The legHAL package

Description of the Matlab Interface to the REMOTE module

Version 1.0

Document History		
Version	**Reason of the version**	**Paragraph/Page**
REV 1.0	Creation	

Approvals	
Authors	**Validation**
Benjamin Robert *Ultrasound Engineer*	

Sommaire

1 Introduction 9

1.1 Principle 9

1.2 Requirements 9

1.3 The Matlab concept of classes 10

1.3.1 Principles of Matlab classes 10

1.3.2 Program structure 11

1.4 The legHAL package architecture 13

2 The *common* package 14

2.1 The *common.constants* class 15

2.2 The *common.legHAL* class 15

2.2.1 The *common.legHAL* variables 16

2.2.2 The *common.legHAL.GetException* method 16

2.3 The *common.parameter* class 16

2.3.1 The *common.parameter* constructor (public) and the *initialize* method (protected) 17

2.3.2 The *setValue* method (public) 18

2.3.3 The *getValue* method (public) 18

2.3.4 The *isempty* method (public) 18

2.4 The *common.remotepar* class (*common.parameter* subclass) 18

2.4.1 The *common.remotepar* constructor (public) and the *initialize* method (protected) 19

2.4.2 Inherited methods 19

2.5 The *common.object* class 19

2.5.1 The *common.object* constructor (public) and the *initialize* method (protected) 20

2.5.2 The *addParam* method (protected) 21

2.5.3 The *isParam* method (public) 22

2.5.4 The *setParam* method (public) 22

2.5.5 The *getParam* method (public) 23

2.5.6 The *isempty* method (public) 23

2.6 The *common.remoteobj* class (*common.object* subclass) 23

2.6.1 The *common.remoteobj* constructor (public) 23

2.6.2 The *buildRemote* method (public) 24

2.6.3 Inherited methods 24

3 REMOTE 25

3.1 The *remote.dmacontrol* class (*common.remoteobj* subclass) 25
3.1.1 The *remote.dmaControl* constructor (public) and the *initialize* method (protected) 26
3.1.2 The *buildRemote* method (public) 27
3.1.3 Inherited methods 27
3.2 The *remote.event* class (*common.remoteobj* subclass) 27
3.2.1 The *remote.event* constructor (public) and the *initialize* method (protected) 28
3.2.2 The *buildRemote* method (public) 29
3.2.3 Inherited methods 30
3.3 The *remote.fc* class (*common.remoteobj* subclass) 30
3.3.1 The *remote.fc* constructor (public) and the *initialize* method (protected) 31
3.3.2 The *buildRemote* method (public) 32
3.3.3 Inherited methods 32
3.4 The *remote.hvmux* class (*common.remoteobj* subclass) 32
3.4.1 The *remote.hvmux* constructor (public) and the *initialize* method (protected) 33
3.4.2 The *buildRemote* method (public) 34
3.4.3 Inherited methods 34
3.5 The *remote.mode* class (*common.remoteobj* subclass) 34
3.5.1 The *remote.mode* constructor (public) and the *initialize* method (protected) 34
3.5.2 The *buildRemote* method (public) 35
3.5.3 Inherited methods 36
3.6 The *remote.rx* class (*common.remoteobj* subclass) 36
3.6.1 The *remote.rx* constructor (public) and the *initialize* method (protected) 37
3.6.2 The *buildRemote* method (public) 38
3.6.3 Inherited methods 39
3.7 The *remote.tgc* class (*common.remoteobj* subclass) 39
3.7.1 The *remote.tgc* constructor (public) and the *initialize* method (protected) 40
3.7.2 The *buildRemote* method (public) 41
3.7.3 Inherited methods 41
3.8 The *remote.tpc* class (*common.remoteobj* subclass) 42
3.8.1 The *remote.tpc* constructor (public) and the *initialize* method (protected) 42
3.8.2 The *buildRemote* method (public) 43
3.8.3 Inherited methods 43
3.9 The *remote.tw* class (*common.remoteobj* subclass) 44
3.9.1 The *remote.tw* constructor (public) and the *initialize* method (protected) 44
3.9.2 The *buildRemote* method (public) 45
3.9.3 The *setApodization* method (protected) 46
3.9.4 Inherited methods 46

3.10 The *remote.tw_arbitrary* class (*remote.tw* subclass) 46
3.10.1 The *remote.tw_arbitrary* constructor (public) and the *initialize* method (protected) 47
3.10.2 The setWaveform method (protected) 48
3.10.3 Inherited methods 48
3.11 The *remote.tw_pulse* class (*remote.tw* subclass) 48
3.11.1 The *remote.tw_pulse* constructor (public) and the *initialize* method (protected) 49
3.11.2 The setWaveform method (protected) 50
3.11.3 Inherited methods 50
3.12 The *remote.tx* class (*common.remoteobj* subclass) 51
3.12.1 The *remote.tx* constructor (public) and the *initialize* method (protected) 51
3.12.2 The *buildRemote* method (public) 52
3.12.3 Inherited methods 52
3.13 The *remote.tx_arbitrary* class (*remote.tx* subclass) 53
3.13.1 The *remote.tx_arbitrary* constructor (public) and the *initialize* method (protected) 53
3.13.2 The setDelays method (protected) 54
3.13.3 Inherited methods 54
3.14 The *remote.tx_flat* class (*remote.tx* subclass) 54
3.14.1 The *remote.tx_flat* constructor (public) and the *initialize* method (protected) 55
3.14.2 The setDelays method (protected) 56
3.14.3 Inherited methods 56
3.15 The *remote.tx_focus* class (*remote.tx* subclass) 57
3.15.1 The *remote.tx_focus* constructor (public) and the *initialize* method (protected) 57
3.15.2 The setDelays method (protected) 58
3.15.3 Inherited methods 59
4 ELUSEV 59
4.1 The *elusev.elusev* class (*common.remoteobj* subclass) 60
4.1.1 The *elusev.elusev* constructor (public) and the *initialize* method (protected) 60
4.1.2 The *buildRemote* method (public) 61
4.1.3 Inherited methods 62
4.2 The *elusev.arbitrary* class (*elusev.elusev* subclass) 62
4.2.1 The *elusev.arbitrary* constructor (public) and the *initialize* method (protected) 63
4.2.2 The *buildRemote* method (public) 64
4.2.3 Inherited methods 65
4.3 The *elusev.dort* class (*elusev.elusev* subclass) 66
4.3.1 ELUSEV.DORT (ELUSEV.ELUSEV) 66
4.3.2 ELUSEV.DORT.INITIALIZE (PROTECTED) 67
4.3.3 ELUSEV.DORT.BUILDREMOTE (PUBLIC) 68

4.3.4 Inherited methods 68
4.4 The *elusev.flat* class (*elusev.elusev* subclass) 69
4.4.1 ELUSEV.FLAT (ELUSEV.ELUSEV) 69
4.4.2 ELUSEV.FLAT.INITIALIZE (PROTECTED) 70
4.4.3 ELUSEV.FLAT.BUILDREMOTE (PUBLIC) 72
4.4.4 Inherited methods 72
4.5 The *elusev.hifu* class (*elusev.elusev* subclass) 72
4.5.1 ELUSEV.HIFU (ELUSEV.ELUSEV) 72
4.5.2 ELUSEV.HIFU.INITIALIZE (PROTECTED) 74
4.5.3 ELUSEV.HIFU.BUILDREMOTE (PUBLIC) 74
4.5.4 ELUSEV.HIFU.BUILDCAVITATION (PROTECTED) 75
4.5.5 ELUSEV.HIFU.BUILDEMISSION (PROTECTED) 75
4.5.6 Inherited methods 75
4.6 The *elusev.lteimaging* class (*elusev.elusev* subclass) 75
4.6.1 ELUSEV.LTEIMAGING (ELUSEV.ELUSEV) 75
4.6.2 ELUSEV.FLAT.LTEIMAGING (PROTECTED) 77
4.6.3 ELUSEV.LTEIMAGING.BUILDREMOTE (PUBLIC) 78
4.6.4 Inherited methods 78
4.7 The *elusev.push* class (*elusev.elusev* subclass) 78
4.7.1 ELUSEV.PUSH (ELUSEV.ELUSEV) 78
4.7.2 ELUSEV.PUSH.INITIALIZE (PROTECTED) 80
4.7.3 ELUSEV.PUSH.BUILDREMOTE (PUBLIC) 81
4.7.4 Inherited methods 81
5 ACMO 81
5.1 The *acmo.acmo* class (*common.remoteobj* subclass) 82
5.1.1 ACMO.ACMO (COMMON.REMOTEOBJ) 82
5.1.2 ACMO.ACMO.INITIALIZE (PROTECTED) 83
5.1.3 ACMO.ACMO.BUILDREMOTE (PUBLIC) 83
5.1.4 Inherited methods 84
5.2 The *acmo.photoacoustic* class (*acmo.acmo* subclass) 84
5.2.1 ACMO.PHOTOACOUSTIC (ACMO.ACMO) 84
5.2.2 ACMO.PHOTOACOUSTIC.INITIALIZE (PROTECTED) 85
5.2.3 ACMO.PHOTOACOUSTIC.BUILDREMOTE (PUBLIC) 86
5.2.4 Inherited methods 87
5.3 The *acmo.ultrafast* class (*acmo.acmo* subclass) 87
5.3.1 ACMO.ULTRAFAST (ACMO.ACMO) 87
5.3.2 ACMO.ULTRAFAST.INITIALIZE (PROTECTED) 89

5.3.3 ACMO.ULTRAFAST.BUILDREMOTE (PUBLIC) 90
5.3.4 Inherited methods 90

6 USSE 91

6.1 The *usse.usse* class (*common.remoteobj* subclass) 91
6.1.1 USSE.USSE (COMMON.REMOTEOBJ) 91
6.1.2 USSE.USSE.INITIALIZE (PROTECTED) 92
6.1.3 USSE.USSE.SELECTPROBE (PUBLIC) 93
6.1.4 USSE.USSE.SELECTHARDWARE (PUBLIC) 93
6.1.5 USSE.USSE.BUILDREMOTE (PUBLIC) 93
6.1.6 USSE.USSE.BUILDINFO (PROTECTED) 93
6.1.7 USSE.USSE.INITIALIZEREMOTE (PUBLIC) 93
6.1.8 USSE.USSE.LOADSEQUENCE (PUBLIC) 94
6.1.9 USSE.USSE.STARTSEQUENCE (PUBLIC) 94
6.1.10 USSE.USSE.CALLBACKS (PROTECTED) 94
6.1.11 USSE.USSE.GETDATA (PUBLIC) 94
6.1.12 USSE.USSE.STOPSEQUENCE (PUBLIC) 94
6.1.13 USSE.USSE.QUITREMOTE (PUBLIC) 95
6.1.14 USSE.USSE.STOPSYSTEM (PUBLIC) 95
6.1.15 Inherited methods 95

6.2 The *usse.lte* class (*usse.usse* subclass) 95
6.2.1 USSE.LTE (USSE.USSE) 95
6.2.2 USSE.LTE.INITIALIZE (PROTECTED) 97
6.2.3 USSE.LTE.ADDIMAGING (PUBLIC) 97
6.2.4 USSE.LTE.ADDHIFU (PUBLIC) 98
6.2.5 USSE.LTE.BUILDREMOTE (PUBLIC) 98
6.2.6 USSE.LTE.GETDATA (PUBLIC) 99
6.2.7 USSE.LTE.SETVOLTAGE (PUBLIC) 99
6.2.8 Inherited methods 99

6.3 USSE.CONTROLSEQUENCE (PUBLIC) 99

7 SYSTEM 99

7.1 The *system.hardware* class 99
7.1.1 SYSTEM.HARDWARE 99
7.1.2 Aixplorer 100
7.1.3 Liver Therapy 100
7.1.4 Brain Therapy 101

7.2 The *system.probe* class 101
7.2.1 SYSTEM.PROBE 101

7.2.2 VC-192 101
7.2.3 VL-256 101
7.2.4 Stanford 102

8 Tutorials 102

8.1 A flat elusev class 102

9 Appendixes 102

9.1 Optimization of the number of events for HIFU sequences 102
9.1.1 Preliminary actions 102
9.1.2 Event duration greater than 5 ms 102
9.1.3 Event duration shorter than 2 ms 102

10 List of figures, tables and codes 102

10.1 List of figures 102

10.2 List of tables 103

10.3 List of codes 103

1 Introduction

The legHAL package is a Matlab Interface for RUBI and it has been designed to use the REMOTE module of RUBI in order to define custom ultrasound sequences, to retrieve data and to treat those using custom Matlab scripts or functions. The whole interface is based upon the Matlab concept of classes and it has been designed to be as flexible as possible.

This document is presenting the legHAL package and more particularly the different Matlab classes which have been developed to define, control, load and use ultrasound sequences with the REMOTE module of RUBI. It is not designed as a User's Guide of the legHAL package, i.e. it cannot be used as documentation for external customers. Only excerpts of this guide might be given to customers.

1.1 Principle

The REMOTE module of RUBI has been developed in order to control a RUBI-managed device via an external program. The connection between the external program and RUBI can be realized via an Ethernet cable.

The legHAL package is based upon Matlab classes to define an ultrasound sequence and communicate with the REMOTE module of RUBI. Typically, the ultrasound sequence is sent to the RUBI-managed device as a structure with mandatory and optional fields. The communication between Matlab and the REMOTE module is managed via mex-files which are used within the Matlab classes of the legHAL package. This Matlab interface has several input levels:

- The organization of existing acquisition modes within the ultrasound sequence – *SWE acquisitions interleaved with B-mode acquisitions*.
- The organization of predefined modules within an acquisition mode – *a push module placed before ultrafast acquisition to define a SWE acquisition mode*.
- The organization of events within a module of an acquisition mode – *the emission/reception for a (-10)-radians flat angle before the emission/reception for a 0-radians flat angle and the emission/reception for a 10-radians flat angle*.
- The definition of predefined elementary objects – *the definition of a flat transmit, a focused transmit, an arbitrary waveform or a periodic waveform*.

The several levels are needed to make this interface work properly. However, it should be noted that the several levels can be encrypted if the corresponding files are binarized, i.e. the m-files are converted into p-files using the Matlab *pcode* function.

1.2 Requirements

The Matlab Interface has the same requirements as the requirements for using the REMOTE module with Matlab:

- Matlab version ≥ 2008b.
- Same version for legHAL package and REMOTE module on the RUBI-managed device.
- The right to set the REMOTE level to system on the RUBI-managed device.

Matlab should be installed on a remote computer connected to the RUBI-managed device via an Ethernet connection. In order to have the best data rate, it is recommended to use on the remote computer a 1-Gbit Ethernet card. Moreover, a direct connection between the remote computer and the RUBI-managed device improves the data transfer speed.

Finally, it should be noted that the behavior of the legHAL package does not depend on the RUBI version. The legHAL package is using a REMOTE module which is not using any of the RUBI generic features. Thus, the only limitations between different RUBI versions are only due to hardware issues instead of software implementations, i.e. it can be used to remotely control an Aixplorer, a Liver Therapy Electronics or a Brain Therapy system.

The source of the legHAL package is located on SVN: *trunk/us/legHAL/src/*

1.3 The Matlab concept of classes

This section is presenting an overview of how to implement classes in Matlab. The general principles part explains the advantage of this concept for code maintaining and code review. A second part is giving the main commands to build a Matlab class and presents the implementation of inheritance.

However, this section is not an exhaustive documentation on Matlab classes. It is just presenting the main notions to understand the description of legHAL Matlab classes. If the reader wants to learn more on this subject, it is recommended to use *Product Help* in Matlab (as usually when programming Matlab function or scripts) and more specifically the *Object-Oriented Programming* section in the Matlab User Guide.

1.3.1 Principles of Matlab classes

The legHAL package is based upon the Matlab concept of classes. This concept gives the opportunity to create objects like in C++, i.e. properties and methods are associated to each instance of the object. Thus, it is possible to call in terms of pure syntax the same function for two different objects and the action of this call might be different as the two objects belong to two different classes with potentially two different implementations of the called function.

> *"The scalar product does not have the same definition for vectors and functions although it has the same name for each type of objects. It corresponds to the same syntax but its implementation is different regarding the objects class."*

In addition, the access to the properties or methods of a class is controlled. The control of the properties is realized on two different levels: the set and get attributes of properties. The set attribute corresponds to the possibility to change or not the value of a property. The get attribute corresponds to the possibility to have access or not to the value of a property.

