

SPELL Driver Development Manual

SPELL version 2.4.4

Distribution list

Full Report:

SES-SSO-SOE

For Information Purposes:

Open Source

Reference:

SES-SSO-SOE-SPELL-2015/03

Date issued:

February 2015

Acronyms

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

SPELL version 2.4.4

Table of Contents

1	Introduction	5
1.1	Purpose of this document.....	5
1.2	Software Requirements	5
1.3	Build System	5
2	SPELL driver integration	6
2.1	Connection layers	6
2.2	Implementation languages	7
3	Driver Development Facilities	9
3.1	Time management.....	9
3.2	The SPELL language	9
3.3	The registry	9
3.4	Driver failures: exceptions	10
4	Telemetry and telecommand models	12
4.1	Telemetry model.....	12
4.2	Telecommand model.....	13
5	Driver Connection Layer	17
5.1	Partial implementations.....	17
5.2	Telemetry service	17
5.3	Telecommand service	26
5.4	Event service	31
5.5	Resources management service	33

SPELL version 2.4.4

5.6	Time management service	35
5.7	Configuration service	37
5.8	Additional modifiers and constants	38
6	Driver configuration.....	40
6.1	Driver XML configuration file	40
6.2	Server XML configuration file	41
6.3	Context XML configuration file	41
6.4	The “Example” driver	41
7	Build and deployment.....	42
7.1	Deployment rules	42
7.2	Independent builds	42
7.3	Drivers as part of the SPELL build	43
7.4	Driver make file	43
7.5	Packaging scripts	44

1 Introduction

1.1 Purpose of this document

This document provides information for creating new SPELL drivers for ground control systems.

This manual is intended to be used by software teams in order to build new drivers for additional ground control systems not supported already by SPELL.

1.2 Software Requirements

The following packages and libraries comprise the minimum set of software requirements, needed to start working with SPELL drivers:

Package	Version
GNU autotools	1.9
GNU Bash	3.1
GNU Binutils	2.16
GNU Make	3.8
Python interpreter	2.5

Of course, additional requirements should be added to this list depending on the software requirements imposed by the technologies used to develop each driver.

1.3 Build System

The GNU auto-tools have been chosen as the build system for SPELL, since they provide a widely-used, GNU-standard way of compiling, configuring and deploying software in GNU/Linux platforms. Please refer to <http://www.gnu.org/software/autoconf/> and <http://www.gnu.org/software/automake/> for details.

In the following sections, a basic knowledge of what auto-tools are and how do they work will be assumed.

2 SPELL driver integration

SPELL drivers are the software abstraction layer that connects the SPELL language (library) services with the concrete Ground Control System services. Thanks to the SPELL drivers the procedures are, from the language point of view, independent of the type of GCS being used.

The driver is in charge of translating generic service requests into GCS-dependent requests, and in charge of translating the GCS responses and information back to a generic format understandable by SPELL.

The following diagram shows the general structure of the SPELL framework and the way a driver is integrated:

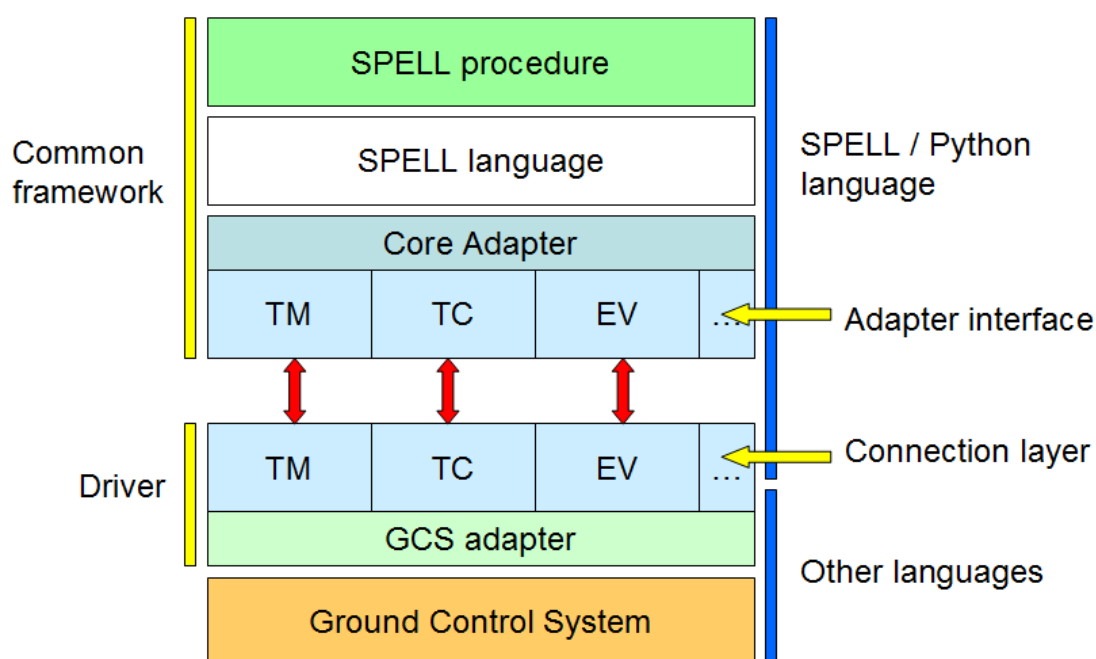


Figure 1: SPELL framework structure

Notice that a majority of the framework is common to all situations, no matter which GCS is being used. The procedures, SPELL language and core services remain the same, no matter which control system is underneath.

2.1 Connection layers

In the diagram shown in Figure 1 the following connection layers between the SPELL framework and the SPELL driver can be seen:

- 1) **The core adapter:** this component is in charge of two tasks. Firstly, it contains the implementation of several services which are independent from the GCS, that is, common to all control systems. Secondly, it acts as the bridge between the SPELL language and the SPELL drivers.
- 2) **The adapter interface:** it is the sub-component of the core adapter which directly connects to the drivers. The connection is made by class inheritance. The adapter interface defines a set of abstract classes and methods that represent the SPELL services required by the language. (An ex-

ample of these services would be methods for acquiring telemetry parameter values). The SPELL drivers must inherit from these classes and implements the abstract methods in order to provide their own service implementations.

- 3) **The driver connection layer:** it is the part of the driver that inherits from the adapter interface and provides specific service implementations, as explained in the previous point. It is in charge of accepting the requests coming from the SPELL language, and forwarding those to the inner layers of the driver for further processing.
- 4) **The driver GCS adapter:** the part of the driver that translates the service requests (received via the driver connection layer) into something understandable by the ground control system. This component communicates directly with the GCS external interfaces, sending and receiving data to and from them.

Notice that both the adapter interface and the driver connection layer are subdivided in service groups, named TM, TC, EV and so on. Each group of service is represented by an adapter class, which is implemented by the corresponding class on the driver connection layer (the inheritance relation is represented by red block arrows in the figure). The service group “TM” contains the services in relation with telemetry parameters; the service group “TC” contains services for telecommand injection, and so forth.


The structure and cycle of calls can be understood easily with an example. Let’s consider the case where a SPELL procedure invokes the GetTM function in order to acquire a telemetry parameter value. The sequence of call would be something like the following:

- 1) The language function GetTM is executed
- 2) GetTM accesses internally the core adapter TM service
- 3) The core adapter may perform some internal processing and generates an internal service request
- 4) The service request is passed to the adapter interface
- 5) The service request arrives to the driver connection layer
- 6) The driver connection layer interprets the request and adapts it to an internal representation
- 7) The connection layer forwards the internal request to the GCS adapter layer
- 8) The adapter layer communicates with the GCS, requesting a TM parameter value in particular
- 9) The GCS answers with the required data
- 10) The GCS adapter layer gives back the data to the driver connection layer
- 11) The connection layer translates the internal data to something understandable by the core adapter
- 12) The data is given back to the adapter interface
- 13) The adapter interface sends the data to the core adapter
- 14) The core adapter sends the data to the GetTM function
- 15) The procedure obtains the required telemetry parameter value

2.2 Implementation languages

As it can be seen in the Figure 1, the SPELL language and core adapter are implemented in Python language, and in some cases they use some additional extensions developed both in Python and C/C++.

