equal
!=	not equal
<	smaller
>	greater
<=	smaller or equal
>=	greater or equal
Logical operator	**Meaning**
and	and
or	or
not	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 use 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
```

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
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.**.

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.

There are two ways of passing a modifier/value pair to a SPELL function: keyword parameters and configuration dictionaries.

3.2.1 Configuration Dictionaries

A Python dictionary containing modifier/value pairs can be passed as an argument to a SPELL function. When used, the new values will override the default configuration.

The following code examples shows the generic form of a SPELL function that uses the default interface configuration [1] and function calls that use configuration dictionaries [2] and [3].

Example 19: SPELL function with configuration dictionary

```
[1] Function( arg1, arg2 )

[2] Function( arg1, arg2, { Mod1:val1, Mod2: val2 })

[3] Function( arg1, arg2,
              config = { Mod1:val1, Mod2: val2 } )
```

The second call will set the given SPELL modifiers `'Mod1'` and `'Mod2'` with the given values, overriding any possible interface default.

The third form [3] shown is almost identical to the form [2], but with a slight difference that affects the way the modifiers are processed (notice the keyword **config**). This mechanism will be explained later.

3.2.2 Keyword Parameters

Please refer to the Python language reference for more details about keyword parameters.

This mechanism consists on passing the modifiers and values as keyword parameters instead of passing them inside a dictionary. The form of the function call changes, but the underlying idea is exactly the same. The difference resides basically on the code readability. In some cases it may be more convenient to write the function call with keyword parameters in order to reduce the amount of curly braces, quotes and other symbols. Besides, using keyword parameters may be used when using *priorities* to define complex configurations, as it will be explained later in this section.

The following code example shows a function call where configuration is overridden by using keyword parameters.

Example 20: SPELL function with keyword parameter

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

3.2.3 Configuration Priorities

The mechanisms explained above may be combined in the same function call. This is usually done when creating complex configurations. The reason to mix the two mechanisms is that SPELL functions will assign different priorities to the modifiers, depending on the mechanism used to pass them. In previous sections, three mechanisms have been shown:

- Using a configuration dictionary
- Using a configuration dictionary with keyword **config**
- Using keyword parameters

All these mechanisms may be combined in the same function call, taking into account the following priorities:

High	– Keyword parameters
Medium	– Dictionary with **config** keyword
Low	– Dictionary

This means that, if a modifier is passed by using these three mechanisms in the same function call, the modifier value finally used will be the one provided by the keyword parameter. In absence of a keyword parameter, the value given inside the dictionary with **config** keyword will override any value given in a normal dictionary. An example can be seen below:

Example 21: configuration priorities

```
[1] Function( arg, {Mod:A}, config={Mod:B}, Mod=C )

[2] Function( arg, {Mod:A}, config={Mod:B} )

[3] Function( arg, {Mod:A}, Mod=C )

[4] Function( arg, config={Mod:B}, Mod=C  )
```

The value of the SPELL modifier 'Mod' in each case will be:

- `[1] Mod = C` because the keyword argument has highest priority
- `[2] Mod = B` because the dictionary with **config** keyword has highest priority
- `[3] Mod = C` because the keyword argument has highest priority
- `[4] Mod = C` because the keyword argument has highest priority

A common situation where such priorities mechanism may be useful is when there is a configuration dictionary which is being used for several function calls along the procedure, but in very particular cases some modifier values of this dictionary have to be overridden for some of the function calls.

The following code shows an example of this situation:

Example 22: overriding configurations

```
CommonConfig = {Mod1:A, Mod2:B}

[1] FunctionA( arg, CommonConfig )

[2] FunctionB( arg, CommonConfig, Mod2=C )

[3] FunctionC( arg, CommonConfig )
```

The same interface configuration is used when calling the functions A and C, but the value of the SPELL modifier ‘Mod2' is overridden in the function B call with the value C. The priority mechanism allows doing this without having to modify the common configuration dictionary for the B function call (and then to rollback changes for the C function call).

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

IMPORTANT: something has to be taken into account when combining dictionaries and keyword arguments: keyword arguments and dictionaries with keyword **config** shall be always placed as the last arguments of the SPELL function call. It is a Python restriction that any keyword argument has to be at the end of the function argument list.

3.2.4 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 23: 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 24: 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 25: 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.
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 26: 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 27: 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 28: 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 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]`
`-ddd hh:mm[:ss]`
	`+001 23:10`
	`+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
• **HOUR**	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 29: 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 30: 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.

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

Example 31: time management

```
# Initialization with absolute UTC time
# Will store a time object with the given time
mytime1 = TIME('2006/04/02 20:32:34')

# Initialization with relative time
# Will store a time object with the given time
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

# Stores the current type plus 3 hours, 20 secs
t2 = NOW + 3*HOUR + 20*SECOND

# Stores the current day plus 30 minutes
t1 = TODAY + TIME('+00:30:00')

# Stores the difference between the two above
t = t2 – t1

# Stores '2006/04/02 20:33:04'
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 32: 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 'TM param'.