"A rectangle class has several properties such as its length/width or its perimeter/area. All these properties are needed to characterize the rectangle (get-public) but the perimeter/area depends directly on the width/length of the rectangle. The width/length can be changed (set-public), but the perimeter/area should not be changed directly (set-private)."

The control of the methods is realized on a single level, i.e. the method can only be called by a method belonging to the class (private method) or it can be called just by using an instance of the class (public method).

"For a rectangle class, the new values of the length/width change automatically its area and perimeter. This implies that by setting a new value to the length/width a private method is called during the setting process to change the perimeter/area properties."

Moreover, it is possible to create subclasses which are inheriting from a superclass. This concept, which is widely used in C++, is a major concept because the subclass has automatically the same properties and functions as the superclass, even for private properties and functions. The subclass is automatically an instance of the superclass. Thus, there is no need to define a second time common properties or methods.

"Real numbers correspond to a specific type of complex numbers for which the imaginary part is null. Thus, the basic arithmetic operations (sum, difference, multiplication and division) do not need to be defined a second time if the real class is inheriting from the complex class."

In addition, the inheritance concept gives the opportunity to overload the superclass methods, i.e. a superclass method can be partly redefined for a specific subclass to add dedicated actions. Or a subclass can have additional methods or properties compared to those defined for the superclass. Furthermore, a given subclass can even inherit from several superclasses.

"A square class inherits from a rectangle class as it has the same properties (length, width, perimeter, area). However, when setting to a new value length (respectively the width), the private function is overloaded as it is changing the width (respectively the length) before changing the perimeter/area values."

1.3.2 Program structure

In terms of program structures, the legHAL classes are gathered within packages which are easily located as their name is *+PackageName*. A class *ClassName* is always located in a directory *@ClassName* and the class declaration is placed in an m-file (respectively a p-file if the file has been binarized) *ClassName.m* (respectively *ClassName.p*). The *ClassName.m* file contains the declaration of the properties and methods as well as the constructor of the class (see a generic example in Code 1).

```
classdef ClassName

  properties ( SetAccess = ?, GetAccess = ?, Constant = ?)
    Properties...
  end
```

```
  methods ( Access = ?, Abstract = ?)
    Methods...
  end

  methods
    function obj = ClassName(varargin)
      Initial actions...
    end
  end

end
```

Code 1 – Example of a class declaration within the *Classname.m* m-file.

For properties, three distinct fields are used:

- The *SetAccess* field sets the authorization to change the property value, i.e. for *private* or *protected* the value can only be changed using a class method and for *public* the value can be changed directly by using a class instance.
- The *GetAccess* field sets the authorization to know the property value, i.e. for *private* or *protected* the value can only be accessed within a class method and for *public* the value can be accessed by using a class instance.
- The *Constant* field sets the possibility to change the property value, i.e. for *true* the value cannot be changed within a method or by using a class instance and for *false* (default value) it is possible to change the value according to the *SetAccess* field value.

For methods, two distinct fields are used:

- The *Access* field sets the authorization to use a method, i.e. for *private* or *protected* the method can only be called by a class method and for *public* it can be called by using a class instance.
- The *Abstract* field gives the opportunity to associate a method to a superclass and implement it within a subclass, i.e. for *true* the method is mandatorily defined in subclasses and for *false* it is defined in the superclass (but it can be overloaded in subclasses).

For properties as well as methods, the difference between private and protected is the accessibility by the subclasses. If private is used, the property or the method can only be accessed by class instances and not by subclass instances. If protected is used, the property as well as the method can be accessed by the class instances and the subclass instances.

The constructor is a method which is implemented in the *ClassName.m* file and it creates the instance of the class. Several initial actions can be realized within this method, and it is also linking a subclass to its superclass (see the example in Code 2).

```
classdef SuperClass
  Generic declaration...
```

```
  Constructor...
end

classdef SubClass < SuperClass

  Generic declaration...

  methods
    function obj = SubClass(varargin)

      obj = obj@SuperClass(varargin{1:end});

      Initialization actions...

    end
  end

end
```

Code 2 – Example of superclass and subclass declarations.

To create an instance of the class *ClassName* which is part of the package *PackageName*, the code to type within a script, a function or the Matlab command window is:

```
VarName = PackageName.ClassName(varargin);
```

where *VarName* is the name of the instance and *varargin* corresponds to the input parameters.

1.4 The legHAL package architecture

As exposed previously, the legHAL package is based upon the concept of Matlab class. The main advantage of this concept is its flexibility. It gives the opportunity to implement methods which are dedicated to specific objects, e.g. a method to compute the delays of each emitting transducers to generate a flat emission is dedicated to a transmit class and it is not relevant for other classes.

Moreover, the control of the data format is increased by using Matlab classes because the input variables and output variables can be completely defined within the class implementation via specific methods to the class. By using the inheritance concept, it is also possible to ease the code review as the common function are implemented for the superclass and the sole dedicated methods are implemented for specific classes.

Although the inheritance ease the code review and its maintenance, it is more difficult to track bugs especially if there are several levels of superclasses as in the legHAL package, i.e. the bug can occur for a specific subclass in a method of its parent or “grand-parent” class. In addition, the inheritance is slowing down the execution of the several methods because of the numerous methods calls, i.e. to build the structure of a waveform some actions might occur in a general waveform method and others in a more specific waveform method. However, the possibility to have general definition and more

specific ones counterbalance this slower execution, i.e. the legHAL package is designed for research purposes and not for diagnostic purposes. Moreover, the slow-down in execution speed is observed for the sequence definition and not its execution on the RUBI-managed device.

The core of the REMOTE module is the structure defining the ultrasound sequence. Thus, the goal of the legHAL package is to build an acceptable structure, i.e. controls should be realized on the structure in order to load a sequence which will not generate errors when it is executed on the RUBI-managed device. In order to build this structure, parameters as well as objects are needed while others are just used in order to build the structure. The differentiation between parameters and objects, intermediate classes and remote classes is based upon general classes which are the basis of the legHAL package. These classes are implementing all the generic methods which are used by all implemented subclasses.

Based on the common package classes, four packages (*remote*, *elusev*, *acmo*, *usse*) are used to build an ultrasound sequence. The *remote* package contains all the classes to build the mandatory fields of the structure defining the sequence to be loaded via the REMOTE module. The *elusev* package (ELementary UltraSound EVents) contains classes that are gathering the emission definition, reception definition and their layout within events. The *acmo* package (ACquisition MOde) contains classes that are built out of *elusev* objects and is adding *tgc*, *mode* and data transfer. For instance, it can be compared to the B-mode or the Color mode. The *usse* package (UltraSound SEquence) contains all the classes which manage ultrasound sequences and the RUBI-managed device as well as *acmo* instances. These classes give the opportunity to load ultrasound sequences, to change the REMOTE level, …

Fig. 1 – Hierarchy of generic objects to define an ultrasound sequence.

In parallel to the common package and the previously described packages, there is a package – the *system* package – which is used by all packages because it contains two classes defining the *hardware* (the type of RUBI-managed device) and the *probe* which will imply changes for several objects, e.g. the transmit delays for a flat emission.

2 The *common* package

The *common* package corresponds to the basic classes of the legHAL package, i.e. these classes are the superclasses of all the classes of the different packages. There are two types of hierarchies:

- The first hierarchy is differentiating the parameters and the objects.
- The second hierarchy is differentiating classes which have an impact on the REMOTE structure and other classes without a direct action on the creation of the REMOTE structure.

In addition to the basic classes, the *constants* class has been implemented to define all constants that might be needed by the legHAL package. The *legHal* class defines constant values of the legHAL interface, e.g. the version number.

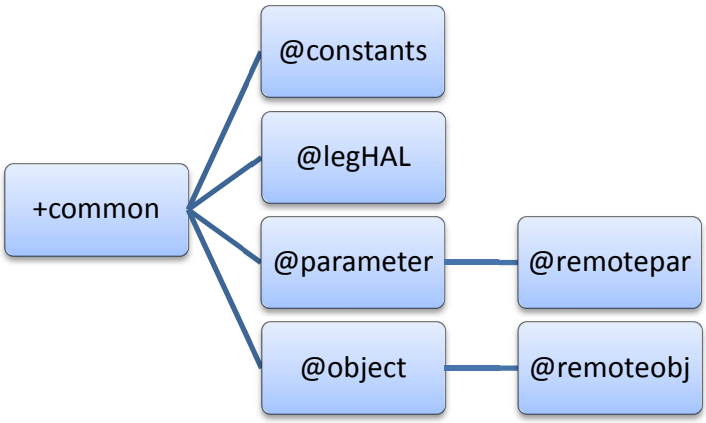

Fig. 2 – Structure and hierarchy of the *common* package.

2.1 The *common.constants* class

The *constants* class is intended to contain all constant variables that are needed to build the REMOTE structure, e.g. the speed of sound. This class is meant to evolve with the legHAL package. As this class contains only constant variables, it does not have any constructor but only constant properties. To use a common.constants variable, the code to add in a script/function or in Matlab command window is:

```
common.constants.VarName
```

where *VarName* is a variable of this class. All variables belonging to this class are listed below.

Variable	Value	Units
SoundSpeed	1540	$m.s^{-1}$

Tab. 2.1 – List of variables of the *constants* class.

2.2 The *common.legHAL* class

The *legHAL* class is similar to the *constants* class as it is intended to contain generic values or methods to be used within the legHAL package. More specifically, these variables and methods are used by the classes and methods of the legHAL package. Thus, the legHAL class only contains constant variables and static methods.

2.2.1 The *common.legHAL* variables

Similarly to the constants class, *legHAL* variables are called by the code:

```
common.legHAL.VarName
```

where *VarName* is a variable of this class. All variables belonging to this class are listed below.

Variable	Value	Units
legHAL	x.x[1]	N/A

Tab. 2.2 – List of variables of the *legHAL* class.

2.2.2 The *common.legHAL.GetException* method

The *GetException* method is intended to issue a Matlab exception with a customized identifier depending on the object class and the method in which an exception was first raised. The syntax of this method is:

```
NewException = ...
    common.legHAL.GetException(OldException, ObjName, MethName);
```

where *NewException* is an *MException* instance containing the generated Matlab exception, *OldException* the original Matlab exception, *ObjName* the full name of the object class (e.g. *remote.rx*) and *MethName* the name of the method where the Matlab exception was firstly raised. It should be noted that the original Matlab exception *OldException* is added as the cause of the generated exception *NewException*.

2.3 The *common.parameter* class

The *parameter* class is defining a class which is extending the definition of class variables. Its goal is to define a variable, its type of values, authorized values and it also gives a description of the variable and of its authorized values. This class is characterized by the variables and the methods listed below.

Variable	Type	SetAccess	GetAccess	Description
Name	char	protected	public	Name of the parameter.
Desc	char	protected	public	General description.
Type	char	protected	public	Type of the value (char, int32, double …).
Value	= Type	protected	protected	Value of the parameter.

[1] The version of the legHAL is incremented regarding the remote version of RUBI.

Variable	Type	SetAccess	GetAccess	Description
AuthValues	= {Type}	protected	public	Authorized values (cell array of values).
AuthRange	{0 or 1}	protected	protected	Type of authorized values (cell array), i.e. single value (0) or range of values (1).
AuthDesc	{char}	protected	public	Description for each authorized values.
Debug	0 or 1	public	public	Debug mode is off (0) or on (1).

Tab. 2.3 – List of the variables of the *parameter* class.

Method	Access	Nargin	Nargout	Description
initialize	protected	0/5/6	1	Build a *parameter* instance.
setValue	public	1	1	Return the value.
getValue	public	0	1	Set the value.
isempty	public	0	1	Check if the *Value* is empty.

Tab. 2.4 – List of the methods of the *parameter* class.

2.3.1 The *common.parameter* constructor (public) and the *initialize* method (protected)

The *parameter* constructor method and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method in order to control the syntax as well as the type of the input arguments and their values.

An empty call of the *parameter* constructor is generating an error with a message containing the authorized syntaxes of the method (with *Obj* a parameter instance):

```
Obj = common.parameter();
Obj = Obj.initialize();
```

The first syntax is creating a *parameter* instance with an empty value and the *Debug* variable set to 0:

```
Obj = common.parameter(Name, Type, Desc, AuthValues, AuthDesc);
Obj = Obj.initialize(Name, Type, Desc, AuthValues, AuthDesc);
```

where *Obj* is a *parameter* instance, *Name* the name of the parameter, *Type* the type of its value, *Desc* its description, *AuthValues* a cell array containing the different authorized values (range of values or single values), *AuthDesc* a cell array containing a description of each authorized values.

The second syntax is creating a *parameter* instance and sets its *Debug* value:

```
Obj = common.parameter(Name, Type, Desc, AuthValues, AuthDesc, Debug);
Obj = Obj.initialize(Name, Type, Desc, AuthValues, AuthDesc, Debug);
```

where *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The *initialize* method controls the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments:

- the type and value of the *Debug* argument if defined,
- the type of the *Name* variable should be a *char* value,
- the type of the *Type* variable should be a *char* value and correspond to an existing class (Matlab class or customized class),
- the type of the *Desc* variable should be a *char* value,
- the *AuthValues* and *AuthDesc* variables should be cell arrays with the same dimension,
- the type of the *AuthValues* cells should be of the *Type* class,
- the type of the *AuthDesc* cells should be *char* values.

2.3.2 The *setValue* method (public)

The setValue method of the *parameter* class has a single syntax:

```
Obj = Obj.setValue(Value);
```

where *Obj* is a parameter instance and *Value* is the new value of the *parameter* instance. The type of the new value is controlled regarding the expected type of the *parameter* instance. If it is not a convenient type of data, the new value is not set and an error is thrown. It should be noted that a *parameter* instance can be set to a vector or a n-dimensions matrix value, but the values cannot belong to distinct authorized values.

2.3.3 The *getValue* method (public)

The *getValue* method of the *parameter* class has a single syntax:

```
Value = Obj.getValue();
```

where *Value* is the value of the *parameter* instance *Obj*. Prior to returning the value of the *parameter* instance, an error can be thrown if no value has been defined.

2.3.4 The *isempty* method (public)

The *isempty* method of the *parameter* class has a single syntax:

```
Success = Obj.isempty();
```

where *Success* is 1 if the value of the *parameter* instance is not defined, or 0 otherwise.

2.4 The *common.remotepar* class (*common.parameter* subclass)

The *remotepar* class is defined as a parameter which is mandatory to build the remote structure to define an ultrasound sequence. It is indeed a restriction of the *parameter* class, i.e. the value of a

remotepar instance can only be set to a numeric value. This class is characterized by the same variables and methods listed in the previous tables (see Tab. 2.3 for the variables and Tab. 2.4 for the methods).

2.4.1 The *common.remotepar* constructor (public) and the *initialize* method (protected)

Similarly to the *parameter* class, the *remotepar* constructor method and the *initialize* method (protected method) have the two same syntaxes:

```
Obj = common.parameter(Name, Type, Desc, AuthValues, AuthDesc);
Obj = Obj.initialize(Name, Type, Desc, AuthValues, AuthDesc);
```

where *Obj* is a *remotepar* instance, *Name* the name of the remote parameter, *Type* the type of its value, *Desc* its description, *AuthValues* a cell array containing the different authorized values (range of values or single values), *AuthDesc* a cell array containing a description of each authorized values. The second syntax is creating a *remotepar* instance and set its debug value:

```
Obj = common.parameter(Name, Type, Desc, AuthValues, AuthDesc, Debug);
Obj = Obj.initialize(Name, Type, Desc, AuthValues, AuthDesc, Debug);
```

where *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The *initialize* method is first calling the *initialize* method of the *parameter* superclass. Then, the *initialize* method is controlling if the type of data is numeric as only numeric values are authorized for the *remotepar* class.

2.4.2 Inherited methods

As a subclass of the *parameter* class, the *remotepar* class inherits several methods:

- the *setValue* method sets the new value of the *remotepar* instance,
- the *getValue* method retrieves the value of the *remotepar* instance,
- the *isempty* method checks if the *Value* variable is empty.

2.5 The *common.object* class

The *object* class is defining a class which is designed to contain several *parameter* (or *remotepar*) instances. Its goal is to define an object which is characterized by several parameters and which has methods to manage them. This class is characterized by the variables and the methods listed below.

Variable	Type	SetAccess	GetAccess	Description
Name	char	protected	public	Name of the parameter.
Desc	char	protected	public	General description.