In particular, the core adapter interface layer is implemented in Python. As a consequence, the first layer in the SPELL driver that connects to the framework, that is, the driver connection layer, *shall be also implemented in Python*, since the connection between adapter and driver is done by class inheritance.

February 2015	<div>SPELL Build Manual</div> <div>SPELL - Driver Development Manual - 2.4.4.docx</div> <div>  </div>
Page 8 of 45	

But the inner layers of the driver (GCS adapter) can be implemented in other programming languages. The GCS adapter can use CORBA, raw TCP/IP sockets, MySQL connections, pipes, or something else. SPELL does not introduce specific restrictions on the technology to be used in the internal driver implementation.

The sole condition for the GCS adapter implementation language is that there must be a way to join that implementation with the Python language, since the connection layer shall be made in Python.

There are several possibilities for this, including the **Swig** framework (for C/C++), **JPytype** (for Java), **PythonNet** (for .NET), and so on. It is also possible to develop native Python extensions in C/C++ and to use them from within Python code.

Please refer to the chapter 5 for the implementation details of the driver connection layers.

3 Driver Development Facilities

The SPELL framework provides a number of utilities, data and services that can be used in driver implementations.

The most important are the telemetry and telecommand models, which are described in the chapter 4. Other elements provided by the framework are described in the following.

3.1 Time management

The package `spell.lib.adapter.utctime` provides the `TIME` class, which provides general time management services. It also provides predefined time constants like `NOW`, `TODAY`, `SECONDS`, etc (see the language reference [1] for details). These utilities can (and should) be used by drivers in order to work with time: telemetry value acquisition times, command update times, and other similar data.

3.2 The SPELL language

The SPELL language constants and modifiers become available for drivers by importing the packages `spell.lang.constants` and `spell.lang.modifiers` respectively. The drivers need to know the modifiers and constants in order to process the configuration values passed by the core adapter in each service request, in order to change the default behaviour of the driver services when needed.

The drivers **SHALL NOT** use the SPELL functions internally. For accessing certain services like displaying messages and issuing prompt requests, specific mechanisms are provided as described in the next section.

3.3 The registry

The adapter interface components, as well as other services can be accessed at any time from the driver via the *registry*.

The registry is a singleton provided in the package `spell.lib.adapter.registry`. It can be used as a Python dictionary for accessing the adapter components, in the following way:

```
REGISTRY['TM'] → provides the adapter interface TM service
```

The available components are:

Identifier	Service
TM	Telemetry service
TC	Telecommand service
EV	Event service
TIME	Time service
RSC	Resource service
CIF	Client interface
MEM	Memory management service
CTX	Context configuration access
EXEC	Access to executor services

Some of the adapter components may not exist in the registry if the driver does not support that feature. Other components like the CIF, EXEC and CTX are always present as they are provided by the SPELL core. They should not be removed, replaced or modified by drivers.

The client interface allows the driver to interact with the user, by sending messages or issuing prompts. It provides the two following methods:

Method	Description
<code>write()</code>	Send a message to the user with the specified severity and scope
<code>prompt()</code>	Issue a prompt to the user

The `write()` method accepts one or two arguments:

1. **Message text:** string with the message to be sent
2. **Message severity** (optional): severity of the message: `INFORMATION` (default), `WARNING` or `ERROR`.


The `prompt()` method accepts the following arguments:

1. **Prompt message:** the text of the prompt
2. **Prompt option list** (optional): list containing the possible choices.
3. **Prompt configuration** (optional): dictionary with modifiers.

For details on the option list and the configuration dictionary, please check the SPELL language reference manual.

3.4 Driver failures: exceptions

When the driver fails to perform an operation, it is recommended to throw a `DriverException` and let the SPELL core adapter to process it automatically. This type of exception is available on the package `spell.lib.exception`.

February 2015	SPELL Build Manual SPELL - Driver Development Manual - 2.4.4.docx	
Page 11 of 45		

A driver exception accepts two arguments in the constructor: the exception message, and the explanation of the cause, if any:

```
raise DriverException("An error occurred", "Because of ..." )
```

4 Telemetry and telecommand models

Telemetry parameters and Telecommands are represented in two ways in the SPELL framework:

- **Using strings:** the representation includes the parameter/telecommand mnemonic (a.k.a short name), and possibly a more complex string with the format

“T MNEMONIC Description”

(with ‘C’ instead of ‘T’ in the case of commands). This representation is rather simple but limited. For example, it does not allow the definition of command arguments.

- **Using models:** in general, SPELL uses instances of two classes, `TmItemClass` and `TcItemClass` to represent telemetry parameters and commands, respectively. These classes allow the definition of all the aspects regarding parameters and commands, like format, arguments, acquisition time, validity, and so forth.

It is strongly recommended to use the model representation as much as possible, since the adapter uses normally this kind of representation to work and to communicate with drivers. These classes are available in the SPELL packages

```
spell.lib.adapter.tm_item
```

```
spell.lib.adapter.tc_item
```

4.1 Telemetry model

The telemetry item model provides the following methods for accessing data:

Method	Description
<code>name()</code>	Obtain the parameter mnemonic.
<code>setName()</code>	Modify the parameter mnemonic.
<code>description()</code>	Obtain the parameter description.
<code>fullName()</code>	Obtain the mnemonic and description in the form “T Mnemonic Desc”.
<code>value()</code>	Obtain the parameter value. The model stores both the raw and the calibrated value of the parameter. The value given by default is the calibrated one, although this behaviour can be modified with the <code>value()</code> method configuration parameters.
<code>raw()</code>	Obtain the raw parameter value directly (same effect as the <code>value()</code> method when used with the <code>RAW</code> format option).
<code>_setRaw()</code>	Set the parameter raw value.
<code>eng()</code>	Obtain the engineering or calibrated value directly (same effect as the <code>value()</code> method when used with the <code>ENG</code> format option).
<code>setEng()</code>	Set the parameter engineering value.
<code>status()</code>	Obtain the parameter validity. Returns <code>True</code> if the parameter is valid, <code>False</code> otherwise.
<code>_setStatus()</code>	Set the parameter validity.
<code>time()</code>	Obtain the acquisition time.
<code>_setTime()</code>	Set the acquisition time.

The model provides additional methods which will not be covered in this manual, since they should not be used by drivers.

4.1.1 Working with the telemetry model

As it will be described later in this document, when the framework requests the acquisition of new telemetry values, it passes a telemetry item model to the driver. The driver shall communicate with the GCS as explained before, and once it acquires the required data, it must update the parameter model with these.

To do so, the described setter methods are to be used. Typically, a driver shall set the raw value, the engineering value, the acquisition time and the validity status fields.

4.2 Telecommand model

The telecommand item model represents a telecommand or sequence of commands, with argument names and values, when applicable. The model provides the following methods for accessing data:

Method	Description
<code>name()</code>	Obtain the telecommand mnemonic.
<code>setName()</code>	Modify the telecommand mnemonic.
<code>desc()</code>	Obtain the telecommand description.
<code>setDescription()</code>	Modify the telecommand description.
<code>_setExecutionStageStatus()</code>	Set the execution stage and status for the telecommand or one of its elements.
<code>getExecutionStageStatus()</code>	Get the execution stage and status for the telecommand or one of its elements.
<code>_setCompleted()</code>	Sets the completed and success flags of the command or one of its elements.
<code>getIsCompleted()</code>	Check if the command (or one of the elements) is completed.
<code>getIsSuccess()</code>	Check if the command (or one of the elements) is success or failed.