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 33: 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 34: 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 35: 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:

a) **name()**: returns the parameter mnemonic.
b) **description()**: returns the parameter description
c) **fullName()**: returns a combination of mnemonic and description
d) **raw()**: returns the raw value
e) **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 36: 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 perform 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 37: 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 38: verify a single TM value in raw format

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

4.3.2 Last recorded values an 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 39: 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 40: 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 41: 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 42: getting failure information

```
result = Verify( […] )

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

The code above could produce the following output:

Example 43: 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 44: 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 45: 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 46: 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 47: delay the verification

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

4.3.8 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:

a) SKIP: ignore the **False** comparison and continue, giving **True** as return value
b) RECHECK: repeat the TM verification using the next TM parameter sample
c) CANCEL: ignore the false comparison and continue, giving **False** as return value
d) ABORT: abort the procedure execution

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

4.3.9 Verifying multiple parameter values

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

Example 48: 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 49: 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 50: 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.

4.4 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 51: 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 52: 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 53: 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 54: 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 55: 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 56: 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 57: 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

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 unless the command is marked by the GCS as a critical command.

Example 58: confirm command execution

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

4.5.6 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 59: commands with arguments

```
Send( command= tc_item )  # The item contains args

Send( command= 'CMDNAME', args=[…] )
```

4.5.7 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 60: sending sequence

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

4.5.8 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 61: 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 62: 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.

4.5.9 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 63: 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 64: sending a list of commands blocked

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

4.5.10 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 65: sending with timeout

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

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

4.5.11 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.

4.5.12 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 66: 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).

4.5.13 Return value

This function returns **True** unless there is a command failure.

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 67: 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 68: 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 69: 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**)

4.6 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 70: 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 71: wait absolute time condition

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

4.6.3 Stop procedure execution until a telemetry condition is fullfilled

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 72: 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 73: 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 PAUSING the procedure meanwhile the system is waiting for the condition to be fulfilled. If the procedure is put to RUNNING state again, the condition check will be resumed. On the other hand, if the statement is SKIPPED, the condition check is aborted and the next procedure line is executed.

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 74: 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 75: 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 76: 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.

When reading or manipulating limits, the particular definition to be modified shall be specified. This is dony 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.

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 a delta value, detect if there is a step up/down in the TM parameter value bigger than the delta.
- **Spike**: defined by a delta value, detect if there is a change in the TM parameter bigger than the delta.

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.

- **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.

- **Step/Spike:** the dictionaries are { **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 77: reading all limit definitions

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

→{"ID1": { LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4 },
   "ID2": { LoRed:y1, LoYel:y2, HiYel:y3, HiRed:y4 }}
```

In the example the parameter 'PARAM' has two limit definitions, identified by "ID1" and "ID2" respectively. To retrieve the currently applicable definition only:

Example 78: reading applicable limit definitions

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

→{"ID1": { LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4 }}
```

Assuming that "ID1" is the applicable definition at the moment of the call. Notice that 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 79: reading limit definitions by identifier

```
GetLimits( 'PARAM', Select = "ID1" )

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

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

4.8.5 Modifying OOL definitions

The **SetLimits** function is used for this purpose. First, the definition to be modified shall be obtained with **GetLimits** (see previous section). The obtained definition can be manipulated and then passed to **SetLimits** function:

Example 80: modifying limit definitions

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

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

Example 81: modifying limit definitions (2)

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

or

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

Notice that in this second case the pass a definition, not a dictionary of definitions, to the function. Then, the definition to be modified has to be identified with **Select.** *If no definition identifier is provided, the SPELL driver shall modify the first OOL definition found for the TM parameter*:

Example 82: modifying limit definitions (3)

```
mydef={LoRed:x1, LoYel:x2, HiYel:x3, HiRed:x4}
SetLimits( 'PARAM', mydef )

(modifies the first definition found)
```

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 83: 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 84: automatic limit adjustment

```
SetLimits( [['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 four-values Hard-Soft limit definition.

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 85: loading limits from file

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

4.8.9 Checking alarm status

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

Example 86: 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 87: 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.

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 88: event messages

```
Event( 'Message' )

Event( 'Message', WARNING )

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

4.11 Managing target system 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 89: change configuration value

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

Example 90: retrieve configuration value

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

Both functions will fail if the given variable does not exist or cannot be written/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 91: 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 92: types of prompt

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

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

Example 93: 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 94: 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 95: 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 is 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 96: 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 an 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 target system displays

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

Example 97: opening displays

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

The display should open on the same host as the SPELL client. If a different host should be used, the **Host** modifier can change the target host machine:

Example 98: opening displays in a given host

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

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

Example 99: 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 100: display format

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

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 101: 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.