Variable	Type	SetAccess	GetAccess	Description
Type	char	protected	public	Class of the object.
Pars[2]	{*parameter*}	protected	protected	Container of the associated parameters.
Objs[3]	{*object*}	protected	protected	Container of the associated objects.
Debug	0 or 1	public	public	Debug mode is off (0) or on (1).

Tab. 2.5 – List of the variables of the *object* class.

Method	Access	Nargin	Nargout	Description
initialize	protected	0/5/6	1	Build a *parameter* instance.
addParam	protected	1/2	1	Add a new parameter or object.
isParam	public	N/A[4]	1/2	Check if a parameter (or object) belongs to the *object* instance.
setParam	public	N/A4	1	Set the value of one or several parameters (and/or objects) of the *object* instance.
getParam	public	1	1	Retrieve the value of a parameter (or object) belonging to the *object* instance.
isempty	Public	0	1	Check if the *Value* is empty.

Tab. 2.6 – List of the methods of the *object* class.

2.5.1 The *common.object* constructor (public) and the *initialize* method (protected)

The *object* constructor method and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method in order to control the syntax, the type of the input arguments and their values.

An empty call of the *object* constructor is generating an error with a message containing the authorized syntaxes of the method (with *Obj* an object instance):

```
Obj = common.object();
Obj = Obj.initialize();
```

[2] The *Pars* variable is a cell array. The first column contains the name of the parameter, the second the class of the parameter and the third is either an instance of the expected class (or a subclass) or a cell array containing instances of the expected class (or a subclass).

[3] The *Objs* variable is a cell array. The first column contains the name of the parameter, the second the class of the parameter and the third is either an instance of the expected class (or a subclass) or a cell array containing instances of the expected class (or a subclass).

[4] The number of input arguments is infinite, i.e. the number of input arguments is greater or equal to 1 and smaller or equal to the number of parameters and objects.

The first syntax is creating an *object* instance with its *Debug* variable set to 0:

```
Obj = common.object(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is an *object* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *object* instance and sets its *Debug* variable:

```
Obj = common.object(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is an *object* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *object* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = common.object(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *object* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method controls the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments:

- the type and value of the *Debug* argument if defined,
- the type of the *Name* variable should be a *char* value,
- the type of the *Type* variable is set by calling the *class* method (Matlab method),
- the type of the *Desc* variable should be a *char* value.

After the execution of the *initialize* method, the *object* constructor is calling the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax. The constructor method finally sorts the *Pars* variable (and the *Objs* variable) according to the parameter names (and the object names). This last action is implemented in order to optimize the legHAL package execution.

2.5.2 The *addParam* method (protected)

The *addParam* method adds a *parameter* instance (or an *object* instance) to the calling *object* instance. As this method is protected, it should be called within a method belonging to the *object* instance. For instance, this method should be called in the overriding *initialize* method of an *object* subclass.

The first syntax is adding a *parameter* instance (or an *object* instance) to the *object* instance:

```
Obj = Obj.addParam(Param);
```

where *Obj* is the *object* instance and *Param* a *parameter* instance (or an *object* instance).

The *addParam* method first determines the class of *Param* in order to put the instance in the *Pars* variable (respectively the *Objs* variable) if it is a *parameter* instance (respectively an *object* instance). According to the *Param.Name* value, the method then controls if no parameter of the object instance has the same name. The new parameter is finally added to the *Pars* variable or the *Objs* variable.

The second syntax adds a container of parameters:

```
Obj = Obj.addParam(ParName, ParType);
```

where *ParName* is the name of the container and *ParType* the type of authorized values. The container of parameters is a cell array which can only contain objects of the *ParType* class and its subclasses.

The *addParam* method first determines if the class *ParType* is *common.parameter* or a subclass (respectively *common.object* or a subclass) to create a new cell array value in the *Pars* variable (respectively the *Objs* variable). According to *ParName*, the method then controls if no parameter of the object instance has the same name. The new parameter is finally added to the *Pars* variable or the *Objs* variable, i.e. the first column value is set to *ParName*, the second to *ParType* and the third to an empty cell array.

2.5.3 The *isParam* method (public)

The *isParam* method checks if a list of parameters belongs to the *object* instance. The first syntax is:

```
Success = Obj.isParam(ParName, ...);
```

where *Success* is 1 if all parameter names *ParName* (*char* values) belong to the *object* instance *Obj*, and 0 otherwise. The second syntax returns the variable name and the index of each parameter *ParName*:

```
[VarName Idx] = Obj.isParam(ParName, ...);
```

where *VarName* is a cell array containing the name of the variable (Pars or Objs) and *Idx* a cell array containing the index in the variable for each parameter name.

After controlling the syntax of the *isParam* call, the method determines the index of each parameter name within the *Pars* and *Objs* variables. Furthermore, it creates a logical variable containing a 1 if the parameter name identifies an actual parameter of the *object* instance.

2.5.4 The *setParam* method (public)

The *setParam* method sets a parameter to a value which is specified in the method call:

```
Obj = Obj.setParam(ParName, ParValue, ...);
```

where *Obj* is the object instance, *ParName* the name of a parameter and *ParValue* the new value of the *ParName* parameter.

The *setParam* method first calls the *isParam* method to locate the parameters to be changed. It then changes the value of the *parameter* instance or the *object* instance regarding the class of the parameter. If the parameter is a cell array, it simply adds the new parameter.

If the parameter is a cell array and the new value is cell array of instances of authorized class, all cells are added to the parameter.

2.5.5 The *getParam* method (public)

The *getParam* method returns the value of a parameter of the *object* instance:

```
Par = Obj.getParam(ParName);
```

where *Par* is the value of a *parameter* instance or an *object* instance, *Obj* the object instance and *ParName* the name of the parameter to be returned. As previously, the *getParam* method first controls the existence of the parameter and then returns its value.

2.5.6 The *isempty* method (public)

The *isempty* method of the *object* class has a single syntax:

```
Success = Obj.isempty();
```

where *Success* is 1 if at least one parameter value is empty or one object is empty (i.e. call of the *isempty* method associated to the objects of the *object* instance).

2.6 The *common.remoteobj* class (*common.object* subclass)

The *remoteobj* class is defined as an object which has mandatory fields to build a remote structure to define an ultrasound sequence. This class is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables and Tab. 2.6 for the methods). In addition, this class has a dedicated method to build the remote structure: the *buildRemote* method.

Method	Access	Nargin	Nargout	Description
buildRemote	public	0	1	Returns the remote structure with its mandatory fields (*remotepar* instances).

Tab. 2.7 – List of the methods of the *remoteobj* class.

2.6.1 The *common.remoteobj* constructor (public)

Similarly to the *object* class, an empty call of the *remoteobj* constructor is generating an error with a message containing the authorized syntaxes of the method (with *Obj* an object instance):

```
Obj = common.remoteobj();
```

The first syntax is creating a *remoteobj* instance with its *Debug* variable set to 0:

```
Obj = common.remoteobj(Name, Desc);
```

where *Obj* is a *remoteobj* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a remote*obj* instance and sets its *Debug* variable:

```
Obj = common.remoteobj(Name, Desc, Debug);
```

where *Obj* is a *remoteobj* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *remoteobj* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = common.remoteobj(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *remoteobj* class. It should be noted that the *Debug* value is optional and it is set to 0 by default. No additional controls are added to those described in 2.5.1.

2.6.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *remoteobj* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *remoteobj* instance.

The buildRemote first controls the syntax of the method call and the version of the legHAL package. It then exports all *remotepar* parameters by controlling the class of each parameters of the *remoteobj* instance. Thus, it is important to use parameter labels which are compatible with the remote module of RUBI.

2.6.3 Inherited methods

As a subclass of the *object* class, the *remoteobj* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the remote*obj* instance,
- the *isParam* method checks if a list of parameters belongs to the *remoteobj* instance,
- the *setParam* method sets a parameter of the *remoteobj* instance to a new value,
- The *getParam* method returns the value of a parameter of the *remoteobj* instance,

- the *isempty* method checks if all parameters of the *remoteobj* instance (*parameter* instances and *object* instances) are empty.

3 REMOTE

The *remote* package contains all the classes defining the mandatory fields of the structure defining an ultrasound sequence for RUBI-managed device. All classes of the *remote* package are subclasses of the *common.remoteobj* class. Thus, they are inheriting from the *remoteobj* and *object* classes. However, they are overriding the *initialize* method of the *object* class and *buildRemote* method of the *remoteobj* class.

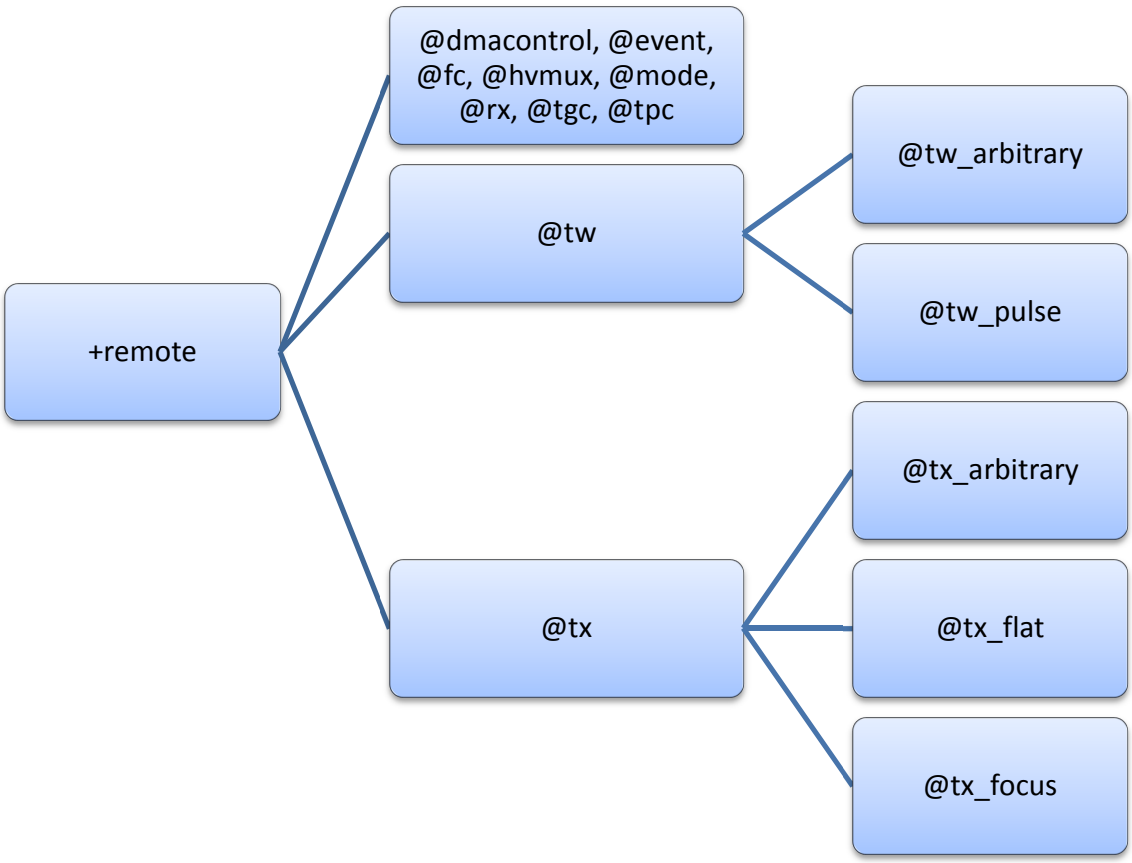

Fig. 3 – Structure and hierarchy of the *remote* package.

3.1 The *remote.dmacontrol* class (*common.remoteobj* subclass)

The *dmacontrol* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In

addition, this class has a dedicated parameter: the *localBuffer* parameter. It also overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
localBuffer	int32	0	[0 Inf]	Index of the local buffer to transfer.

Tab. 3.1 – List of the parameters of the *dmacontrol* class.

3.1.1 The *remote.dmaControl* constructor (public) and the *initialize* method (protected)

The *dmaControl* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *dmaControl* constructor is generating a generic *dmaControl* instance with its *Name* variable set to "DMACONTROL" and its *Desc* variable to "default control of direct memory access":

```
Obj = remote.dmacontrol();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *dmaControl* instance with its *Debug* variable set to 0:

```
Obj = remote.dmacontrol(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *dmaControl* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *dmaControl* instance and sets its *Debug* variable:

```
Obj = remote.dmacontrol(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *dmaControl* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *dmaControl* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.dmacontrol(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *dmaControl* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The initialize method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameter (see Tab. 3.1), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.1.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *dmaControl* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *dmaControl* instance.

As the *dmaControl* class does not need any additional actions, the *buildRemote* method just calls the *buildRemote* method of the *remoteobj* class to export the *localBuffer* parameter.

3.1.3 Inherited methods

As a subclass of the *remoteobj* class, the *dmaControl* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *dmaControl* instance,
- the *isParam* method checks if a list of parameters belongs to the *dmaControl* instance,
- the *setParam* method sets a parameter of the *dmaControl* instance to a new value,
- The *getParam* method returns the value of a parameter of the *dmaControl* instance,
- the *isempty* method checks if all parameters of the *dmaControl* instance (*parameter* instances and *object* instances) are empty.

3.2 The *remote.event* class (*common.remoteobj* subclass)

The *event* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
txId	int32	0	0 [1 Inf]	No associated transmit. Index of the associated transmit.
rxId	int32	0	0 [1 Inf]	No associated receive. Index of the associated receive.

Parameter	Type	Default	Auth Values	Description
hvmuxId	int32	0	0 [1 Inf]	No associated hvmux. Index of the associated hvmux.
tpcTxe	int32	0	0 [1 Inf]	No extended power. Extended power (pushes reservoir).
localBuffer	int32	0	[0 Inf]	Index of the associated local buffer.
dmaControlId	int32	0	0 [1 Inf]	No DMA. Index of the associated DMA.
noop	int32	MinNoop[5]	[MinNoop 5e5]	Duration of event dead time [µs].
softIRQ	int32	-1	-1 [0 Inf]	None. Soft IRQ label.
return	int32	0	0 1	No changes in the events execution. Return to the calling event.
goSubEventId	int32	0	0 [1 Inf]	No changes in the events execution. Execute event id.
waitExtTrig	int32	0	0 1	No trigger in. Event execution needs a trigger in.
genExtTrig	single	0	0 [1e-3 720.8]	No trigger out. Duration of the trigger out [µs].
genExtTrigDelay	single	0	[0 1000]	Duration between the trigger out generation and the transmit start [µs].
duration	int32		0 [1 1e7]	Empty event. Event duration [µs].
numSamples	int32		[0 4096]	Number of acquired samples.
skipSamples	int32		[0 4096]	Number of skipped samples.

Tab. 3.2 – List of the parameters of the *event* class.

3.2.1 The *remote.event* constructor (public) and the *initialize* method (protected)

The *event* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

[5] The *MinNoop* value is the minimum duration of the event no-operation (see the *system.hardware* class - 7.1).

An empty call of the *event* constructor is generating a generic *event* instance with its *Name* variable set to “EVENT” and its *Desc* variable to “default event”:

```
Obj = remote.event();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating an *event* instance with its *Debug* variable set to 0:

```
Obj = remote.event(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is an *event* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *event* instance and sets its *Debug* variable:

```
Obj = remote.event(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is an *event* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *event* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.event(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *event* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.2), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.2.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *event* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *event* instance.

The *buildRemote* method first controls the *genExtTrigDelay* parameter and set it to 0 if no trigger out is generated, and calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. The *genExtTrig* and *genExtTrigDelay* fields of the remote structure are then rescaled as they are expected to be durations in ns. Several fields are added because they are processed in other parts of the legHAL package:

- tgcId = 1 – computed in the *buildRemote* method of the *usse.usse* class.
- modeId = 1 – computed in the *buildRemote* method of the *usse.usse* class.
- pauseForEndDma = 0 – always set to 1 to avoid buffer crash.
- jumpToEventId = 0 – only used to loop sequence execution[6].
- incrementLocalBufAddr = 1 – avoid data acquisition at the same location in the buffer.

Although the *duration* parameter can be set from 1 µs to 10^7 µs, it can only be set to an integer between 0 and 2047. To use event lasting more than 2.05 ms, the field *periodExtend* gives the opportunity to change the duration multiplier. The duration multiplier is 1 µs if the *periodExtend* field is set to 0, and it is 5 ms if the *periodExtend* field is set to 1. Thus, the *duration* parameter is set to a 5-ms multiplier if it is greater than 2047 µs.