4.2.1 Working with the telecommand model

The SPELL framework uses the telecommand models to keep track of the telecommand execution status during command injection in the GCS. When the execution of a command is requested to the driver, it shall communicate with the GCS in order to (1) have the command sent to the spacecraft, and (2) obtain updates on the status of the command while it is being uplinked, verified and executed.

4.2.1.1 Telecommand elements

In some situations, a telecommand that has a single element representation at procedure level may need to be exploded in a set of several telecommands when being processed by the ground control system. An example of this is the sequences, or command lists. In the procedure code, only one mnemonic can be seen (the name of the sequence). But when injected to the ground control system, it becomes a set of telecommands (namely, a *complex command*), those which are defined as components of the sequence in the GCS TM/TC database.

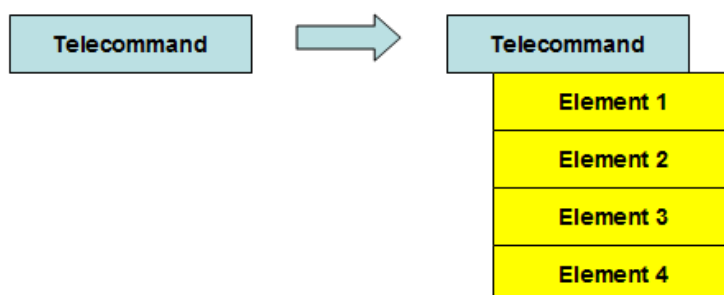


Figure 2: A simple command becomes a complex command once exploded by GCS

These new telecommands are called *elements of the original telecommand model*. SPELL knows nothing about which elements will become part of the original one because this information is obtained from the ground control system at runtime. Therefore it is the driver who is in charge of updating the telecommand model by adding the telecommand elements as soon as they are known.

To do so, the driver can use the following telecommand model methods:

Method	Description
<code>setElements()</code>	Sets the elements of the command model
<code>isComplex()</code>	Returns True if the model contains more than one element

The `_setElements()` method takes a Python list of strings containing the mnemonics of the command elements. Notice that initially, a command model contains a single element, the one corresponding to the original telecommand mnemonic. When a list of N elements is added to the model, the total amount of elements in the model is N+1, being the first element the original mnemonic.

The internal representation of the element is slightly modified by the model. Each element name becomes "index@<mnemonic>", where <mnemonic> is the element name given by the driver. This has to be done in order to be able to differentiate elements with the same mnemonic string, but in different positions of the model.

4.2.1.2 Execution stages

It is assumed in SPELL that a command being executed goes through a set of stages, from the moment it is created by the GCS, until the moment its execution finishes on board of the spacecraft. The same applies for each telecommand element, for complex command models.

The command stages are represented by a pair of strings, being the two elements:

- The stage name
- The stage status

These strings are arbitrary and can be freely defined by the driver. An example of stage-status pairs follows:

Stage name	Possible status for the stage
CREATION	IN PROGRESS, SUCCESS, FAILED
INJECTION	IN PROGRESS, SUCCESS, FAILED
UPLINK	IN PROGRESS, SUCCESS, FAILED
ACCEPT ON BOARD	SUCCESS, FAILED
EXECUTION	IN PROGRESS, SUCCESS, FAILED

Every command would go through the defined stages sequentially during its execution. Once the latest stage is reached in SUCCESS status, or once any stage becomes FAILED, the command execution finishes. The driver shall use the `_setExecutionStageStatus()` method to update the information in the model. The method accepts two, three or four arguments:

1. Stage name string
2. Stage status string
3. Comment (optional, used to add status information for the user)
4. Element identifier (optional), used when the driver wants to change the status of a particular element inside a complex telecommand model. The element identifier shall match the format described in the previous section, "index@<mnemonic>". If no element identifier is used, the given stage and status are assigned to the original command. Of course, no element identifiers shall be used when working with simple commands (with no elements).

The SPELL core adapter uses the command stage and status information to keep the user informed about what is going on during the command execution, but it does not pay attention to them in order to decide whether a command has failed or not. Actually, SPELL does not care about the command status at any time. It is the driver who is in charge of monitoring the command, as it is explained in the next section.

The driver can use the `_setCompleted()` method in order to set the command completion flags. The method accepts one or two arguments:

1. Success (True/False): if True, the command is set as finished successfully. If False, the command is set as finished and failed.
2. Element identifier (optional): used when the driver wants to change the flags of a particular element inside a complex telecommand model. The element identifier shall match the format described in the previous section, "index@<mnemonic>". If no element identifier is used, the flag values are assigned to the original command. Of course, no element identifiers shall be used when working with simple commands (with no elements).

Notice that, when working with complex commands, the overall command execution is considered finished **only** when all contained elements have the completed flag set. Therefore, the driver shall ensure that it is calling the `_setCompleted()` method for every element in the model, at some point.

4.2.1.3 Command injection

When the core adapter invokes the command injection service of the driver, a raw command model is passed. The driver shall update the command model structure when needed, finding out if the command shall become a complex command (according to the information of the GCS), and updating the model elements (see section 0).

The adapter passes a configuration dictionary which may contain modifiers affecting the way the command is treated and/or the way it is verified. See section 0 for details.

4.2.1.4 Command verification

As it has been mentioned, when a command is sent from SPELL, the core adapter just waits until the command injection service request finishes on the driver side. The command monitor and verification needs to be done by the driver, who should hold the core adapter call until the command reaches a “finished” state.

The driver should start updating and checking the model information as soon as the command is passed to the ground control system, in order to determine if the execution has been completed or not, and whether it has been success or not. For these matters, the driver uses the mentioned model methods. During the process, the current status of the command should be acquired from the GCS, and the command model should be updated accordingly.

As mentioned in the previous section, some SPELL modifiers may be passed to the driver in order to change the behaviour of the command verification (see section 5.3.2.3).

5 Driver Connection Layer

As it has been explained in section 2.1, the connection layer is the access point to the SPELL driver software from the adapter side. It is composed of a set of Python classes, each one associated with a particular set of services like telemetry acquisition, telecommand injection and events management.

In this section, the structure of a connection layer is described. SPELL drivers need to implement it in order to get plugged into the SPELL framework.

5.1 Partial implementations

The general rule in the adapter interface is not to force the driver to implement all services and features. The driver connection layers are not obliged to implement all features when inheriting from the adapter interface classes.

If a particular service (method) is not available in the GCS, the adapter method that provides such feature should be left without implementation in the connection layer class. In this case, whenever a SPELL procedure tries to use the service, SPELL will reply with a standard exception reporting that the service required is not available for the current GCS and driver. Of course the functionality in particular cannot be used in procedures but that does not affect other services.

It is also possible that the driver can support a service (method), but not all the possibilities of use of that service. For example, the driver may be able to acquire telemetry parameter values, but it may be not possible to choose if the acquired values shall be presented in engineering (calibrated) format or in raw format. These kinds of options are specified in the configuration parameters passed to the driver connection layer, and whenever one of these options is not supported, it can be just ignored. It is advisable, though, that the driver raises a warning message informing the user about the limitation.

Finally, it is also possible that a driver does not implement one or more whole components of the connection layer. For example, a driver may not provide implementation for event management, but only for telemetry and telecommand management. In this case, the configuration file of the driver shall indicate that the EV component is not available. Again, this does shall not affect the rest of functionalities of the driver.

5.2 Telemetry service

The first layer component is the Telemetry service. It is represented by a class that inherits from the adapter class `TmInterface`, located in the Python package `spell.lib.adapter.tm`. This service is in charge of:

1. **Telemetry acquisition:** given a TM point mnemonic or name, the value of the point shall be acquired from the GCS and given back. There are some configuration parameters that, when provided, indicate the way in which the value should be acquired. For example, it should be possible to obtain the value in raw or un-calibrated format. It is also possible that the SPELL language requires the driver to acquire not the current parameter value, but the next one coming; in this situation the driver shall wait until the parameter sample arrives, and then return the new value.
2. **Telemetry injection:** given a TM point mnemonic or name and a value, the driver shall inject the given value into the GCS as the new value for the telemetry point.