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 102: 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

    - Undefined variable errors
    - '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.

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 103: 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 104: 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 color 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 105: 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 106: 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 107: spacecraft database

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

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

Example 108: spacecraft database keys

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

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

Example 109: 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

Manoeuvre databases provide support for the execution of stationkeeping manoeuvres. Procedures may access manoeuvre 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 manoeuvre message databases is **mmd://**

Example 110: loading a manoeuvre file

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

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

a) 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.

b) The base location corresponding to a given URI is specified in the SPELL configuration files.

c) 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 manoeuvre 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.

Manoeuvre 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 111: 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 112: 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 113: using database mappings (2)

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

4.15.4 User databases

4.15.4.1 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 controller by SPELL. Databases shall be accessed and manipulated using SPELL functions.

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

Example 114: 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.

4.15.4.2 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 115: 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.

4.15.4.3 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 116: 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

4.15.4.4 Reverting changes to user databases

Reverting changes done to an 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 commited until SaveDictionary function is used. To append or modify a database value:

Example 117: modifying a database value

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

4.15.5 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 118: 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 119: 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'
```

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 120: 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 121: 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 122: 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 123: starting procedure using priorities

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

Folder priorities can be overriden 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 124: 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 125: 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 126: 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 127: starting a sub-procedure in manual mode

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

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

Example 128: starting a sub-procedure (complex)

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

4.16.1.5 Return value

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

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
BuildTC	Build a telecommand item	4.4
CreateDictionary	Create a new user database	4.15.4.2
ChangeLanguageConfig	Change default modifier values	3.2.4
DisableAlarm	Disable TM parameter OOL alarms	4.8.6
DisableUserAction	Disable a programmed user action	4.14.4
DismissUserAction	Remove a programmed user action	4.14.4
Display	Display a message	4.9
EnableAlarm	Enable TM parameter OOL alarms	4.8.6
EnableUserAction	Enable a programmed user action	4.14.4
Event	Inject an event	4.10
Finish	Finish the execution successfully	4.14.3
GetResource	Get a configuration parameter	4.11
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
OpenDisplay	Open a TM display	4.13
Pause	Pause the execution	4.14.3
PrintDisplay	Print a TM display	4.13
Prompt	Prompts user for information	4.12
SaveDictionary	Save changes done to a dictionary	4.15.4.3
Send	Send telecommands	4.5
SetGroundParameter	Inject a TM parameter value	4.7
SetUserAction	Program an user action function	4.14.4
SetResource	Set a configuration parameter	4.11
SetLimits	Set TM parameter OOL definitions	4.8
StartProc	Start a subprocedure	4.16
Verify	Verify a set of TM conditions	4.3
WaitFor	Wait for a time or TM condition	4.6

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.
`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`	`True,False`	If True, the user shall confirm explicitly that a telecommand should be sent to the S/C.

`Default`	`Prompt`	`<STRING>,`
		`True/False,`
		`<TIME>`
			Sets the default value for a prompt. Takes effect when used in combination with the Timeout modifier.

`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> (relative)`	Determines the time to wait before starting the TM verification
`Delay`	`WaitFor`	`<TIME> (relative)`	Sets the amount of time (relative) to wait before continuing the execution.
`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`	`<STRING>`	Sets the host name where a TM display is shown.
`IgnoreCase`	`Verify`	`True,False`	If True, it indicates that string comparisons should be case insensitive when performing a TM verification

`Interval`	`WaitFor`	`<TIME> or`
		`list of <TIME>`
		`(relative)`
			Sets the update interval when waiting for a given time condition

`LoadOnly`	`Send`	`True,False`	If True, the sent telecommand is considered as successfully verified as

Modifier	***Function***	***Value***	***Description***
			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
`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 ('|').
`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.
`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.
`Time`	`Send`	`<TIME> (absolute)`	Used for time-tagging commands.
`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

Modifier	***Function***	***Value***	***Description***
			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`	`OK,CANCEL,YES, NO,YES_NO, OK_CANCEL,LIST, ALPHA,LIST|ALPHA, LIST|NUM, NUM`	Determines the type of prompt to be performed.
`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.
`ValueFormat`	`GetTM, Verify`	`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/ BOOLEAN/TIME/ FLOAT`	Forces the type of a telecommand argument value
`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|REPEAT|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

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.