It is approximatively the same for the no-operation duration, i.e. it cannot be set to a value greater than 1023 µs although the *noop* parameter can be set to 5e5 µs. The *buildRemote* method is indeed evaluating the *noopMultiplier* field to define a *noop* field smaller than 1024 µs and keep the real no-operation duration.

3.2.3 Inherited methods

As a subclass of the *remoteobj* class, the *event* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *event* instance,
- the *isParam* method checks if a list of parameters belongs to the *event* instance,
- the *setParam* method sets a parameter of the *event* instance to a new value,
- The *getParam* method returns the value of a parameter of the *event* instance,
- the *isempty* method checks if all parameters of the *event* instance (*parameter* instances and *object* instances) are empty.

3.3 The *remote.fc* class (*common.remoteobj* subclass)

The *fc* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has a dedicated parameter (the *Bandwidth* parameter), and it overrides the protected *initialize* method and the public *buildRemote* method. Linked to the Bandwidth parameter, there is also an additional variable DABFilterCoeff which is private for its SetAccess and GetAccess properties.

[6] The *jumpToEventId* field can be used to create sub-sequences to be executed in a defined order.

Parameter	Type	Default	Auth Values	Description
Bandwidth	Int32	100	-1 [0 100]	FIR BP coeffs for impulse. Pre-calculated FIR BP coeffs.

Tab. 3.3 – List of the parameters of the *fc* class.

3.3.1 The *remote.fc* constructor (public) and the *initialize* method (protected)

The *fc* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *fc* constructor is generating a generic *event* instance with its *Name* variable set to "FC" and its *Desc* variable to "default digital filter coefficients":

```
Obj = remote.fc();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating an *fc* instance with its *Debug* variable set to 0:

```
Obj = remote.fc(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is an *fc* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *fc* instance and sets its *Debug* variable:

```
Obj = remote.fc(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *fc* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *fc* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.fc(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *fc* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameter (see Tab. 3.3), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.3.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *fc* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *fc* instance.

The *buildRemote* method first retrieves the value of the *Bandwidth* parameter. Regarding the value of the FIR bandwidth, the six coefficients are set to the *data* field of the remote structure:

- [0 0 0 0 0 1] for Bandwidth = -1,
- The coefficients of the *DABFilterCoeff* variable for the closest value to the *Bandwidth* value.

These coefficients are reproduced for all receiving channels, i.e. the value of the constant value system.hardware.NbRxChan (see 7.1).

3.3.3 Inherited methods

As a subclass of the *remoteobj* class, the *fc* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *fc* instance,
- the *isParam* method checks if a list of parameters belongs to the *fc* instance,
- the *setParam* method sets a parameter of the *fc* instance to a new value,
- The *getParam* method returns the value of a parameter of the *fc* instance,
- the *isempty* method checks if all parameters of the *fc* instance (*parameter* instances and *object* instances) are empty.

3.4 The *remote.hvmux* class (*common.remoteobj* subclass)

The *hvmux* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has a dedicated parameter (the *blocks* parameter), and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
blocks	int32		[0 Inf]	Position of the HV muxes.

Tab. 3.4 – List of the parameters of the *hvmux* class.

3.4.1 The *remote.hvmux* constructor (public) and the *initialize* method (protected)

The *hvmux* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *hvmux* constructor is generating a generic *event* instance with its *Name* variable set to “HVMUX” and its *Desc* variable to “default mux control”:

```
Obj = remote.hvmux();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating an *hvmux* instance with its *Debug* variable set to 0:

```
Obj = remote.hvmux(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is an *hvmux* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *hvmux* instance and sets its *Debug* variable:

```
Obj = remote.hvmux(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *hvmux* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *hvmux* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.hvmux(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *hvmux* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameter (see Tab. 3.4), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.4.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *hvmux* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *hvmux* instance.

As the *hvmux* class does not need any additional actions, the *buildRemote* method just calls the *buildRemote* method of the *remoteobj* class to export the *blocks* parameter.

3.4.3 Inherited methods

As a subclass of the *remoteobj* class, the *hvmux* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *hvmux* instance,
- the *isParam* method checks if a list of parameters belongs to the *hvmux* instance,
- the *setParam* method sets a parameter of the *hvmux* instance to a new value,
- The *getParam* method returns the value of a parameter of the *hvmux* instance,
- the *isempty* method checks if all parameters of the *hvmux* instance (*parameter* instances and *object* instances) are empty.

3.5 The *remote.mode* class (*common.remoteobj* subclass)

The *mode* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
NbHostBuffer	int32		[1 10]	Number of host buffers.
ModeRX	int32		[0 Inf]	Describe the mode (numSamples, rxId).

Tab. 3.5 – List of the parameters of the *mode* class.

3.5.1 The *remote.mode* constructor (public) and the *initialize* method (protected)

The *mode* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *mode* constructor is generating a generic *mode* instance with its *Name* variable set to “MODE” and its *Desc* variable to “default mode”:

```
Obj = remote.mode();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *mode* instance with its *Debug* variable set to 0:

```
Obj = remote.mode(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *mode* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *mode* instance and sets its *Debug* variable:

```
Obj = remote.mode(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *mode* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *mode* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.mode(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *mode* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.5), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.5.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *mode* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *mode* instance.

The *buildRemote* method first calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. As the ModeRX parameter is a remotepar instance, it has been exported and the total number of acquired samples can be evaluated. It corresponds indeed to number of acquired samples per receiving channel.

The minimum number of blocks is then estimated out of the number of acquired samples per channel using the same method as in RUBI. It is then possible to create the two mandatory fields of the remote structure to define the mode:

- The *channelSize* field to the memory size in bits per channel,
- The hostBufferSize field corresponds to the memory size for all receiving channels.

Finally, the index of the host buffer is set regarding the value of the *NbHostBuffer* parameter.

3.5.3 Inherited methods

As a subclass of the *remoteobj* class, the *mode* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *mode* instance,
- the *isParam* method checks if a list of parameters belongs to the *mode* instance,
- the *setParam* method sets a parameter of the *mode* instance to a new value,
- The *getParam* method returns the value of a parameter of the *mode* instance,
- the *isempty* method checks if all parameters of the *mode* instance (*parameter* instances and *object* instances) are empty.

3.6 The *remote.rx* class (*common.remoteobj* subclass)

The *rx* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
fcId	int32		[1 Inf]	Index of the associated fc.
clipMode	int32	0	0	No clipping.
			1	Maximum clip mode.
			2	Medium clip mode.
			3	Lowest clip mode.
HifuRx	int32	0	0	Contiguous receiving channels.
			1	Distinct receiving channels.
RxElemts	int32		0	No receiving elements.
			[1 NbElemts[7]]	Index of the receiving elements.
RxFreq	single		[1 60]	Sampling frequency [MHz].

[7] The *NbElemts* value is the number of probe elements (see the *system.probe* class - 7.2).

Parameter	Type	Default	Auth Values	Description
QFilter	int32		1	200 % sampling mode.
			2	100 % sampling mode.
			3	50 % sampling mode.
harmonicFilter	int32	1	1	No filter.
			2	-15dB @ f<2MHz, min @ f=3MHz, to -2.5dB @ f∈ [3MHz;5MHz].
			0	-15dB @ f<2MHz, min @ f=3.75MHz, to 0dB @ f∈ [3MHz;10MHz].
vgaInputFilter	int32	0	0	No notch.
			4	Notch @ f = 2.25 MHz.
			2	Notch @ f = 3.75 MHz.
			1	Notch @ f = 5.5 MHz.
vgaLowGain	int32	0	0	No attenuation.
			1	-20 dB after 10 µs.

Tab. 3.6 – List of the parameters of the *rx* class.

3.6.1 The *remote.rx* constructor (public) and the *initialize* method (protected)

The *rx* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *rx* constructor is generating a generic *rx* instance with its *Name* variable set to "RX" and its *Desc* variable to "default receive":

```
Obj = remote.rx();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating an *rx* instance with its *Debug* variable set to 0:

```
Obj = remote.rx(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is an *rx* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *rx* instance and sets its *Debug* variable:

```
Obj = remote.rx(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is an *rx* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *rx* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.rx(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *rx* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.6), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.6.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *rx* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *rx* instance.

The *buildRemote* method first calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. The buildRemote method then controls the sampling frequency in order to estimate the *ADRate* and *ADFilter* values. Indeed, the sampling frequency is:

$$\boldsymbol{RxFreq = \frac{ClockFreq}{ADRate \cdot ADFilter}} \qquad \textbf{Eq. 1}$$

where *ClockFreq* is a system-specific value, *ADRate* and *ADFilter* values belongs to discrete values which are specific to the system hardware[8]. Thus, the closest authorized sampling frequency is estimated to create the *ADRate* and *ADFilter* fields of the remote structure.

The receiving elements are then controlled:

- the receiving channels must belong to the probe elements7,
- the number of receiving channels cannot exceed the number of receiving channels *NbRxChan*[9].

[8] *ClockFreq*, *ADRate* and *ADFilter* are system specific variables (see the system.hardware class - 7.1).
[9] The number of receiving channels *NbRxChan* is a system-specific value (see the system.hardware class - 7.1).

Moreover, the 128 receiving channels are set if the HifuRx is set to 0, i.e. the receiving channels should correspond to contiguous probe elements in that case. If there are less than *NbRxChan* receiving channels, *NbRxChan* receiving channels are selected and these receiving channels are centered at the desired receiving channels center. However, these receiving channels are modified if they correspond to non existing probe elements:

- the first *NbRxChan* receiving channels9 are used if negative index channels are selected,
- the last *NbRxChan* receiving channels9 of the probe are used if channels index are greater than the number of elements *NbElemts*7.

However, the remote structure is not expecting the indexes of the receiving channels. Indeed, the *buildRemote* method builds the *lvMuxBlock* field which is defining the position of the receiving muxes. For each receiving channel, the value is set to 0 if the receiving element belongs to the first *NbRxChan*9 probe elements, and otherwise to 1. These values are then considered as an 8-bit binary number and converted into a decimal number. For instance if NbRxChan = 128, the *lvMuxBlock* field thus contains 16 decimal numbers $rxChans_k$ with $0 \leq RxChans_k \leq 255$.

3.6.3 Inherited methods

As a subclass of the *remoteobj* class, the *rx* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *rx* instance,
- the *isParam* method checks if a list of parameters belongs to the *rx* instance,
- the *setParam* method sets a parameter of the *rx* instance to a new value,
- The *getParam* method returns the value of a parameter of the *rx* instance,
- the *isempty* method checks if all parameters of the *rx* instance (*parameter* instances and *object* instances) are empty.

3.7 The *remote.tgc* class (*common.remoteobj* subclass)

The *tgc* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
Duration	single	0	[0 4000]	Duration of the TGC [µs].
ControlPts	single	900	[0 960]	Values of the TGC control points.
FastTGC	int32	0	0	Classical TGC.
			1	Fast TGC mode.

Tab. 3.7 – List of the parameters of the *tgc* class.

3.7.1 The *remote.tgc* constructor (public) and the *initialize* method (protected)

The *tgc* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tgc* constructor is generating a generic *tgc* instance with its *Name* variable set to "TGC" and its *Desc* variable to "default time gain control":

```
Obj = remote.tgc();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tgc* instance with its *Debug* variable set to 0:

```
Obj = remote.tgc(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tgc* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tgc* instance and sets its *Debug* variable:

```
Obj = remote.tgc(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tgc* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tgc* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tgc(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tgc* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.7), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.7.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *tgc* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *tgc* instance.

The *buildRemote* method first calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. This method then controls the *ControlPts* parameter:

- the ControlPts parameter must be a vector, or at least a single value,
- if the ControlPts parameter is a single value, the value is changed into a 1x8 vector.

The *modeId* field of the remote structure is set to 1 as its value is only set when calling the *buildRemote* method of the *acmo* class (see 5.1.2). And the *wf* field of the remote structure is then built out of the *Duration* and *ControlPts* parameters (see Fig. 4). The TGC waveform corresponds to 512 values which are applied every 0.8 µs. The waveform corresponds to the interpolation of the *ControlPts* values regularly placed during the *Duration* of the TGC. All waveform values are set to the last *ControlPts* value during the remaining duration (409.6 µs - *Duration*).

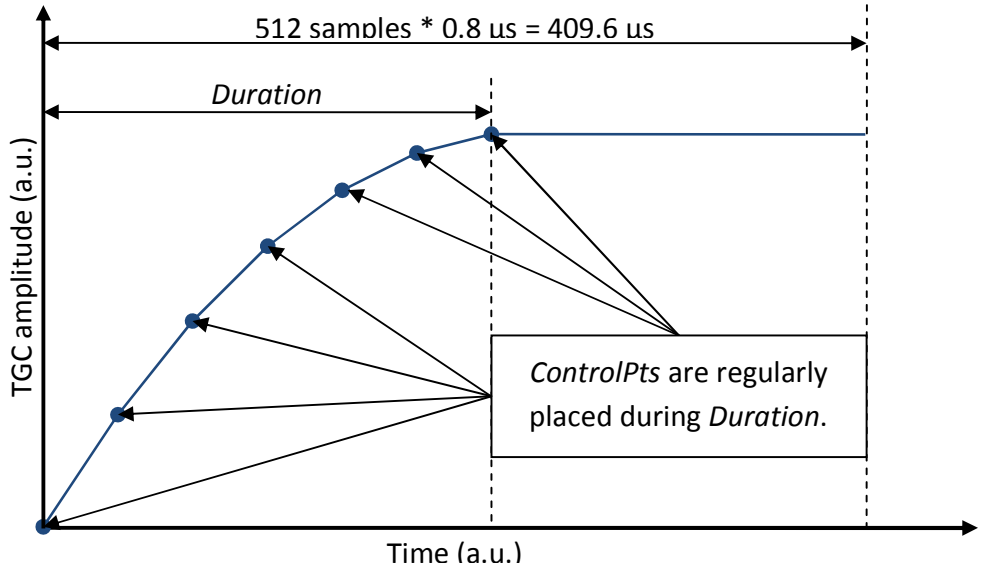

Fig. 4 – TGC curve built out of the *Duration* and *ControlPts* parameters.

3.7.3 Inherited methods

As a subclass of the *remoteobj* class, the *tgc* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tgc* instance,
- the *isParam* method checks if a list of parameters belongs to the *remoteobj* instance,

- the *setParam* method sets a parameter of the *tgc* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tgc* instance,
- the *isempty* method checks if all parameters of the *tgc* instance (*parameter* instances and *object* instances) are empty.

3.8 The *remote.tpc* class (*common.remoteobj* subclass)

The *tpc* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
ImgVoltage	single	10	[10 80]	Maximum imaging voltage [V].
PushVoltage	single	10	[10 80]	Maximum push voltage [V].
ImgCurrent	single	0.1	[0 2]	Maximum imaging current [A].
PushVoltage	single	0.1	[0 20]	Maximum push current [A].

Tab. 3.8 – List of the parameters of the *tpc* class.

3.8.1 The *remote.tpc* constructor (public) and the *initialize* method (protected)

The *tpc* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tpc* constructor is generating a generic *tpc* instance with its *Name* variable set to "TPC" and its *Desc* variable to "default power control":

```
Obj = remote.tpc();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tpc* instance with its *Debug* variable set to 0:

```
Obj = remote.tpc(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tpc* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tpc* instance and sets its *Debug* variable:

```
Obj = remote.tpc(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tpc* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tpc* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tpc(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tpc* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.8), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.8.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *tpc* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *tpc* instance.

The *buildRemote* method first calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. This method then creates the *mode* and *txeRatio* fields of the remote structure which are set to 1 which are the only authorized values for both fields.

3.8.3 Inherited methods

As a subclass of the *remoteobj* class, the *tpc* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tpc* instance,
- the *isParam* method checks if a list of parameters belongs to the *tpc* instance,
- the *setParam* method sets a parameter of the *tpc* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tpc* instance,
- the *isempty* method checks if all parameters of the *tpc* instance (*parameter* instances and *object* instances) are empty.

3.9 The *remote.tw* class (*common.remoteobj* subclass)

The *tw* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
repeat	int32	0	0 [1 1023]	No repetition. Number of repetition.
repeat256	int32	0	0 1	No 256-repetition. 256-repetition.
ApodFct	char	"none"	{ApodFct}[10]	Apodisation function.
TxElemts	int32		[1 NbElemts7]	Index of the transmit elements.
DutyCycle	single		[0 1]	Maximum duty cycle.

Tab. 3.9 – List of the parameters of the *tw* class.