3. **Telemetry point creation:** the adapter provides a generic service to create telemetry point models, but it is possible for the driver to override this creation service in order to enhance it or add additional functionalities to it.

In addition to these services, the driver telemetry interface shall implement some interface control methods.

5.2.1 Telemetry acquisition

5.2.1.1 Method to implement

The telemetry acquisition is performed via the following method of the connection layer:

```
_refreshItem( tmItem<TmItemClass>, config<dict> ): [ value, status ]
```

The first argument 'tmItem' provides a SPELL telemetry parameter model. This model has been already described in the section 4.1.

The second argument 'config' provides a dictionary with SPELL modifiers that indicate the options for the operation. These options indicate how the telemetry acquisition should be performed.

5.2.1.2 Driver duties

The driver shall access the GCS interface in order to obtain the telemetry parameter value. The acquisition shall be performed as specified by the configuration parameters, according to the description in section 5.2.1.3. A `DriverException` shall be thrown in case of errors.

Once the acquisition is done, the driver shall (1) update the information in the given telemetry parameter model, and (2) return the parameter value as described in section 5.2.1.4.

5.2.1.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **Wait=True/False:** when provided and the value is `True`, the driver shall wait until the next sample for the given TM point is received on the GCS, instead of returning the current value of the point. The `_refreshItem` method shall not return until such new value is available.
2. **ValueFormat=RAW/ENG:** when provided, it determines the format of the TM point value to be returned. If its value is `ENG`, the driver shall provide the engineering or calibrated value of the parameter. If its value is `RAW`, the driver shall provide the raw, un-calibrated value of the parameter.
3. **Timeout=<TIME>:** when given, it indicates the maximum amount of time that the driver shall wait for the new sample of a TM point to arrive to the GCS. It only makes sense when the `Wait` modifier is provided with value `True`. If the new sample does not arrive in time, the driver shall raise a `DriverException` reporting the failure to acquire the value in time. Notice that the value for this modifier can be an integer, float or a `TIME` instance.

Other modifiers should be ignored. As explained before, in case the driver does not support any of the options described, it can ignore the modifier but a warning message should be sent to the user to make him/her aware of the limitation. Alternatively, the driver limitation should be well documented.

5.2.1.4 Return value

The method `_refreshItem` shall return a Python list of two elements:

1. Acquired value of the TM point
2. `True/False` depending on the validity of the parameter at the moment of the acquisition

5.2.2 Telemetry injection

5.2.2.1 Method to implement

The telemetry injection is performed via the following method of the connection layer:

```
_injectItem( tmItem<TmItemClass>, value, config<dict> ): bool
```

The first argument '`tmItem`' provides a SPELL telemetry parameter model. This model has been already described in the section 4.1.

The second argument '`value`' contains the value to be injected, and can be a string or number.

The third argument '`config`' provides a dictionary with SPELL modifiers that indicate the options for the operation. These options indicate how the telemetry injection should be performed.

5.2.2.2 Driver duties

The driver shall access the GCS interface in order to inject the telemetry parameter value. The injection shall be performed as specified by the configuration parameters, according to the description in section 5.2.2.3. A `DriverException` shall be thrown in case of errors.

5.2.2.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **ValueFormat=RAW/ENG:** indicates if the given value corresponds to a raw value or a calibrated value.

5.2.2.4 Return value

Once the injection is done the method shall return `True` if the operation was success.

5.2.3 Telemetry creation

5.2.3.1 Method to implement

This method should be implemented when the driver requires doing special operations in relation with telemetry parameters creation; for example, performing a check against the GCS TM/TC database, or modifying the parameter name and/or description according to some GCS rule.

The telemetry creation is performed via the following method of the connection layer:

```
_createTmItem( mnemonic<string>, description<string> ): TmItemClass
```

If this method is overridden, it shall return an instance of the `TmItemClass` model, whose constructor accepts the same parameters as `_createTmItem()`. Once the model is created, the driver can perform other operations.

The first argument, 'mnemonic', corresponds to the name or mnemonic of the telemetry parameter.

The second argument, 'description', contains the parameter description string, or an empty string.

5.2.3.2 Driver duties

As described, the driver shall create an instance of a telemetry parameter model at least. A `DriverException` shall be thrown in case of errors.

5.2.3.3 Relevant configuration parameters

There are no configuration parameters for this service.

5.2.3.4 Return value

An instance of the telemetry parameter mode shall be returned.

5.2.4 Out of Limits acquisition

5.2.4.1 Method to implement

The method to implement in the driver connection layer is used by the adapter in order to retrieve Out of Limit definitions for a given telemetry parameter.

```
_getLimits( tmItem<TmItemClass>, config<dict> ): {}
```

If this method is implemented, it shall return a dictionary containing the limit definitions. Please refer to the SPELL language manual for details on telemetry parameter limits.

The first argument, 'tmItem', corresponds to the telemetry parameter model.

The second argument, 'config', contains the configuration modifiers.

5.2.4.2 Driver duties

The driver shall access the GCS database in order to retrieve the applicable limit definition for the given parameter, and construct and return the dictionary that describes it. A `DriverException` shall be thrown in case of errors.

5.2.4.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **Select=ACTIVE/ALL/<string>**: indicates which OOL definition should be obtained. Depending on the GCS features, each telemetry parameter may have more than one OOL definition associated. Normally, only one is active at a time. When **ACTIVE** is given, the characteristics of the active definition should be obtained. When **ALL** is given, all available definitions should be obtained. Finally, it is also possible to specify the identifier of the definition to be obtained.

5.2.4.4 Return value

A dictionary containing the limit definition(s) should be returned.

In case of having just one definition requested (via **ACTIVE** or the definition identifier), the dictionary keys are the limit parameters (i.e. **LoRed**, **LoYel**, **HiRed**, **HiYel** for analog TM parameters, **Nominal**, **Warning**, **Error** and **Ignore** for status TM parameters, **Delta** for spike or step limits in analog TM parameters). Please refer to the SPELL language reference manual for details.

For the case of several definitions, the dictionary keys are the definition identifiers, and the dictionary values are as well dictionaries with the format described in the previous paragraph.

5.2.5 Acquisition of a single Out of Limits characteristic

5.2.5.1 Method to implement

The method to implement is similar to the one for full OOL definition acquisition, but is used to retrieve a single characteristic of a OOL definition:

```
_getLimit( tmItem<TmItemClass>, limit, config<dict> ): <value>
```

If this method is implemented, it shall return the value corresponding to the limit characteristic requested. Please refer to the SPELL language manual for details on telemetry parameter limits.

The first argument, 'tmItem', corresponds to the telemetry parameter model.

The second argument, 'limit', contains the name of the characteristic to be retrieved (e.g. LoRed, Nominal, Delta, and so on).

The third argument, 'config', contains the configuration modifiers.

5.2.5.2 Driver duties

The driver shall access the GCS database in order to retrieve the applicable limit definition for the given parameter, and obtain the specific characteristic requested. A `DriverException` shall be thrown in case of errors.

5.2.5.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **Select=ACTIVE/ALL/<string>**: same meaning as for full OOL acquisition, described in the previous section.

5.2.5.4 Return value

The value returned depends on the nature of the requested OOL characteristic. If a LoRed value is requested, the value returned is a number. If Nominal is requested, the value returned is a list containing the nominal status values.

For the case of having several definitions requested, the return value is a dictionary whose keys are the OOL definition identifiers, and the dictionary values are the OOL characteristic values in the format described in the previous paragraph.

5.2.6 Out of Limits modification

5.2.6.1 Method to implement

The method to implement in the driver connection layer is used by the adapter in order to modify Out of Limit definitions for a given telemetry parameter.

```
_setLimits( tmItem<TmItemClass>, limits<dict>, config<dict> ): True/False
```

The first argument, 'tmItem', corresponds to the telemetry parameter model.

The second argument, 'limits', contains a dictionary with the OOL definition characteristics.

The third argument, 'config', contains the configuration modifiers.

5.2.6.2 Driver duties

The driver shall access the GCS in order to modify the applicable limit definition for the given parameter. The OOL definition is provided in the form of a dictionary. A `DriverException` shall be thrown in case of errors.

The same format rules apply as for the OOL retrieval. In case of modifying a single definition, the `limits` parameter contains a dictionary with the OOL characteristics (e.g. `LoRed`, `Nominal`, `Delta`). For several OOL definitions, the dictionary contains itself more dictionaries, each one identified by the corresponding definition identifier.

5.2.6.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. `Select=ACTIVE/ALL/<string>`: same as described in previous sections.

5.2.6.4 Return value

The driver shall return `True` when the OOL definition(s) is modified successfully, `False` otherwise.

5.2.7 Modification of single Out of Limits characteristic

5.2.7.1 Method to implement

The method to implement in the driver connection layer is used by the adapter in order to modify a single characteristic of an Out of Limit definition for a given telemetry parameter.

```
_setLimits( tmItem<TmItemClass>, limit, value, config<dict> ): True/False
```

The first argument, '`tmItem`', corresponds to the telemetry parameter model.

The second argument, '`limit`', contains the name of the characteristic to modify.

The third argument, '`value`', contains the value for the characteristic.

The fourth argument, '`config`', contains the configuration modifiers.

5.2.7.2 Driver duties

The driver shall access the GCS in order to modify the characteristic of the applicable limit definition. A `DriverException` shall be thrown in case of errors.

The same format rules apply as for the OOL retrieval. The '`limit`' parameter contains only the identifier of the characteristic (e.g. `LoRed`, `Nominal`, `Delta`) and the '`value`' parameter contains its value. When several OOL definitions are considered, the value is applied to all of them.

5.2.7.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **Select=ACTIVE/ALL/<string>**: same as described in previous sections.

5.2.7.4 Return value

The driver shall return `True` when the OOL definition(s) is modified successfully, `False` otherwise.

5.2.8 Interface control

The telemetry interface shall implement two control methods which are invoked by the SPELL core adapter:

```
setup( ctxConfig, drvConfig )
cleanup()
```

5.2.8.1 Interface setup

The method `setup()` is invoked when the SPELL driver is loaded; that is, when the procedure is loaded or reloaded. Some configuration information is provided via the two arguments, `ctxConfig` (configuration for the current SPELL context) and `drvConfig` (driver configuration parameters).

The context configuration given provides the parameters specified in the context XML configuration file. These parameters can be grouped in two categories: context formal parameters (pre-defined) and driver *context-specific* parameters (arbitrary, depend on the driver).

The context configuration class passed in `ctxConfig` provides the following getter methods:

Method	Description
<code>getName()</code>	Get the context name
<code>getSC()</code>	Get the spacecraft identifier
<code>getDriver()</code>	Get the current driver name
<code>getFamily()</code>	Get the GCS family (usually PRIME/BACKUP)
<code>getGCS()</code>	Get the GCS server name
<code>getDriverParameter()</code>	Get a driver parameter value, given a property name
<code>getDriverParameterList()</code>	Get the list of defined driver parameter names

The driver configuration given provides the parameters specified in the driver XML configuration file as *driver properties*. These properties do not depend on the SPELL context being used. Property names and values can be arbitrary, and are accessed as follows:

```
drvConfig['propertyName'] → gives the property value
```


No matter which operations are performed by the driver in this method, it should call the super class `setup()` method in first place.

5.2.8.2 Interface cleanup

The method `cleanup()` is invoked when the driver is unloaded: when the executor status FINISHED or ABORTED are reached for example.

These methods are the indicated points to initialize (and cleanup) the resources used by the GCS adapter layer. No exceptions should be raised in the `cleanup()` method. No matter which operations are performed by the driver in this method, it should call the super class `cleanup()` method in first place.

5.2.9 Interface instance

The telemetry interface shall be instantiated and stored in a package-global variable called "TM":

```
TM = MyTelemetryInterface()
```

This instance is used by the core adapter while managing and accessing the driver telemetry services.

5.2.10 Telemetry service skeleton

The following code gives an example of a driver TM service implementation skeleton:

```
import spell.lib.adapter.tm.TmInterface as superClass

class MyDriverTmInterface( superClass ):

    def __init__(self):
        superClass.__init__(self)
        <initialization stuff here>

    def setup( self, ctxConfig, drvconfig ):
        superClass.setup(self,ctxConfig,drvConfig)
        <TM resources initialization>

    def cleanup( self ):
        superClass.cleanup(self)
        <TM resources cleanup>

    def _createTmItem( self, mnemonic, description ):
        <tm model creation>
        return instance
```

```

def _refreshItem( self, tmItem, config ):
    <tm value acquisition>
    return [value,validity]

def _injectItem( self, tmItem, value, config ):
    <tm value injection>
    return True/False

# Singleton instance
TM = MyDriverTmInterface()

```

5.3 Telecommand service

The telecommand service is represented by a class that inherits from the adapter class `TcInterface`, located in the Python package `spell.lib.adapter.tc`. This service is in charge of:

1. **Telecommand injection:** given a telecommand mnemonic or name, the command shall be injected in the GCS and uplinked to the spacecraft. There are some configuration parameters that, when provided, indicate the way in which the command injected and its execution monitored.
2. **Telecommand list injection:** similar to command injection, but working with Python lists containing several commands.
3. **Telecommand block injection:** when supported by the GCS, this operation is similar to the list injection, but the commands shall be uplinked as a single frame to the spacecraft and verified as a single entity.

In addition to these services, the driver telecommand interface shall implement some interface control methods.

5.3.1 Telecommand injection

5.3.1.1 Method to implement

The telecommand injection is performed via the following method of the connection layer:

```

_sendCommand( tcItem<TcItemClass>, config<dict> ): bool

```

The first argument 'tcItem' provides a SPELL telecommand model. This model has been already described in the section 4.2.

The second argument 'config' provides a dictionary with SPELL modifiers that indicate the options for the operation. These options indicate how the command injection and verification should be performed.

5.3.1.2 Driver duties

When the core adapter invokes the command injection service of the driver, a raw command model is passed.

The first thing to be done by the driver is to obtain the actual set of commands that is going to be sent to the spacecraft. As it was explained, there may be cases where the original command specified on the SPELL procedure corresponds to a more complex set of commands at the uplink time (sequences are an example of this).

When the original command is received by the driver, it shall contact the GCS to obtain the complex command, if applicable. Once the command is acquired, the adapter Telecommand model shall be updated with the correct information by adding the new elements (see the telecommand model description for details on section 0).

Once the model is ready, the driver shall send the data to the GCS. Some modifiers need to be taken into account in order to know whether the command should be sent as time-tagged, as load-only, or somehow modified. The section 5.3.1.3 describes the relevant modifiers.

When the command(s) is (are) injected, it is the driver duty to ensure the correct uplink and execution. The core adapter just waits until the command injection request finishes on the driver side. During the command verification, the driver should communicate with the GCS in order to acquire or receive status updates of the commands. As well as the updated status is obtained, the command model should be updated accordingly as described in section 4.2.1.2.

Some specific situations need to be taken into account during the verification process:

- Time-tagged commands should be declared as verified as soon as they are loaded, not executed. The same applies for the load-only commands.
- There may be status values given by the GCS which are not relevant for the SPELL procedure. These should be ignored.
- When the command reaches a failed status, a `DriverException` shall be raised.
- Some modifiers may affect the verification, like the `Timeout` modifier.

A `DriverException` shall be thrown in case of errors.