3.9.1 The *remote.tw* constructor (public) and the *initialize* method (protected)

The *tw* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tw* constructor is generating a generic *tw* instance with its *Name* variable set to "TW" and its *Desc* variable to "default waveform":

```
Obj = remote.tw();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tw* instance with its *Debug* variable set to 0:

```
Obj = remote.tw(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tw* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tw* instance and sets its *Debug* variable:

```
Obj = remote.tw(Name, Desc, Debug);
```

[10] The available apodisation functions are: "none", "bartlett", "blackman", "connes", "cosine", "gaussian", "hamming", "hanning", "welch".

```
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tw* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tw* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tw(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tw* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.9), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.9.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *tw* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *tw* instance.

The *buildRemote* method first calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. The method then controls the dimension of the *repeat* and *repeat256* parameters of the *tw* instance:

- the dimension of both parameters should be equal to 1,
- or to the number of the number of transmit channels[11].

Additional mandatory fields are added to the structure corresponding to the *tw* instance, i.e. the *data*, *extendBL* and *legacy* fields are set to 0 (default value). The waveform is then generated by calling the *setWaveform*[12] and *setApodization* methods. A control of the residual energy is implemented to avoid an accumulation of charges in the pulsers. Finally, the *nbWfPoints* and *maxWfPoints* fields are set out of the waveform by evaluating the number of points of the waveform.

[11] The number of transmit channels is the *NbTxChan* variable of the *system.hardware* class.
[12] The *setWaveform* method is not defined for the *tw* class but it is defined for each subclasses.

3.9.3 The *setApodization* method (protected)

The *setApodization* method returns a waveform *WfApod* after applying an apodization window regarding the transmit channels position and converting real values to {-1; 0; 1}-values:

```
WfApod = Obj.setApodization(Wf);
```

where *Obj* is the *tw* instance and *Wf* the initial waveform. To build the apodization window, the positions of the first and last transmitting elements are determined by considering the waveform which is defined for all transmitting elements. Regarding the *ApodFct*[10] parameter of the *tw* instance, the apodization window is estimated for the emitting elements. The value of the duty cycle is then multiplied to the values of the apodization function for each emitting elements.

As the pulser can only emit three values {-1; 0; 1}, it is not possible to just build a waveform out of its desired values times the apodization function. Thus, a last step is conducted to determine the number of {-1; 0; 1} values for each part of the waveform. In order to keep the shape of the waveform, the new waveform is processed by:

- locating the truly negative and positive waveform arches,
- estimating the energy of each positive and negative waveform arches,
- determining the number of 1s (or -1s) for positive (or negative) waveform arches,
- replacing the desired waveform by zeros and centered non-zero values (1 or -1).

The new waveform is then returned by the *setApodization* method to the calling method (normally the *buildRemote* method of the *tw* class is the only calling method of the *setApodization* method).

3.9.4 Inherited methods

As a subclass of the *remoteobj* class, the *tw* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tw* instance,
- the *isParam* method checks if a list of parameters belongs to the *tw* instance,
- the *setParam* method sets a parameter of the *tw* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tw* instance,
- the *isempty* method checks if all parameters of the *tw* instance (*parameter* instances and *object* instances) are empty.

3.10 The *remote.tw_arbitrary* class (*remote.tw* subclass)

The *tw_arbitrary* class is a *tw* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 3.9 for the parameters, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, it overrides the protected *initialize* method and it implements the *setWaveform* method.

Parameter	Type	Default	Auth Values	Description

Parameter	Type	Default	Auth Values	Description
Waveform	single	0	[-1 1]	Waveform defined for each channel, or to be repeated for each channel.
RepeatCh	int32	0	0 1	Waveform defined for each channel. Waveform to be repeated for each channel.

Tab. 3.10 – List of the parameters of the *tw_arbitrary* class.

3.10.1 The *remote.tw_arbitrary* constructor (public) and the *initialize* method (protected)

The *tw_arbitrary* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tw_arbitrary* constructor is generating a generic *tw_arbitrary* instance with its *Name* variable set to "TW_ARBITRARY" and its *Desc* variable to "default arbitrary waveform":

```
Obj = remote.tw_arbitrary();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tw_arbitrary* instance with its *Debug* variable set to 0:

```
Obj = remote.tw_arbitrary(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tw_arbitrary* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tw_arbitrary* instance and sets its *Debug* variable:

```
Obj = remote.tw_arbitrary(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tw_arbitrary* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tw_arbitrary* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tw_arbitrary(Name, Desc, ParsName, ParsValue, ...,
    [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tw_arbitrary* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *tw* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, adds several controls on the input arguments as described in section 2.5.1 and the *tw* parameters (see Tab. 3.9).

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.10), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.10.2 The setWaveform method (protected)

The *setWaveform* method returns an arbitrary waveform *Wf* which is defined for all emitting channels (*TxElemts* parameter) considering the *RepeatCh* parameter:

```
Wf = Obj.setWaveform();
```

where *Obj* is the *tw_arbitrary* instance. As described previously (section 3.9.2), the *buildRemote* method of the *tw* class calls the *setWaveform* method before calling the *setApodization* method.

If the *RepeatCh* parameter is set to 0, the waveform is supposedly defined for all emitting channels. Thus, the method simply controls that the waveform is defined for each emitting parameters, i.e. the first dimension of the *Waveform* parameter should be equal to the length of the *TxElemts* parameter.

If the RepeatCh parameter is set to 1, the waveform needs to be replicated for all emitting channels. Indeed, the *Waveform* parameter is replicated for all emitting channels to build a *Wf* variable which first dimension is equal to the number of emitting channels.

3.10.3 Inherited methods

As a subclass of the *tw* class, the *tw_arbitrary* class inherits several methods:

- the *buidRemote* method builds the mandatory fields of the remote structure for *tw_arbitrary* instances,
- the setApodization method returns a waveform after applying an apodization window,
- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tw_arbitrary* instance,
- the *isParam* method checks if a list of parameters belongs to the *tw_arbitrary* instance,
- the *setParam* method sets a parameter of the *tw_arbitrary* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tw_arbitrary* instance,
- the *isempty* method checks if all parameters of the *tw_arbitrary* instance (*parameter* instances and *object* instances) are empty.

3.11 The *remote.tw_pulse* class (*remote.tw* subclass)

The *tw_pulse* class is a *tw* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 3.9 for the parameters, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, it overrides the protected *initialize* method and it implements the *setWaveform* method..

Parameter	Type	Default	Auth Values	Description
TwFreq	single		[0 20]	Waveform frequency [MHz].
NbHcycle	int32		[0 60]	Number of half cycles.
Polarity	int32	1	-1	Negative 1st arch.
			1	Positive 1st arch.
TxClock180MHz	int32	1	0	90-MHz waveform sampling.
			1	180-MHz waveform sampling.

Tab. 3.11 – List of the parameters of the *tw_pulse* class.

3.11.1 The *remote.tw_pulse* constructor (public) and the *initialize* method (protected)

The *tw_pulse* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tw_pulse* constructor is generating a generic *tw_pulse* instance with its *Name* variable set to "TW_PULSE" and its *Desc* variable to "default periodic waveform":

```
Obj = remote.tw_pulse();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tw_pulse* instance with its *Debug* variable set to 0:

```
Obj = remote.tw_pulse(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tw_pulse* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tw_pulse* instance and sets its *Debug* variable:

```
Obj = remote.tw_pulse(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tw_pulse* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tw_pulse* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tw_pulse(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tw_pulse* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *tw* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, adds several controls on the input arguments as described in section 2.5.1 and the *tw* parameters (see Tab. 3.9).

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.11), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.11.2 The setWaveform method (protected)

The *setWaveform* method returns a periodic waveform *Wf* which is defined for all emitting channels (*TxElemts* parameter):

```
Wf = Obj.setWaveform();
```

where *Obj* is the *tw_pulse* instance. As described previously (section 3.9.2), the *buildRemote* method of the *tw* class calls the *setWaveform* method before calling the *setApodization* method.

The *setWaveform* method extracts the several dedicated parameters in order to build the periodic signal. The *TwFreq* and *TxClock180MHz* parameters determine the number of samples to build a half cycle for the desired periodic signal. The *NbHcycle* parameter describes the number of half cycle to build the waveform while the *Polarity* parameter sets the sign of the first arch of the periodic signal. The periodic waveform is finally replicated for all emitting channels.

3.11.3 Inherited methods

As a subclass of the *tw* class, the *tw_pulse* class inherits several methods:

- the *buidRemote* method builds the mandatory fields of the remote structure for *tw_pulse* instances,
- the setApodization method returns a waveform after applying an apodization window,
- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tw_pulse* instance,
- the *isParam* method checks if a list of parameters belongs to the *tw_pulse* instance,
- the *setParam* method sets a parameter of the *tw_pulse* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tw_pulse* instance,
- the *isempty* method checks if all parameters of the *tw_pulse* instance (*parameter* instances and *object* instances) are empty.

3.12 The *remote.tx* class (*common.remoteobj* subclass)

The *tx* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameter	Type	Default	Auth Values	Description
TxClock180MHz	int32	1	0 1	90-MHz waveform sampling. 180-MHz waveform sampling.
twId	int32	0	0 [1 Inf]	No associated tw instance. Index of the associated tw instance.
Delays	single	0	[0 1000]	Transmit delays [µs].

Tab. 3.12 – List of the parameters of the *tx* class.

3.12.1 The *remote.tx* constructor (public) and the *initialize* method (protected)

The *tx* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tx* constructor is generating a generic *tx* instance with its *Name* variable set to "TX" and its *Desc* variable to "default transmit":

```
Obj = remote.tx();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tx* instance with its *Debug* variable set to 0:

```
Obj = remote.tx(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tx* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tx* instance and sets its *Debug* variable:

```
Obj = remote.tx(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tx* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tx* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tx(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tx* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.12), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.12.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* parameters belonging to the *tx* instance:

```
Struct = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *tx* instance.

The *buildRemote* method first calls the *setDelays* method which belongs to all *tx* subclasses in order to set the delays values to the *Delays* parameter. The method then controls the desired delays and more specifically the structure of the *Delays* parameter:

- the *Delays* parameter needs to be a vector,
- the delays for the non-emitting channels are set to 0,
- all delays emitting channels which are not belonging to the probe elements are set to 0.

The *buildRemote* method finally calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances.

3.12.3 Inherited methods

As a subclass of the *remoteobj* class, the *tx* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tx* instance,
- the *isParam* method checks if a list of parameters belongs to the *tx* instance,
- the *setParam* method sets a parameter of the *tx* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tx* instance,

- the *isempty* method checks if all parameters of the *tx* instance (*parameter* instances and *object* instances) are empty.

3.13 The *remote.tx_arbitrary* class (*remote.tx* subclass)

The *tx_arbitrary* class is a *tx* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 3.12 for the parameters, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class overrides the protected *initialize* method and implements a dedicated *setDelays* method.

3.13.1 The *remote.tx_arbitrary* constructor (public) and the *initialize* method (protected)

The *tx_arbitrary* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tx_arbitrary* constructor is generating a generic *tx_arbitrary* instance with its *Name* variable set to “TX_ARBITRARY” and its *Desc* variable to “default arbitrary transmit”:

```
Obj = remote.tx_arbitrary();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tx_arbitrary* instance with its *Debug* variable set to 0:

```
Obj = remote.tx_arbitrary(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tx_arbitrary* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tx_arbitrary* instance and sets its *Debug* variable:

```
Obj = remote.tx_arbitrary(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tx_arbitrary* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tx_arbitrary* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tx_arbitrary(Name, Desc, ParsName, ParsValue, ...,
   [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tx_arbitrary* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *tx* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, adds several controls on the input arguments as described in section 2.5.1 and the *tx* parameters (see Tab. 3.12).

Then, the *initialize* method adds the dedicated parameters (see Tab. 3.12), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.13.2 The setDelays method (protected)

The *setDelays* method updates the *Delays* parameter of the *tx_arbitrary* instance:

```
Obj = Obj.setDelays();
```

where *Obj* is the *tx_arbitrary* instance. As described previously (section 3.12.2), the *buildRemote* method of the *tx* class calls the *setDelays* method before exporting the *delays* mandatory field of the remote structure.

The *setDelays* method first sets the delays to 0 for all transmit channels, and then it updates the delays using the *Delays* parameter if its length is greater than 1. It should be noted that an error is thrown if the length of the Delays parameter is different from the number of probe elements[7] or transmitting channels[11]. Finally, the *Delays* parameter is updated with the estimated delays.

3.13.3 Inherited methods

As a subclass of the *tx* class, the *tx_arbitrary* class inherits several methods:

- the *buidRemote* method builds the mandatory fields of the remote structure for *tx_arbitrary* instances,
- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tx_arbitrary* instance,
- the *isParam* method checks if a list of parameters belongs to the *tx_arbitrary* instance,
- the *setParam* method sets a parameter of the *tx_arbitrary* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tx_arbitrary* instance,
- the *isempty* method checks if all parameters of the *tx_arbitrary* instance (*parameter* instances and *object* instances) are empty.

3.14 The *remote.tx_flat* class (*remote.tx* subclass)

The *tx_flat* class is a *tx* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 3.12 for the parameters, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has a dedicated parameter (*FlatAngle*), it overrides the protected *initialize* method and implements a dedicated *setDelays* method.

Parameter	Type	Default	Auth Values	Description

Parameter	Type	Default	Auth Values	Description
FlatAngle	single	0	[-40 40]	Flat angle value [°].

Tab. 3.13 – List of the parameters of the *tx_flat* class.

3.14.1 The *remote.tx_flat* constructor (public) and the *initialize* method (protected)

The *tx_flat* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tx_flat* constructor is generating a generic *tx_flat* instance with its *Name* variable set to "TX_FLAT" and its *Desc* variable to "default flat transmit":

```
Obj = remote.tx_flat();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tx_flat* instance with its *Debug* variable set to 0:

```
Obj = remote.tx_flat(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tx_flat* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tx_flat* instance and sets its *Debug* variable:

```
Obj = remote.tx_flat(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tx_flat* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tx_flat* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tx_flat(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tx_flat* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *tx* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, adds several controls on the input arguments as described in section 2.5.1 and the *tx* parameters (see Tab. 3.12).

Then, the *initialize* method adds the inherited and dedicated parameters (see Tab. 3.12 and Tab. 3.13), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.14.2 The setDelays method (protected)

The *setDelays* method updates the *Delays* parameter of the *tx_flat* instance:

```
Obj = Obj.setDelays();
```

where *Obj* is the *tx_flat* instance. As described previously (section 3.12.2), the *buildRemote* method of the *tx* class calls the *setDelays* method before exporting the *delays* mandatory field of the remote structure.

The *setDelays* method first retrieves the *FlatAngle* parameter and sets the delays to 0 for all transmit channels. The delay offsets are then estimated regarding the *FlatAngle* value and the probe type (linear or curved):

$$\boldsymbol{Delays} = \frac{\boldsymbol{ElPos} \cdot \sin(\boldsymbol{FlatAngle})}{\boldsymbol{SoundSpeed}} \qquad \textbf{Eq. 2}$$

$$\boldsymbol{Delays} = \frac{\sqrt{\boldsymbol{ElPos}^2 + 2 \cdot \boldsymbol{ElPos} \cdot \boldsymbol{TxFocus} \cdot \sin(\boldsymbol{FlatAngle}) + \boldsymbol{TxFocus}^2}}{\boldsymbol{SoundSpeed}} \qquad \textbf{Eq. 3}$$

where *ElPos* is the element position [m] and *SoundSpeed* the speed of sound in the medium[13].

Eq. 2 defines the delays for a linear probe. The element positions are estimated by multiplying the index of each emitting element by the probe pitch[14].

Eq. 3 defines the delays for a curved probe. The element positions correspond to the curvilinear abscissa, i.e. the index of each emitting element times the probe pitch. The *TxFocus* variable corresponds to the hypothetic position of the focus which is equal to 50 m.

3.14.3 Inherited methods

As a subclass of the *tx* class, the *tx_flat* class inherits several methods:

- the *buidRemote* method builds the mandatory fields of the remote structure for *tx_flat* instances,
- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tx_flat* instance,
- the *isParam* method checks if a list of parameters belongs to the *tx_flat* instance,
- the *setParam* method sets a parameter of the *tx_flat* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tx_flat* instance,
- the *isempty* method checks if all parameters of the *tx_flat* instance (*parameter* instances and *object* instances) are empty.

[13] The *SoundSpeed* variable is a constant variable of the *common.constants* class (see 2.1).

[14] The probe pitch corresponds to the distance between the centers of two adjacent elements of the probe. The *PrPitch* variable belongs to the *system.probe* class.

3.15 The *remote.tx_focus* class (*remote.tx* subclass)

The *tx_focus* class is a *tx* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 3.12 for the parameters, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, it overrides the protected *initialize* method and implements a dedicated *setDelays* method.

Parameter	Type	Default	Auth Values	Description
PosX	single		[0 100]	Lateral position of the focal point [mm].
PosZ	single		[O 100]	Axial position of the focal point [mm].

Tab. 3.14 – List of the parameters of the *tx_focus* class.

3.15.1 The *remote.tx_focus* constructor (public) and the *initialize* method (protected)

The *tx_focus* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *tx_focus* constructor is generating a generic *tx_focus* instance with its *Name* variable set to “TX_FOCUS” and its *Desc* variable to “default focused transmit”:

```
Obj = remote.tx_focus();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating a *tx_focus* instance with its *Debug* variable set to 0:

```
Obj = remote.tx_focus(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *tx_focus* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating a *tx_focus* instance and sets its *Debug* variable:

```
Obj = remote.tx_focus(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is a *tx_focus* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating a *tx_focus* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = remote.tx_focus(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *tx_focus* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *tx* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, adds several controls on the input arguments as described in section 2.5.1 and the *tx* parameters (see Tab. 3.12).