5.3.1.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **Timeout=<TIME>**: specifies the maximum amount of time to wait for the command to reach a finished state. If the command takes more time than the specified, an error should be reported.
2. **Block=True/False**: only considered for command groups. Indicates if the commands shall be sent as a single frame to the spacecraft. May not be supported by the GCS or the S/C platform.
3. **ConfirmCritical=True/False**: when `True`, a prompt should be issued to the user before sending critical commands (see section 5.3.1.5)
4. **LoadOnly=True/False**: if `True`, the verification shall finish as soon as the command is loaded on board.
5. **Time=<TIME>**: indicates that the command shall be time-tagged using the given time. Verification shall finish as soon as the command is loaded on board.
6. **SendDelay=<TIME>**: indicates the amount of time that the GCS (not the driver) should wait before sending the command.
7. **Sequence=True/False**: indicates whether the command shall be treated as a command sequence, whose definition is in the GCS TM/TC database.

Other modifiers should be ignored. As explained, in case the driver does not support any of the options described, it can ignore the modifier but a warning message should be sent to the user to make him/her aware of the limitation. Alternatively, the driver limitation should be well documented.

5.3.1.4 Return value

The method shall return `True` if the command execution is completed and success, `False` otherwise.

5.3.1.5 Critical commands

When the `ConfirmCritical` modifier is passed with value `True`, the driver shall check for critical commands in the model, before actually sending it to the GCS for the uplink. There is a dedicated method that, when implemented by the driver, the core adapter calls at the proper time in order to perform the check:

```
_checkCriticalCommands( tcItemList<list>, config<dict> ): None
```

The first argument, '`tcItemList`', contains a Python list with all the command models to be checked. The driver shall go over that list and check if there is any critical command. If any is found, the driver shall invoke the following adapter method:

```
_confirmExecution( tcItemName, msg = None ): None
```

A telecommand name can be provided as a single argument, or a more complex message can be provided in the optional argument '`msg`'. The invocation of this method makes the system to issue a prompt to the user asking for confirmation. If the user decides to cancel the execution, an internal exception is thrown and the command injection is aborted.

It is strongly recommended to invoke the execution confirmation method only once, even when there is more than one critical command in the list of models to be checked. In this case, a complex message should be constructed with a list of command names, and it should be given as argument for the confirmation method.

5.3.2 Telecommand list injection

5.3.2.1 Method to implement

The list injection is performed via the following method of the connection layer:

```
_sendList( tcItemList<list>, config<dict> ): bool
```

The first argument, 'tcItemList', provides a list of SPELL telecommand models. The second argument 'config' provides a dictionary with SPELL modifiers that indicate the options for the operation.

5.3.2.2 Driver duties

The procedure is similar to the one for single command injection. Typically, the driver should create a complex Telecommand model named "GROUP" and define as its elements the set of exploded commands which are part of the command list. The verification and monitor process and considerations are then similar to the one for simple command injection.

5.3.2.3 Relevant configuration parameters

The modifiers applicable for command injection are the same as for simple injection. There is a difference though: each telecommand model may contain specific configuration parameters, which override the global configuration parameters given in the method configuration dictionary parameter.

Therefore, the general configuration modifiers shall be applied, except in the cases where specific modifiers are given for a particular command. The command-specific configuration dictionary can be accessed via the following methods of the command model:

Method	Description
getConfig(key)	Get the value of the given modifier if given, or None
hasConfig(key)	Check if the given modifier exist in the command model configuration

5.3.2.4 Return value

Once the injection is done for all commands the method shall return `True` if the operation was success.

5.3.3 Telecommand blocks injection

The injection of command blocks is analogue to command lists. When supported by the ground control system, the driver shall just indicate to the GCS that the command list should be sent and monitored as a block. From the SPELL point of view, there should be no difference. Command block injection is requested by adding the `Block` modifier with value `True` to the global injection configuration dictionary.

5.3.4 Interface control

The telecommand interface shall implement two control methods which are invoked by the SPELL core adapter. These methods play exactly the same role as in the TM interface (see section 5.2.8)

```

setup( ctxConfig, drvConfig )
cleanup()

```

5.3.5 Interface instance

The telecommand interface shall be instantiated and stored in a package-global variable called "TC":

```
TC = MyTelecommandInterface()
```

This instance is used by the core adapter while managing and accessing the driver telecommand services.

5.3.6 Telecommand service skeleton

The following code gives an example of a driver TC service implementation skeleton:

```
import spell.lib.adapter.tc.TcInterface as superClass
class MyDriverTcInterface( superClass ):

    def __init__(self):
        superClass.__init__(self)
        <initialization stuff here>

    def setup( self, ctxConfig, drvconfig ):
        superClass.setup(self,ctxConfig,drvConfig)
        <TC resources initialization>

    def cleanup( self ):
        superClass.cleanup(self)
        <TC resources cleanup>

    def _sendCommand( self, tcItem, config ):
        <tc injection and verification>
        return True/False

    def _sendList( self, tcItemList, config ):
        <tc list injection and verification>
        return True/False

    def _sendBlock( self, tcItemList, config ):
        <tc list injection and verification>
        return True/False

    def _checkCriticalCommands( self, tcItemList, config ):
        <check for critical commands>
        self._confirmExecution( msg )
        return

# Singleton instance
TC = MyDriverTcInterface()
```

5.4 Event service

The event service is represented by a class that inherits from the adapter class `EvInterface`, located in the Python package `spell.lib.adapter.ev`. This service is in charge of:

1. **Event injection:** given an event message and severity, inject the message into the ground control system event log book.

In addition to this service, the driver event interface shall implement some interface control methods.

5.4.1 Event injection

5.4.1.1 Method to implement

The event injection is performed via the following method of the connection layer:

```
_raiseEvent( message<string>, config<dict> ): bool
```

The first argument 'message' provides the event message string.

The second argument 'config' provides a dictionary with SPELL modifiers that indicate the options for the operation. These options indicate how the event injection should be performed.

5.4.1.2 Driver duties

The driver shall send the event message to the GCS in order to have it published in the GCS logbook. In SPELL, event messages have an associated severity (`INFORMATION`, `WARNING` and `ERROR`). The GCS may support other event attributes. It is the duty of the driver to translate the SPELL attributes into GCS attributes when needed.

A `DriverException` shall be thrown in case of errors.

5.4.1.3 Relevant configuration parameters

The modifiers that can be used for this service are:

1. **Severity=INFORMATION|WARNING|ERROR:** specifies the event severity.

Other modifiers should be ignored.

5.4.1.4 Return value

The method shall return `True` if the event injection is completed and success, `False` otherwise.

5.4.2 Interface control

The event interface shall implement two control methods which are invoked by the SPELL core adapter:

```
setup( ctxConfig, drvConfig )
cleanup()
```

These methods play exactly the same role as in the telemetry interface (see section 5.2.8)

5.4.3 Interface instance

The event interface shall be instantiated and stored in a package-global variable called “EV”:

```
EV = MyEventInterface()
```

This instance is used by the core adapter while managing and accessing the driver event services.

5.4.4 Event service skeleton

The following code gives an example of a driver EV service implementation skeleton:

```
import spell.lib.adapter.tc.EvInterface as superClass

class MyDriverEvInterface( superClass ):

    def __init__(self):
        superClass.__init__(self)
        <initialization stuff here>

    def setup( self, ctxConfig, drvconfig ):
        superClass.setup(self,ctxConfig,drvConfig)
        <EV resources initialization>

    def cleanup( self ):
        superClass.cleanup(self)
        <EV resources cleanup>

    def _raiseEvent( self, message, config ):
        <event injection>
        return True/False

# Singleton instance
EV = MyDriverEvInterface()
```


5.5 Resources management service

The resources management service is represented by a class that inherits from the adapter class `ResourceInterface`, located in the Python package `spell.lib.adapter.resources`. This service is in charge of:

1. **Resource value acquisition:** given a resource name, obtain its current value on the GCS.
2. **Resource value injection:** given a resource name and a value, set the new value for the resource in the GCS.

In addition to this service, the driver resource interface shall implement some interface control methods.

5.5.1 Resource injection

5.5.1.1 Method to implement

The resource injection is performed via the following method of the connection layer:

```
_setResource( name<string>, value, config<dict> ): bool
```

The first argument 'name' provides the resource name. The second argument 'value' contains the resource value to be set.

The third argument 'config' provides a dictionary with SPELL modifiers that indicate the options for the operation. These options indicate how the resource injection should be performed.

5.5.1.2 Driver duties

A "resource" from the SPELL point of view is a pair name-value that defines a configuration value in the GCS, for example, a flag that determines how certain type of telemetry is processed, or a switch that controls a GCS feature.

The driver shall communicate with the GCS interface and provide the resource name and new value. Once the new value is correctly set, the driver shall return the control to the adapter. A `DriverException` shall be thrown in case of errors.

5.5.1.3 Relevant configuration parameters

There are no relevant modifiers for this service.

5.5.1.4 Return value

The method shall return `True` if the resource injection is completed and success, `False` otherwise.

5.5.2 Resource acquisition

5.5.2.1 Method to implement

The resource acquisition is performed via the following method of the connection layer:

```
_getResource( name<string>, config<dict> ): bool
```

The first argument 'name' provides the resource name.

The second argument 'config' provides a dictionary with SPELL modifiers that indicate the options for the operation. These options indicate how the resource acquisition should be performed.

5.5.2.2 Driver duties

The driver shall communicate with the GCS interface and provide the resource name in order to acquire the current value. A `DriverException` shall be thrown in case of errors.

5.5.2.3 Relevant configuration parameters

There are no relevant modifiers for this service.

5.5.2.4 Return value

The method shall return the acquired resource value.

5.5.3 Interface control

The resource interface shall implement two control methods which are invoked by the SPELL core adapter:

```
setup( ctxConfig, drvConfig )  
cleanup()
```

These methods play exactly the same role as in the telemetry interface (see section 5.2.8)

5.5.4 Interface instance

The resource interface shall be instantiated and stored in a package-global variable called "RSC":

```
RSC = MyResourceInterface()
```

This instance is used by the core adapter while managing and accessing the driver resource services.

5.5.5 Resource management service skeleton

The following code gives an example of a driver RSC service implementation skeleton:

```
import spell.lib.adapter.resources.ResourceInterface as superClass

class MyDriverResourceInterface( superClass ):

    def __init__(self):
        superClass.__init__(self)
        <initialization stuff here>

    def setup( self, ctxConfig, drvconfig ):
        superClass.setup(self,ctxConfig,drvConfig)
        <RSC resources initialization>

    def cleanup( self ):
        superClass.cleanup(self)
        <RSC resources cleanup>

    def _setResource( self, name, value, config ):
        <resource injection>
        return True/False

    def _getResource( self, name, config ):
        <resource acquisition>
        return value

# Singleton instance
RSC = MyDriverResourceInterface()
```

5.6 Time management service

The time management service is represented by a class that inherits from the adapter class `TimeInterface`, located in the Python package `spell.lib.adapter.gcstime`. This service is in charge of:

1. **UTC time acquisition:** obtain the current UTC time according to the GCS.

In addition to this service, the driver time management interface shall implement some interface control methods.

If this service is not implemented by the driver, the system will use the local UTC time on the SPELL host machine as the reference time.

5.6.1 Time acquisition

5.6.1.1 Method to implement

The time acquisition is performed via the following method of the connection layer:

```
_getUTC(): datetime.datetime
```

5.6.1.2 Driver duties

The driver shall communicate with the GCS in order to obtain the UTC time information. Then, a `datetime` object from the Python module `datetime` shall be constructed and given back to the adapter. A `DriverException` shall be thrown in case of errors.

5.6.1.3 Relevant configuration parameters

There are no relevant modifiers for this service.

5.6.1.4 Return value

The method shall return a `datetime` object with the correct time information.

5.6.2 Interface control

The time interface shall implement two control methods which are invoked by the SPELL core adapter:

```
setup( ctxConfig, drvConfig )  
cleanup()
```

These methods play exactly the same role as in the telemetry interface (see section 5.2.8)

5.6.3 Interface instance

No interface instance is required for the time management.

5.6.4 Time management service skeleton

The following code gives an example of a driver TIME service implementation skeleton:

```

import spell.lib.adapter.gcstime.TimeInterface as superClass

class MyDriverTimeInterface( superClass ):

    def __init__(self):
        superClass.__init__(self)
        <initialization stuff here>

    def setup( self, ctxConfig, drvconfig ):
        superClass.setup(self,ctxConfig,drvConfig)
        <TIME resources initialization>

    def cleanup( self ):
        superClass.cleanup(self)
        <TIME resources cleanup>

    def _getUTC( self ):
        <obtain UTC time>
        return <datetime>

```

5.7 Configuration service

The configuration interface is the entry point of the driver connection layer, meaning that the core adapter calls the `setup()` method of this interface in first place when the driver is about to be loaded. It also calls the `cleanup()` method of this interface in the last place when the driver is unloaded.

Therefore, the `setup()` and `cleanup()` methods are the indicated places to allocate or prepare general driver resources (which might be used by all other driver interfaces), and to release those resources respectively.

As an example, the configuration service `setup()` method could be the place where a MySQL database connection is prepared, or where a CORBA infrastructure is set up. Then, other interfaces like TM or TC can use these resources. On the other hand, the `cleanup()` method would release that database connection or the CORBA middleware resources.

The configuration interface shall just provide the two control methods which are invoked by the SPELL core adapter:

```

setup( ctxConfig, drvConfig )
cleanup()

```

5.7.1 Configuration service skeleton

The following code gives an example of a driver configuration service implementation skeleton:

```

import spell.lib.adapter.config.ConfigInterface as superClass

class MyDriverConfigInterface( superClass ):

    def __init__(self):
        superClass.__init__(self)
        <initialization stuff here>

    def setup( self, ctxConfig, drvconfig ):
        superClass.setup(self,ctxConfig,drvConfig)
        <gcs adapter initialization>

    def cleanup( self ):
        superClass.cleanup(self)
        <gcs adapter resources cleanup>

```

5.8 Additional modifiers and constants

The drivers may define additional functions, modifiers and constants, although this is not recommended since it breaks the “be-generic” rule of SPELL. Obviously, when SPELL procedures are implemented using modifiers which are driver-specific, it is not possible to run these procedures with other GCS than the one associated with the driver that provides these modifiers. Put in a simple way, the procedures will not be generic.

To define new functions, modifiers and constants, the driver may include extra Python modules as part of the driver connection layer: “functions.py”, “constants.py” and “modifiers.py”.

5.8.1 Functions module

This module can contain any SPELL/Python function definition.

```

def MyDriverFunction(...):
    ---

```

The function will become automatically available for procedures. It is a good practice to check first the main module `spell.lang.functions` in order to ensure that no functions are redefined by the driver.

5.8.2 Constants module

This module can contain any constant definition in the form

```

CONSTANT = <Value>

```

The constant will become automatically available for procedures. The constants module **SHALL NOT** import other SPELL modules at all. It is a good practice to check first the main module `spell.lang.constants` in order to ensure that no constants are redefined by the driver.

5.8.3 Modifiers module

This module can contain any constant definition in the form

```
ModifierName = 'ModifierName'
```

The modifier will become automatically available for procedures. The modifiers module **SHALL NOT** import other SPELL modules at all. It is a good practice to check first the main module `spell.lang.modifiers` in order to ensure that no constants are redefined by the driver.

It is important to follow the shown syntax, where the string value matches the variable name.

6 Driver configuration

Each SPELL driver has an associated XML configuration file where special parameters and values can be specified, in order to adjust the driver behaviour. This file contains also some parameters needed by the SPELL core adapter for loading the driver properly. The driver configuration file deployment place is described in the next chapter.