Then, the *initialize* method adds the inherited and dedicated parameters (see Tab. 3.12 and Tab. 3.14), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

3.15.2 The setDelays method (protected)

The *setDelays* method updates the *Delays* parameter of the *tx_focus* instance:

```
Obj = Obj.setDelays();
```

where *Obj* is the *tx_focus* instance. As described previously (section 3.12.2), the *buildRemote* method of the *tx* class calls the *setDelays* method before exporting the *delays* mandatory field of the remote structure.

The *setDelays* method first retrieves the *PosX* and *PosZ* parameters and sets the delays to 0 for all transmit channels. The delay offsets are then estimated regarding the *PosX* and *PosZ* values and the probe type (linear or curved).

For linear probes, the Cartesian position of the probe elements is determined out of the probe pitch[14]. The delays are then estimated out of the element positions and the focal point:

$$\boldsymbol{Delays} = \frac{\sqrt{(\boldsymbol{ElemtPosX} - \boldsymbol{PosX})^2 + (\boldsymbol{ElemtPosZ} - \boldsymbol{PosZ})^2}}{\boldsymbol{SoundSpeed}} \qquad \textbf{Eq. 4}$$

where *ElemtPosx* is the elements position and *Pos* the focal point position.

For curved probes, the Cartesian position of the elements is estimated as well as the focal point position according to the curvature of the probe:

$$\boldsymbol{ElemtAPos} = \frac{(\boldsymbol{ElPos} - \mathbf{0.5}) \cdot \boldsymbol{PrPitch}}{\boldsymbol{PrRadius}} \qquad \textbf{Eq. 5}$$

$$\begin{aligned} \boldsymbol{ElemtPosX} &= \boldsymbol{PrRadius} \cdot \mathbf{cos}(\boldsymbol{ElemtAPos}) \\ \boldsymbol{ElemtPosX} &= \boldsymbol{PrRadius} \cdot \mathbf{sin}(\boldsymbol{ElemtAPos}) \end{aligned} \qquad \textbf{Eq. 6}$$

$$\begin{aligned} \boldsymbol{PosX} &= (\boldsymbol{PosZ} + \boldsymbol{PrRadius}) \cdot \mathbf{cos}\left(\frac{\boldsymbol{PosX}}{\boldsymbol{PosZ} + \boldsymbol{PrRadius}}\right) \\ \boldsymbol{PosZ} &= (\boldsymbol{PosZ} + \boldsymbol{PrRadius}) \cdot \mathbf{sin}\left(\frac{\boldsymbol{PosX}}{\boldsymbol{PosZ} + \boldsymbol{PrRadius}}\right) \end{aligned} \qquad \textbf{Eq. 7}$$

where *PrPitch* is the pitch of the probe[14] and *PrRadius* the curvature radius of the probe[15]. The delays are then estimated as in the liear probe case (see Eq. 4), i.e. using the *ElemtsPos* and *Pos* values.

3.15.3 Inherited methods

As a subclass of the *tx* class, the *tx_focus* class inherits several methods:

- the *buidRemote* method builds the mandatory fields of the remote structure for *tx_focus* instances,
- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *tx_focus* instance,
- the *isParam* method checks if a list of parameters belongs to the *tx_focus* instance,
- the *setParam* method sets a parameter of the *tx_focus* instance to a new value,
- The *getParam* method returns the value of a parameter of the *tx_focus* instance,
- the *isempty* method checks if all parameters of the *tx_focus* instance (*parameter* instances and *object* instances) are empty.

4 ELUSEV

The *elusev* package contains all the classes defining groups of events, i.e. the *elusev* class and its subclasses only contain instances of event, *tx*, *tw*, *rx* and *fc* classes and subclasses. All classes of the *elusev* package are subclasses of the *common.remoteobj* class.

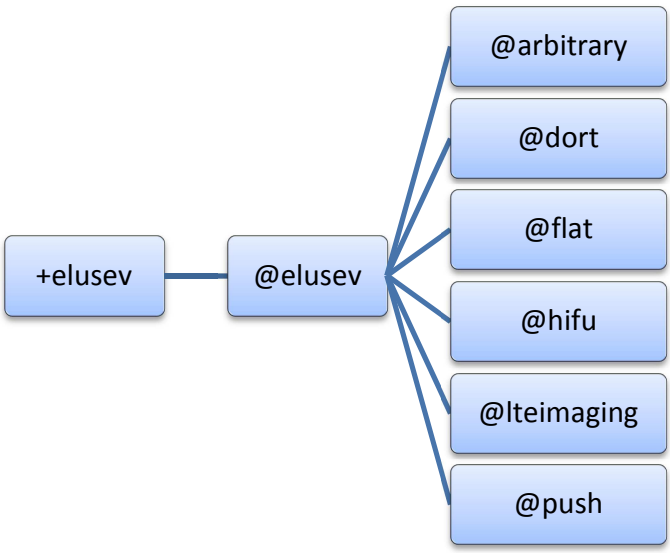

Fig. 5 – Structure and hierarchy of the *elusev* package.

[15] The probe radius corresponds to the curvature radius of the probe which characterizes its shape. The *PrRadius* variable belongs to the *system.probe* class if its *Type* variable is set to *curved*.

In addition, these classes are all inheriting from the *remoteobj* and *object* classes and follow the same structure:

- The *initialize* method is defining the parameters of the class.
- The *buildRemote* method is formatting the parameters values to build the corresponding REMOTE structure.
- The *setValue* method gives the opportunity to change parameters value.

4.1 The *elusev.elusev* class (*common.remoteobj* subclass)

The *elusev* class is a *remoteobj* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters and variables listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameters	Type	Default	Auth Values	Description
TrigIn	int32	0	0	No trigger in.
			1	Trigger in enabled.
TrigOut	single	0	0	No trigger out.
			[10^{-3} 720.8]	Duration of the trigger out.
TrigOutDelay	single	0	[0 1000]	Duration of the trigger out delay.
TrigAll	int32	0	0	Triggers only on 1st event.
			1	Triggers on all events.
Repeat	int32	1	[1 Inf]	Number of *elusev* repetitions.
TX	remote.tx			Container of *tx* instances (subclasses).
TW	remote.tw			Container of tw instances (subclasses).
RX	remote.rx			Container of rx instances.
FC	remote.fc			Container of fc instances.
Event	remote.event			Container of event insaytcen

Tab. 4.1 – List of the parameters of the *elusev* class.

4.1.1 The *elusev.elusev* constructor (public) and the *initialize* method (protected)

The *elusev* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *elusev* constructor is generating a generic *elusev* instance with its *Name* variable set to “ELUSEV” and its *Desc* variable to “default elementary ultrasound events”:

```
Obj = elusev.elusev();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating an *elusev* instance with its *Debug* variable set to 0:

```
Obj = elusev.elusev(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *elusev* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *elusev* instance and sets its *Debug* variable:

```
Obj = elusev.elusev(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is an *elusev* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *elusev* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = elusev.elusev(Name, Desc, ParsName, ParsValue, ..., [Debug]);
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *elusev* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *remoteobj* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, and adds several controls on the input arguments as described in section 2.5.1.

Then, the *initialize* method adds the dedicated parameters (see Tab. 4.1), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

4.1.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* and parameters and *remoteobj* containers belonging to the *elusev* instance:

```
Struct = Obj.buildRemote();
[Obj Struct] = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *elusev* instance. It should be noted that the second syntax of the *buildRemote* method returns the structure and the updated *elusev* instance.

The *buildRemote* method first retrieves the trigger parameters in order to set the *TriggerOutDelay* parameter to 0 if the duration of the *TriggerOut* parameter is set to 0, i.e. no trigger out needs to be generated. According to the value of the *TrigAll* parameter, the several trigger parameters (*TrigOut*, *TrigOutDelay* and *TrigIn* parameters) are set to:

- the first event of the *elusev* if *TrigAll* is set to 0,
- all events of the *elusev* if *TrigAll* is set to 1.

The *buildRemote* method then calls the *buildRemote* method of the *remoteobj* class to export all *remotepar* instances. And the method finally exports the structure associated to the several containers of the *elusev* instance, i.e. the *tx*, *tw*, *rx*, *fc* and *event* containers. It should be noted that the several containers (tx, tw, rx, fc and event parameters) are filled by using the *setParam* method.

4.1.3 Inherited methods

As a subclass of the *remoteobj* class, the *elusev* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *elusev* instance,
- the *isParam* method checks if a list of parameters belongs to the *elusev* instance,
- the *setParam* method sets a parameter of the *elusev* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *elusev* instance,
- the *isempty* method checks if all parameters of the *elusev* instance (*parameter* instances and *object* instances) are empty.

4.2 The *elusev.arbitrary* class (*elusev.elusev* subclass)

The *arbitrary* class is an *elusev* subclass which is characterized by the same variables and methods listed in the previous tables (see Tab. 2.5 for the variables, Tab. 4.1 for the parameters, Tab. 2.6 and Tab. 2.7 for the methods). In addition, this class has dedicated parameters listed below, and it overrides the protected *initialize* method and the public *buildRemote* method.

Parameters	Type	Default	Auth Values	Description
Waveform	single		[-1 1]	Waveform definition.
Delays	single	0	[0 1000]	Transmit delays for each channel [µs].
Pause	int32		[MinNoop[5] 1e6]	Pause duration after each event [us].
PauseEnd	int32		[MinNoop[5] 1e6]	Pause after last event of the elusev [µs].
ApodFct	char		{ApodFct}[10]	Apodisation function.
RxFreq	single		[1 60]	Sampling frequency [MHz].

Parameters	Type	Default	Auth Values	Description
RxCenter	single		[0 100]	Position of the receive center [mm].
RxWidth	single		[0 100]	Width of the receive aperture.
RxDuration	single		[0 1000]	Duration of the acquisition [µs].
RxDelay	single		[0 1000]	Duration of the acquisition delay [µs].
RxBandwidth	int32	1	1 2 3	200% sampling mode. 100% sampling mode. 50% sampling mode.
FIRBandwidth	int32	100	-1 [0 100]	FIR BP coeffs for impulse. Pre-calculated FIR BP coeffs.

Tab. 4.2 – List of the parameters of the *arbitrary* class.

4.2.1 The *elusev.arbitrary* constructor (public) and the *initialize* method (protected)

The *arbitrary* contructor and the *initialize* method (protected method) have the same syntaxes. The *initialize* method is always called within the constructor method (call in the *object* constructor).

An empty call of the *arbitrary* constructor is generating a generic *arbitrary* instance with its *Name* variable set to “ARBITRARY” and its *Desc* variable to “default arbitrary elementary ultrasound events”:

```
Obj = elusev.arbitrary();
```

It should be noted that if the first two input arguments are not character variables, the *Name* and *Desc* variables are set to these default values. The first syntax is creating an *arbitrary* instance with its *Debug* variable set to 0:

```
Obj = elusev.arbitrary(Name, Desc);
Obj = Obj.initialize(Name, Desc);
```

where *Obj* is a *arbitrary* instance, *Name* the name of the object, *Desc* its description.

The second syntax is creating an *arbitrary* instance and sets its *Debug* variable:

```
Obj = elusev.arbitrary(Name, Desc, Debug);
Obj = Obj.initialize(Name, Desc, Debug);
```

where *Obj* is an *arbitrary* instance, *Name* the name of the object, *Desc* its description and *Debug* is the value of the *Debug* variable, i.e. the debug mode is enabled if *Debug* = 1.

The third syntax is creating an *arbitrary* instance and sets one or several parameters (and/or objects) to specified values:

```
Obj = elusev.arbitrary(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

```
Obj = Obj.initialize(Name, Desc, ParsName, ParsValue, ..., [Debug]);
```

where *ParsName* is the name of a parameter (or object) and *ParsValue* its new value. Several parameters (and/or objects) can be set using this syntax. The only prerequisite is that there should not be any redundant definition, i.e. a single parameter can only be set once calling the constructor of the *arbitrary* class. It should be noted that the *Debug* value is optional and it is set to 0 by default.

The *initialize* method calls the *initialize* method of the *elusev* class (i.e. the *initialize* method of the *object* class) to control the syntax of the method call, as well as the constructor call, adds several controls on the input arguments as described in section 2.5.1 and the *elusev* parameters (see Tab. 4.1).

Then, the *initialize* method adds the dedicated parameters (see Tab. 4.2), and the *object* constructor calls the *setParam* method (see 2.5.4) to set parameters (and/or objects) if it was called using the third syntax.

4.2.2 The *buildRemote* method (public)

The *buidRemote* method builds a structure containing the values and the labels of all *remotepar* and parameters and *remoteobj* containers belonging to the *arbitrary* instance:

```
Struct = Obj.buildRemote();
[Obj Struct] = Obj.buildRemote();
```

where *Struct* is the returned structure variable and *Obj* the *arbitrary* instance. It should be noted that the second syntax of the *buildRemote* method returns the structure and the updated *arbitrary* instance.

As an *elusev* subclass, it is possible to add any *tx*, *tw*, *rx*, *fc* or *event* instances to an *arbitrary* instance by calling the *setParam* method. Thus, the *buildRemote* method first determines the number of added instances for the *tx*, *tw*, *rx* and *fc* containers. This initial stage is needed to properly build the arbitrary elementary ultrasound events which are built out of the parameters listed in Tab. 4.2.

After retrieving the several parameters, the buildRemote method determines the number of independent firings, i.e. it corresponds to the biggest third dimension between the *Waveform* parameter and the *Delays* parameter. Then, this method controls several parameters:

- The *Waveform* parameter – A *RepeatWf* variable is created in order to replicate *RepeatWf*-times the waveform if needed, i.e. the number of firings is greater than 1 and the third dimension of the waveform is equal to 1. Then, the waveform is replicated for all probe elements if the first dimension of the waveform is equal to 1. The waveform is truncated to the probe elements if it is defined for all transmit channels and the number of probe elements is smaller than the number of transmit channels. It should be noted that the waveform cannot be defined for a number of elements different from 1, the number of probe elements or the number of transmit channels.
- The *Delays* parameter – A *RepeatDt* variable is created in order to replicate *RepeatDt*-times the delays if needed, i.e. the number of firings is greater than 1 and the third dimension of the

delays is equal to 1. Then, the delays are replicated for all probe elements if the first dimension of the delays is equal to 1. The delays are truncated to the probe elements if they are defined for all transmit channels and the number of probe elements is smaller than the number of transmit channels. It should be noted that the delays cannot be defined for a number of elements different from 1, the number of probe elements or the number of transmit channels.

- The *RxFreq* parameter – The sampling frequency is controlled as exposed in section 3.6.2.
- The *RxDuration* parameters – The acquisition duration should correspond to the acquisition of a number of samples which is a multiple of 128 and smaller than 4096. It should be noted that the number of samples is divided by 2 if the *RxBandwidth* parameter is set to 2 (100% sampling mode), and by 4 if the *RxBandwidth* parameter is set to 3 (50% sampling mode). If the RxDuration parameter needs to be changed, it is set to the closest authorized value.
- The *RxDelay* parameter – The RxDelay parameter is set to the closest value which corresponds to an integer number of skipped samples.

The receiving elements are identified by considering the *RxCenter* and *RxWidth* parameters as well as the number of elements[7] and the pitch of the probe[14]. The type of acquisition is then determined by knowing the receiving elements, i.e. a synthetic acquisition is needed as soon as the number of receiving elements is greater than the number of receiving channels[9]. The *rx* and *fc* instances can thus be added to the arbitrary instance (according to *RxFreq*, *RxBandwidth* and *FIRBandwidth* parameters), i.e. 1 (*rx*, *fc*) for non-synthetic acquisition and 2 (*rx*, *fc*) for synthetic acquisitions.

The emission is then created for each of the firings by adding (*tx_arbitrary*, *tw_arbitrary*) considering the previously defined waveforms and delays as well as the *ApodFct* parameter. For each firing, the duration of the purely emitting event is estimated by adding the maximum delay to the number of points of the waveform times the inverse of the clock frequency[8].

The events are then created using the estimated number of samples and skipped samples, and the several *tx* instances. It should be noted that if the acquisition is synthetic, the first event for a particular *tx* instance has the smallest acceptable pause (MinNoop[5]), while the second one has a pause lasting for *Pause* µs. The duration of each event corresponds to the maximum value between the duration of the purely emitting event and the acquisition duration plus the acquisition delay.

The *buildRemote* method finally calls the *buildRemote* method of the *elusev* class to export all *remotepar* instances and the structure associated to the several containers of the *elusev* instance (*tx*, *tw*, *rx*, *fc* and *event* containers).

4.2.3 Inherited methods

As a subclass of the *elusev* class, the *arbitrary* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *arbitrary* instance,
- the *isParam* method checks if a list of parameters belongs to the *arbitrary* instance,

- the *setParam* method sets a parameter of the *arbitrary* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *arbitrary* instance,
- the *isempty* method checks if all parameters of the *arbitrary* instance (*parameter* instances and *object* instances) are empty.

4.3 The *elusev.dort* class (*elusev.elusev* subclass)

4.3.1 ELUSEV.DORT (ELUSEV.ELUSEV)

Create an ELUSEV.DORT instance.

OBJ = ELUSEV.DORT() creates a generic ELUSEV.DORT instance.

OBJ = ELUSEV.DORT(DEBUG) creates a generic ELUSEV.DORT instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ELUSEV.DORT(NAME, DESC, DEBUG) creates an ELUSEV.DORT instance with its name and description values set to NAME and DESC (character values).

OBJ = ELUSEV.DORT(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ELUSEV.DORT instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- WAVEFORM (single) sets the waveforms. [-1 1]
- PAUSE (int32) sets the pause duration after DORT events. [system.hardware.MinNoop 1e6] us
- PAUSEEND (int32) sets the pause duration at the end of the ELUSEV. [system.hardware.MinNoop 1e6] us
- DUTYCYCLE (single) sets the waveform duty cycle. [0 1]
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 1
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated- default = 100

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ELUSEV.DORT.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ELUSEV.DORT.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ELUSEV.DORT.
- GETPARAM retrieves the value/object of a ELUSEV.DORT parameter.
- SETPARAM sets the value of a ELUSEV.DORT parameter.
- ISEMPTY checks if all ELUSEV.DORT parameters are correctly defined.

4.3.2 ELUSEV.DORT.INITIALIZE (PROTECTED)

Build the remoteclass ELUSEV.DORT.

OBJ = OBJ.INITIALIZE() returns a generic ELUSEV.DORT instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ELUSEV.DORT instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ELUSEV.DORT instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ELUSEV.DORT instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- WAVEFORM (single) sets the waveforms. [-1 1]
- PAUSE (int32) sets the pause duration after DORT events. [system.hardware.MinNoop 1e6] us

- PAUSEEND (int32) sets the pause duration at the end of the ELUSEV. [system.hardware.MinNoop 1e6] us
- DUTYCYCLE (single) sets the waveform duty cycle. [0 1]
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 1
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated- default = 100

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

4.3.3 ELUSEV.DORT.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ELUSEV.DORT instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ELUSEV.DORT instance OBJ and the remote structure STRUCT.

4.3.4 Inherited methods

As a subclass of the *elusev* class, the *dort* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *dort* instance,
- the *isParam* method checks if a list of parameters belongs to the *dort* instance,
- the *setParam* method sets a parameter of the *dort* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *dort* instance,

- the *isempty* method checks if all parameters of the *dort* instance (*parameter* instances and *object* instances) are empty.

4.4 The *elusev.flat* class (*elusev.elusev* subclass)

4.4.1 ELUSEV.FLAT (ELUSEV.ELUSEV)

Create an ELUSEV.FLAT instance.

OBJ = ELUSEV.FLAT() creates a generic ELUSEV.FLAT instance.

OBJ = ELUSEV.FLAT(DEBUG) creates a generic ELUSEV.FLAT instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ELUSEV.FLAT(NAME, DESC, DEBUG) creates an ELUSEV.FLAT instance with its name and description values set to NAME and DESC (character values).

OBJ = ELUSEV.FLAT(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ELUSEV.FLAT instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the flat emission frequency. [1 15] MHz
- NBHCYCLE (int32) sets the number of half cycle. [1 200]
- POLARITY (int32) sets the waveform polarity. -1 = negative 1st arch, 1 = positive 1st arch - default = 1
- FLATANGLES (single) sets the flat angles. [-40 40] °
- PAUSE (int32) sets the pause duration after flat events. [system.hardware.MinNoop 1e6] us
- PAUSEEND (int32) sets the pause duration at the end of the ELUSEV. [system.hardware.MinNoop 1e6] us
- DUTYCYCLE (single) sets the maximum duty cycle. [0 1]
- TXCENTER (single) sets the transmission center position. [0 100] mm
- TXWIDTH (single) sets the transmission width. [0 100] mm
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, Welch
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXCENTER (single) sets the reception center position. [0 100] mm
- RXWIDTH (single) sets the reception width. [0 100] mm
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 1
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100
- PULSEINV (int32) enables the pulse inversion mode. 0 = no pulse inversion, 1 = pulse inversion - default = 0

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ELUSEV.FLAT.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ELUSEV.FLAT.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ELUSEV.FLAT.
- GETPARAM retrieves the value/object of a ELUSEV.FLAT parameter.
- SETPARAM sets the value of a ELUSEV.FLAT parameter.
- ISEMPTY checks if all ELUSEV.FLAT parameters are correctly defined.

4.4.2 ELUSEV.FLAT.INITIALIZE (PROTECTED)

Build the remoteclass ELUSEV.FLAT.

OBJ = OBJ.INITIALIZE() returns a generic ELUSEV.FLAT instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ELUSEV.FLAT instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ELUSEV.FLAT instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ELUSEV.FLAT instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the flat emission frequency. [1 15] MHz
- NBHCYCLE (int32) sets the number of half cycle. [1 200]
- POLARITY (int32) sets the waveform polarity. -1 = negative 1st arch, 1 = positive 1st arch - default = 1
- FLATANGLES (single) sets the flat angles. [-40 40] °
- PAUSE (int32) sets the pause duration after flat events. [system.hardware.MinNoop 1e6] us
- PAUSEEND (int32) sets the pause duration at the end of the ELUSEV. [system.hardware.MinNoop 1e6] us
- DUTYCYCLE (single) sets the maximum duty cycle. [0 1]
- TXCENTER (single) sets the transmission center position. [0 100] mm
- TXWIDTH (single) sets the transmission width. [0 100] mm
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, Welch
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXCENTER (single) sets the reception center position. [0 100] mm
- RXWIDTH (single) sets the reception width. [0 100] mm
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 1
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100
- PULSEINV (int32) enables the pulse inversion mode. 0 = no pulse inversion, 1 = pulse inversion - default = 0

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

4.4.3 ELUSEV.FLAT.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ELUSEV.FLAT instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ELUSEV.FLAT instance OBJ and the remote structure STRUCT.

4.4.4 Inherited methods

As a subclass of the *elusev* class, the *flat* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *flat* instance,
- the *isParam* method checks if a list of parameters belongs to the *flat* instance,
- the *setParam* method sets a parameter of the *flat* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *flat* instance,
- the *isempty* method checks if all parameters of the *flat* instance (*parameter* instances and *object* instances) are empty.

4.5 The *elusev.hifu* class (*elusev.elusev* subclass)

4.5.1 ELUSEV.HIFU (ELUSEV.ELUSEV)

Create an ELUSEV.HIFU instance.

OBJ = ELUSEV.HIFU() creates a generic ELUSEV.HIFU instance.

OBJ = ELUSEV.HIFU(DEBUG) creates a generic ELUSEV.HIFU instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ELUSEV.FLAT(NAME, DESC, DEBUG) creates an ELUSEV.HIFU instance with its name and description values set to NAME and DESC (character values).

OBJ = ELUSEV.FLAT(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ELUSEV.HIFU instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the HIFU emission frequency. [0.5 4] MHz
- TWDURATION (single) sets the HIFU emission duration. [1e3 1e6] us
- TXELEMTS (int32) sets the emitting elements. [1 system.probe.NbElemts]

- PHASE (single) sets the HIFU emission phase offsets. [0 2*pi] rad
- DUTYCYCLE (single) sets the maximum duty cycle. [0 0.9]
- RXFREQ (single) sets the sampling frequency. [0 60] MHz
- RXDURATION (single) sets the acquisition duration. 0 = no acquisition, [1 1e5] us
- RXELEMTS (int32) sets the reception elements. 0 = no acquisition, [1 system.probe.NbElemts]

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ELUSEV.HIFU.
- BUILDCAVITATION builds the cavitation events.
- BUILDEMISSION builds the emission events.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ELUSEV.HIFU.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ELUSEV.HIFU.
- GETPARAM retrieves the value/object of a ELUSEV.HIFU parameter.
- SETPARAM sets the value of a ELUSEV.HIFU parameter.

- ISEMPTY checks if all ELUSEV.HIFU parameters are correctly defined.

4.5.2 ELUSEV.HIFU.INITIALIZE (PROTECTED)

Build the remoteclass ELUSEV.HIFU.

OBJ = OBJ.INITIALIZE() returns a generic ELUSEV.HIFU instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ELUSEV.HIFU instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ELUSEV.HIFU instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ELUSEV.HIFU instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the HIFU emission frequency. [0.5 4] MHz
- TWDURATION (single) sets the HIFU emission duration. [1e3 1e6] us
- TXELEMTS (int32) sets the emitting elements. [1 system.probe.NbElemts]
- PHASE (single) sets the HIFU emission phase offsets. [0 2*pi] rad
- DUTYCYCLE (single) sets the maximum duty cycle. [0 0.9]
- RXFREQ (single) sets the sampling frequency. [0 60] MHz
- RXDURATION (single) sets the acquisition duration. 0 = no acquisition, [1 1e5] us
- RXELEMTS (int32) sets the reception elements. 0 = no acquisition, [1 system.probe.NbElemts]

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

4.5.3 ELUSEV.HIFU.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ELUSEV.HIFU instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ELUSEV.HIFU instance OBJ and the remote structure STRUCT.

4.5.4 ELUSEV.HIFU.BUILDCAVITATION (PROTECTED)

Build the associated remote structure.

RXTXTW = OBJ.BUILDCAVITATION() returns the array RXTXTW containing the definition of the passive reception events.

4.5.5 ELUSEV.HIFU.BUILDEMISSION (PROTECTED)

Build the associated remote structure.

TXTW = OBJ.BUILDEMISSION(TWDURATION) returns the array TXTW containing the definition of the emission events regarding the firing duration TWDURATION.

4.5.6 Inherited methods

As a subclass of the *elusev* class, the *hifu* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *hifu* instance,
- the *isParam* method checks if a list of parameters belongs to the *hifu* instance,
- the *setParam* method sets a parameter of the *hifu* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *hifu* instance,
- the *isempty* method checks if all parameters of the *hifu* instance (*parameter* instances and *object* instances) are empty.

4.6 The *elusev.lteimaging* class (*elusev.elusev* subclass)

4.6.1 ELUSEV.LTEIMAGING (ELUSEV.ELUSEV)

Create an ELUSEV.LTEIMAGING instance.

OBJ = ELUSEV.LTEIMAGING() creates a generic ELUSEV.LTEIMAGING instance.

OBJ = ELUSEV.LTEIMAGING(DEBUG) creates a generic ELUSEV.LTEIMAGING instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ELUSEV.LTEIMAGING(NAME, DESC, DEBUG) creates an ELUSEV.LTEIMAGING instance with its name and description values set to NAME and DESC (character values).

OBJ = ELUSEV.LTEIMAGING(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ELUSEV.LTEIMAGING instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the emission frequency. [1 15] MHz
- NBHCYCLE (int32) sets the number of half cycle. [1 600]

- DELAYS (single) sets the delays for each transmitting elements. [0 1000] us
- PAUSE (int32) sets the pause duration after events. [system.hardware.MinNoop 1000] us
- DUTYCYCLE (single) sets the maximum duty cycle. [0 1]
- TXELEMTS (int32) sets the emitting elements. [1 system.probe.NbElemts]
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, Welch
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXELEMTS (int32) sets the receiving elements. 0 = no acquisition, [1 system.probe.NbElemts]
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ELUSEV.LTEIMAGING.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ELUSEV.LTEIMAGING.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ELUSEV.LTEIMAGING.
- GETPARAM retrieves the value/object of a ELUSEV.LTEIMAGING parameter.
- SETPARAM sets the value of a ELUSEV.LTEIMAGING parameter.
- ISEMPTY checks if all ELUSEV.LTEIMAGING parameters are correctly defined.

4.6.2 ELUSEV.FLAT.LTEIMAGING (PROTECTED)

Build the remoteclass ELUSEV.LTEIMAGING.

OBJ = OBJ.INITIALIZE() returns a generic ELUSEV.LTEIMAGING instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ELUSEV.LTEIMAGING instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ELUSEV.LTEIMAGING instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ELUSEV.LTEIMAGING instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the emission frequency. [1 15] MHz
- NBHCYCLE (int32) sets the number of half cycle. [1 600]
- DELAYS (single) sets the delays for each transmitting elements. [0 1000] us
- PAUSE (int32) sets the pause duration after events. [system.hardware.MinNoop 1000] us
- DUTYCYCLE (single) sets the maximum duty cycle. [0 1]
- TXELEMTS (int32) sets the emitting elements. [1 system.probe.NbElemts]
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, Welch