The drivers may also interact with the SPELL Context configuration (see the `setup()` methods in the driver connection layer classes, as described in previous sections).

6.1 Driver XML configuration file

An example of a driver configuration file for a driver is shown below:

```
<driver id="example">

  <name>Example SPELL driver</name>

  <interfaces>TM,TC,EV,TIME,RSC</interfaces>

  <maxproc>5</maxproc>

  <path>$SPELL_HOME/drivers</path>

  <properties>
    <property name="Foo">Bar</property>
  </properties>

</driver>
```

The parameters are described in the following.

- **Driver identifier:** the “id” attribute of the driver tag. It is the global identifier of the driver within the SPELL framework.
- **Driver name:** the “name” XML tag. Contains a descriptive name of the driver.
- **Interfaces provided:** the “interfaces” tag. Contains the identifiers of the core adapter interfaces that are implemented by the driver. In the example, the five interfaces described in the previous chapter are implemented by the driver. If the driver provided only support for telemetry and telcommand, this list would contain the names TM and TC.
- **Maximum accepted procedures:** the “maxproc” tag. Determines the maximum amount of procedures that can be open at the same time with this driver.
- **Path:** the “path” tag. The list of paths that should be included on the system library path in order to enable the driver load.
- **Driver properties.** Any pair name-value can be included here for specifying driver configuration parameters.

6.2 Server XML configuration file

In order to make a driver available for use, it needs to be referenced in a SPELL server configuration file, in the `drivers` section:

```
<drivers>
  <driver>driver_standalone.xml</driver>
  <driver>driver_example.xml</driver>
</drivers>
```

The tags shall reference the name of the driver XML file inside the SPELL configuration directory.

6.3 Context XML configuration file

There may be driver configuration parameters which are defined per-spacecraft. That is, they are context-specific. These configuration parameters can be provided in the SPELL context configuration file, in the section `driverconfig`:

```
<driverconfig>
  <property name="Foo">Bar</property>
</driverconfig>
```

These parameters are conceptually similar to the driver properties specified in the driver XML configuration file, but they may be different depending on which context (spacecraft) is used.

6.4 The “Example” driver

In the SPELL source code workspace a driver named “example” can be found. This driver serves as a template for developing new drivers, so it is strongly recommended to use it as the starting point. It contains an empty implementation of a basic connection layer element.

7 Build and deployment

The SPELL drivers can be compiled, packaged and deployed as part of the SPELL framework or in an independent way. Both alternatives are described in this section.

7.1 Deployment rules

No matter which alternative is chosen for building and packaging the drivers, the way of installing a driver inside the SPELL framework is always the same:

- The driver XML configuration file shall be placed in the SPELL configuration directory, under the “spell” directory. Typically, the corresponding path is \$SPELL_HOME/config/spell. Where SPELL_HOME is an environment variable identifying the SPELL framework installation directory.
- The driver shall be installed in a directory, whose name is the driver name, in the \$SPELL_HOME/drivers directory.
- The driver connection layer files (tm.py, tc.py, config.py and so on) shall be located directly in the driver directory, not in subdirectories.
- The driver directory shall contain a Python module initialization file, __init__.py.
- The driver XML configuration file shall be referenced in a SPELL server XML configuration file in order to be able to use the driver with SPELL procedures (see previous chapter).
- Driver configuration parameters may be included in SPELL context XML configuration files (see previous chapter).

There may be other files and subdirectories within the driver directory, there are no constraints for this. In a nutshell, the general file structure for a driver, once installed on the SPELL drivers directory, should look like the following:

```

SPELL_HOME
  drivers
    drivename
      __init__.py
      config.py
      tm.py
      tc.py
      ...
  config
    spell
      driver_drivename.xml

```

7.2 Independent builds

Drivers can be implemented and built in any kind of development environment. The only constraint imposed by SPELL is the location of the Python classes composing the driver connection layer, as it has been described in the previous section.

7.3 Drivers as part of the SPELL build

Drivers can be implemented and built as part of the SPELL framework. In this case, some rules regarding the directory layout and other configurations shall be followed.

The source files of drivers which are integrated into the SPELL framework are located in the “drivers” directory within the SPELL source code workspace:

```
workspace
  drivers
    example          Example driver (template)
    standalone       Standalone driver (for simulations)
    hifly2           hifly® GCS driver 2.0
    scorpio          Scorpio® GCS driver
  Makefile.am       Auto-tools make template
```

In the example, four drivers can be seen. As mentioned above, the file structure of a driver source code directory shall follow a few rules:

```
drivers/
  example/
    config/
      driver_example.xml
    src/
      __init__.py
      config.py
      tm.py
      tc.py
      ...
      other_stuff/
  Makefile.am
```

- **Config** directory: shall contain the driver XML configuration file.
- **Source** directory: shall contain, at the first level, the GCS connection layer Python modules.
- **Makefile**: specifies how the driver source files shall be compiled.
- **Other files**: within the source directory, other files and folders may exist.

7.4 Driver make file

As it has been mentioned, SPELL is built using the GNU auto-tool system. Therefore, for each software module an Automake Makefile template (`Makefile.am`) shall be defined. This file defines basically three targets:

- **all-local**: holds the instructions for building the driver source code
- **clean-local**: holds the instructions for cleaning up the driver build files
- **install-exec-hook**: instructions for installing the driver in the SPELL home directory.

Please refer to the GNU auto-tools documentation for details about make templates and build targets. The SPELL build manual [2] gives also details about the SPELL build system and the configuration files required for building the framework.

The `Makefile.am` can be written by taking the one for the “example” driver as a template. A typical file would be the following:

```
DRIVER=example
DRIVER_SRC=src
DRIVER_CFG=config

all-local:
    if [[ ! -d ${DRIVER_SRC} ]]; then\
        cp -r ${top_srcdir}/drivers/${DRIVER}/${DRIVER_SRC} . ;\
    fi
    ${top_srcdir}/py-compiledir ${DRIVER_SRC}

clean-local:
    find ${DRIVER_SRC} -name "*.pyc" | xargs rm -f

install-exec-hook:
    if [[ ! -d ${exec_prefix}/drivers ]]; then\
        mkdir ${exec_prefix}/drivers ;\
    fi
    rm -rf ${exec_prefix}/drivers/${DRIVER}
    cp -r ${DRIVER_SRC} ${exec_prefix}/drivers/${DRIVER}
    find ${exec_prefix}/drivers/${DRIVER} -name ".py" | xargs rm -f
    mkdir -p ${exec_prefix}/config/spell
    cp -f ${DRIVER_CFG}/driver_${DRIVER}.xml ${exec_prefix}/config/spell/.
```

A corresponding entry for the 'configure.ac' file, and the corresponding subdirectories in the 'Makefile.am' file of the 'drivers' directory should be added as well.

7.5 Packaging scripts

7.5.1 Packaging script

The script can be found in the `SPELL_HOME/bin` directory, and is named `SPELL-PackageDriver`. It accepts three arguments:

- **Driver name:** the identifier that will be used to refer to the driver in the framework.
- **Driver directory:** path to the directory where the GCS connection layer interfaces are
- **Config directory:** path to the directory where the XML configuration file is located.

```
SPELL-PackageDriver mydriver /path/to/my/driver/src /path/to/my/driver/config
```

The script will package all the files found under the given directories in a tar.gz file, already prepared to be deployed.

7.5.2 Deployment script

The script can be found in the `SPELL_HOME/bin` directory, and is named `SPELL-InstallDriver`. It accepts two arguments:

- **Driver package:** path to a driver tar.gz package created with the packaging script.
- **SPELL home:** path to a valid SPELL installation directory.

```
SPELL-InstallDriver driver_mydriver.tar.bz2 /home/spell/SPELL
```

The driver should appear inside the “drivers” directory after the script execution.