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXELEMTS (int32) sets the receiving elements. 0 = no acquisition, [1 system.probe.NbElemts]
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0

- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

4.6.3 ELUSEV.LTEIMAGING.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ELUSEV.LTEIMAGING instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ELUSEV.LTEIMAGING instance OBJ and the remote structure STRUCT.

4.6.4 Inherited methods

As a subclass of the *elusev* class, the *lteimaging* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *lteimaging* instance,
- the *isParam* method checks if a list of parameters belongs to the *lteimaging* instance,
- the *setParam* method sets a parameter of the *lteimaging* instance to a new value,
- The *getParam* method returns the value of a parameter of the *lteimaging* instance,
- the *isempty* method checks if all parameters of the *lteimaging* instance (*parameter* instances and *object* instances) are empty.

4.7 The *elusev.push* class (*elusev.elusev* subclass)

4.7.1 ELUSEV.PUSH (ELUSEV.ELUSEV)

Create an ELUSEV.PUSH instance.

OBJ = ELUSEV.PUSH() creates a generic ELUSEV.PUSH instance.

OBJ = ELUSEV.PUSH(DEBUG) creates a generic ELUSEV.PUSH instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ELUSEV.PUSH(NAME, DESC, DEBUG) creates an ELUSEV.PUSH instance with its name and description values set to NAME and DESC (character values).

OBJ = ELUSEV.PUSH(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ELUSEV.PUSH instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the push emission frequency. [1 15] MHz
- DURATION (single) sets the push emission duration. [1 200] us
- PAUSE (int32) sets the pause duration after push events. [1 1000]
- DUTYCYCLE (single) sets the maximum duty cycle. [0 1]
- POSX (single) sets the focus lateral position. [0 100] mm
- POSZ (single) sets the focus axial position. [0 100] mm
- TXCENTER (single) sets the emitting center position. [0 100] mm
- RATIOFD (single) sets the ration F/D. [0 1000]
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, welch

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.
- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ELUSEV.PUSH.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ELUSEV.PUSH.

- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ELUSEV.PUSH.
- GETPARAM retrieves the value/object of a ELUSEV.PUSH parameter.
- SETPARAM sets the value of a ELUSEV.PUSH parameter.
- ISEMPTY checks if all ELUSEV.PUSH parameters are correctly defined.

4.7.2 ELUSEV.PUSH.INITIALIZE (PROTECTED)

Build the remoteclass ELUSEV.PUSH.

OBJ = OBJ.INITIALIZE() returns a generic ELUSEV.PUSH instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ELUSEV.PUSH instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ELUSEV.PUSH instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ELUSEV.PUSH instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the push emission frequency. [1 15] MHz
- DURATION (single) sets the push emission duration. [1 200] us
- PAUSE (int32) sets the pause duration after push events. [1 1000]
- DUTYCYCLE (single) sets the maximum duty cycle. [0 1]
- POSX (single) sets the focus lateral position. [0 100] mm
- POSZ (single) sets the focus axial position. [0 100] mm
- TXCENTER (single) sets the emitting center position. [0 100] mm
- RATIOFD (single) sets the ration F/D. [0 1000]
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, welch

Inherited parameters:

- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us = trigger out delay - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEAT (int32) sets the number of ELUSEV repetition. [1 Inf] - default = 1

Inherited objects:

- TX contains REMOTE.TX instances.
- TW contains REMOTE.TW instances.

- RX contains REMOTE.RX instances.
- FC contains REMOTE.FC instances.
- EVENT contains REMOTE.EVENT instances.

4.7.3 ELUSEV.PUSH.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ELUSEV.PUSH instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ELUSEV.PUSH instance OBJ and the remote structure STRUCT.

4.7.4 Inherited methods

As a subclass of the *elusev* class, the *push* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *push* instance,
- the *isParam* method checks if a list of parameters belongs to the *push* instance,
- the *setParam* method sets a parameter of the *push* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *push* instance,
- the *isempty* method checks if all parameters of the *push* instance (*parameter* instances and *object* instances) are empty.

5 ACMO

The *acmo* package contains all the classes defining an ACquisition MOde, i.e. the *acmo* class and its subclasses contain *mode*, *tgc*, *dmacontrol* and *elusev* classes and subclasses. All classes of the *acmo* package are subclasses of the *common.remoteobj* class. In addition, these classes are all inheriting from the *remoteobj* and *object* classes and follow the same structure:

- The *initialize* method is defining the parameters of the class.
- The *buildRemote* method is formatting the parameters values to build the corresponding REMOTE structure.
- The *setValue* method gives the opportunity to change parameters value.

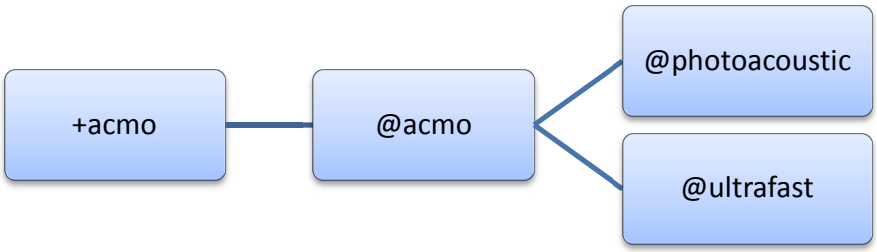

Fig. 6 – Structure and hierarchy of the *acmo* package.

5.1 The *acmo.acmo* class (*common.remoteobj* subclass)

5.1.1 ACMO.ACMO (COMMON.REMOTEOBJ)

Create an ACMO.ACMO instance.

OBJ = ACMO.ACMO() creates a generic ACMO.ACMO instance.

OBJ = ACMO.ACMO(DEBUG) creates a generic ACMO.ACMO instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ACMO.ACMO(NAME, DESC, DEBUG) creates an ACMO.ACMO instance with its name and description values set to NAME and DESC (character values).

OBJ = ACMO.ACMO(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ACMO.ACMO instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- REPEAT (int32) sets the number of ACMO repetition. [1 Inf] - default = 1
- DURATION (single) sets the TGC duration. [0 Inf] us - default = 0
- CONTROLPTS (single) sets the value of TGC control points. [0 960] - default = 900
- FASTTGC (int32) enables the fast TGC mode. 0 = classical TGC, 1 = fast TGC - default = 0
- NBHOSTBUFFER (int32) sets the number of host buffer. [1 10] - default = 2
- ORDERING (int32) sets the ELUSEV execution order during the ACMO. 0 = chronological, [1 Inf] = customed order - default = 0

Dedicated objects:

- MODE contains a single REMOTE.MODE instance.
- TGC contains a single REMOTE.TGC instance.
- DMACONTROL contains a single REMOTE.DMACONTROL instance.
- ELUSEV contains ELUSEV.ELUSEV instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ACMO.ACMO.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ACMO.ACMO.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ACMO.ACMO.
- GETPARAM retrieves the value/object of a ACMO.ACMO parameter.
- SETPARAM sets the value of a ACMO.ACMO parameter.
- ISEMPTY checks if all ACMO.ACMO parameters are correctly defined.

5.1.2 ACMO.ACMO.INITIALIZE (PROTECTED)

Build the remoteclass ACMO.ACMO.

OBJ = OBJ.INITIALIZE() returns a generic ACMO.ACMO instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ACMO.ACMO instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ACMO.ACMO instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ACMO.ACMO instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- REPEAT (int32) sets the number of ACMO repetition. [1 Inf] - default = 1
- DURATION (single) sets the TGC duration. [0 Inf] us - default = 0
- CONTROLPTS (single) sets the value of TGC control points. [0 960] - default = 900
- FASTTGC (int32) enables the fast TGC mode. 0 = classical TGC, 1 = fast TGC - default = 0
- NBHOSTBUFFER (int32) sets the number of host buffer. [1 10] - default = 2
- ORDERING (int32) sets the ELUSEV execution order during the ACMO. 0 = chronological, [1 Inf] = customed order - default = 0

Dedicated objects:

- MODE contains a single REMOTE.MODE instance.
- TGC contains a single REMOTE.TGC instance.
- DMACONTROL contains a single REMOTE.DMACONTROL instance.
- ELUSEV contains ELUSEV.ELUSEV instances.

5.1.3 ACMO.ACMO.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ACMO.ACMO instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ACMO.ACMO instance OBJ and the remote structure STRUCT.

5.1.4 Inherited methods

As a subclass of the *remoteobj* class, the *acmo* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *acmo* instance,
- the *isParam* method checks if a list of parameters belongs to the *acmo* instance,
- the *setParam* method sets a parameter of the *acmo* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *acmo* instance,
- the *isempty* method checks if all parameters of the *acmo* instance (*parameter* instances and *object* instances) are empty.

5.2 The *acmo.photoacoustic* class (*acmo.acmo* subclass)

5.2.1 ACMO.PHOTOACOUSTIC (ACMO.ACMO)

Create an ACMO.PHOTOACOUSTIC instance.

OBJ = ACMO.PHOTOACOUSTIC() creates a generic ACMO.PHOTOACOUSTIC instance.

OBJ = ACMO.PHOTOACOUSTIC(DEBUG) creates a generic ACMO.PHOTOACOUSTIC instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ACMO.PHOTOACOUSTIC(NAME, DESC, DEBUG) creates an ACMO.PHOTOACOUSTIC instance with its name and description values set to NAME and DESC (character values).

OBJ = ACMO.PHOTOACOUSTIC(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ACMO.PHOTOACOUSTIC instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- NBFRAMES (int32) sets the number of acquired frames. [1 1000] - default = 10
- PRF (single) sets the pulse repetition frequency. [1 - 10000] Hz
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXCENTER (single) sets the receive center position. [0 100] mm
- SYNTACQ (int32) enables synthetic acquisition. 0 = classical, 1 = synthetic - default = 0
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us - default = 1
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 1
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100
- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us - default = 0

Inherited parameters:

- REPEAT (int32) sets the number of ACMO repetition. [1 Inf] - default = 1
- DURATION (single) sets the TGC duration. [0 Inf] us - default = 0
- CONTROLPTS (single) sets the value of TGC control points. [0 960] - default = 900
- FASTTGC (int32) enables the fast TGC mode. 0 = classical TGC, 1 = fast TGC - default = 0
- NBHOSTBUFFER (int32) sets the number of host buffer. [1 10] - default = 2
- ORDERING (int32) sets the ELUSEV execution order during the ACMO. 0 = chronological, [1 Inf] = customed order - default = 0

Inherited objects:

- MODE contains REMOTE.MODE instances.
- TGC contains REMOTE.TGC instances.
- DMACONTROL contains REMOTE.DMACONTROL instances.
- ELUSEV contains ELUSEV.ELUSEV instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ACMO.PHOTOACOUSTIC.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ACMO.PHOTOACOUSTIC.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ACMO.PHOTOACOUSTIC.
- GETPARAM retrieves the value/object of a ACMO.PHOTOACOUSTIC parameter.
- SETPARAM sets the value of a ACMO.PHOTOACOUSTIC parameter.
- ISEMPTY checks if all ACMO.PHOTOACOUSTIC parameters are correctly defined.

5.2.2 ACMO.PHOTOACOUSTIC.INITIALIZE (PROTECTED)

Build the remoteclass ACMO.PHOTOACOUSTIC.

OBJ = OBJ.INITIALIZE() returns a generic ACMO.PHOTOACOUSTIC instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ACMO.PHOTOACOUSTIC instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ACMO.PHOTOACOUSTIC instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ACMO.PHOTOACOUSTIC instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- NBFRAMES (int32) sets the number of acquired frames. [1 1000] - default = 10
- PRF (single) sets the pulse repetition frequency. [1 - 10000] Hz
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXCENTER (single) sets the receive center position. [0 100] mm
- SYNTACQ (int32) enables synthetic acquisition. 0 = classical, 1 = synthetic - default = 0
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us - default = 1
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 1
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100
- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us - default = 0

Inherited parameters:

- REPEAT (int32) sets the number of ACMO repetition. [1 Inf] - default = 1
- DURATION (single) sets the TGC duration. [0 Inf] us - default = 0
- CONTROLPTS (single) sets the value of TGC control points. [0 960] - default = 900
- FASTTGC (int32) enables the fast TGC mode. 0 = classical TGC, 1 = fast TGC - default = 0
- NBHOSTBUFFER (int32) sets the number of host buffer. [1 10] - default = 2
- ORDERING (int32) sets the ELUSEV execution order during the ACMO. 0 = chronological, [1 Inf] = customed order - default = 0

Inherited objects:

- MODE contains REMOTE.MODE instances.
- TGC contains REMOTE.TGC instances.
- DMACONTROL contains REMOTE.DMACONTROL instances.
- ELUSEV contains ELUSEV.ELUSEV instances.

5.2.3 ACMO.PHOTOACOUSTIC.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ACMO.PHOTOACOUSTIC instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ACMO.PHOTOACOUSTIC instance OBJ and the remote structure STRUCT.

5.2.4 Inherited methods

As a subclass of the *acmo* class, the *photoacoustic* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *photoacoustic* instance,
- the *isParam* method checks if a list of parameters belongs to the *photoacoustic* instance,
- the *setParam* method sets a parameter of the *photoacoustic* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *photoacoustic* instance,
- the *isempty* method checks if all parameters of the *photoacoustic* instance (*parameter* instances and *object* instances) are empty.

5.3 The *acmo.ultrafast* class (*acmo.acmo* subclass)

5.3.1 ACMO.ULTRAFAST (ACMO.ACMO)

Create an ACMO.ULTRAFAST instance.

OBJ = ACMO.ULTRAFAST() creates a generic ACMO.ULTRAFAST instance.

OBJ = ACMO.ULTRAFAST(DEBUG) creates a generic ACMO.ULTRAFAST instance using the DEBUG value (1 is enabling the debug mode).

OBJ = ACMO.ULTRAFAST(NAME, DESC, DEBUG) creates an ACMO.ULTRAFAST instance with its name and description values set to NAME and DESC (character values).

OBJ = ACMO.ULTRAFAST(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an ACMO.ULTRAFAST instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the flat emission frequency. [1 15] MHz
- NBHCYCLE (int32) sets the number of half cycle. [1 200]
- FLATANGLES (single) sets the flat angles. [-40 40] °
- PRF (single) sets the pulse repetition frequency. [1 10000] Hz
- DUTYCYLE (single) sets the maximum duty cycle. [0 1]
- TXCENTER (single) sets the transmit center position. [0 100] mm
- TXWIDTH (single) sets the transmit width. [0 100] mm
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, Welch
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXCENTER (single) sets the receive center position. [0 100] mm
- RXWIDTH (single) sets the receive width. [0 100] mm
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us

- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 2
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100
- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us - default = 0
- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEATFLAT (int32) sets the number of events repetition. [1 1000]
- PULSEINV (int32) enables the pulse inversion mode. 0 = no pulse inversion, 1 = pulse inversion - default = 0

Inherited parameters:

- REPEAT (int32) sets the number of ACMO repetition. [1 Inf] - default = 1
- DURATION (single) sets the TGC duration. [0 Inf] us - default = 0
- CONTROLPTS (single) sets the value of TGC control points. [0 960] - default = 900
- FASTTGC (int32) enables the fast TGC mode. 0 = classical TGC, 1 = fast TGC - default = 0
- NBHOSTBUFFER (int32) sets the number of host buffer. [1 10] - default = 2
- ORDERING (int32) sets the ELUSEV execution order during the ACMO. 0 = chronological, [1 Inf] = customed order - default = 0

Inherited objects:

- MODE contains REMOTE.MODE instances.
- TGC contains REMOTE.TGC instances.
- DMACONTROL contains REMOTE.DMACONTROL instances.
- ELUSEV contains ELUSEV.ELUSEV instances.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass ACMO.ULTRAFAST.
- BUILDREMOTE builds the associated remote structure.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass ACMO.ULTRAFAST.

- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass ACMO.ULTRAFAST.
- GETPARAM retrieves the value/object of a ACMO.ULTRAFAST parameter.
- SETPARAM sets the value of a ACMO.ULTRAFAST parameter.
- ISEMPTY checks if all ACMO.ULTRAFAST parameters are correctly defined.

5.3.2 ACMO.ULTRAFAST.INITIALIZE (PROTECTED)

Build the remoteclass ACMO.ULTRAFAST.

OBJ = OBJ.INITIALIZE() returns a generic ACMO.ULTRAFAST instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic ACMO.ULTRAFAST instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an ACMO.ULTRAFAST instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an ACMO.ULTRAFAST instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TWFREQ (single) sets the flat emission frequency. [1 15] MHz
- NBHCYCLE (int32) sets the number of half cycle. [1 200]
- FLATANGLES (single) sets the flat angles. [-40 40] °
- PRF (single) sets the pulse repetition frequency. [1 10000] Hz
- DUTYCYLE (single) sets the maximum duty cycle. [0 1]
- TXCENTER (single) sets the transmit center position. [0 100] mm
- TXWIDTH (single) sets the transmit width. [0 100] mm
- APODFCT (char) sets the apodisation function. none, bartlett, blackman, connes, cosine, gaussian, hamming, hanning, Welch
- RXFREQ (single) sets the sampling frequency. [1 60] MHz
- RXCENTER (single) sets the receive center position. [0 100] mm
- RXWIDTH (single) sets the receive width. [0 100] mm
- RXDURATION (single) sets the acquisition duration. [0 1000] us
- RXDELAY (single) sets the acquisition delay. [0 1000] us
- RXBANDWIDTH (int32) sets the decimation mode. 1 = 200%, 2 = 100%, 3 = 50% - default = 2
- FIRBANDWIDTH (int32) sets the digital filter coefficients. -1 = none, [0 100] = pre-calculated - default = 100
- TRIGIN (int32) enables the trigger in. 0 = no trigger in, 1 = trigger in - default = 0
- TRIGOUT (single) enables the trigger out. 0 = no trigger out, [1e-3 720.8] us = trigger out duration - default = 0
- TRIGOUTDELAY (single) sets the trigger out delay. [0 1000] us - default = 0

- TRIGALL (int32) enables triggers on all events. 0 = triggers on 1st event, 1 = triggers on all events - default = 0
- REPEATFLAT (int32) sets the number of events repetition. [1 1000]
- PULSEINV (int32) enables the pulse inversion mode. 0 = no pulse inversion, 1 = pulse inversion - default = 0

Inherited parameters:

- REPEAT (int32) sets the number of ACMO repetition. [1 Inf] - default = 1
- DURATION (single) sets the TGC duration. [0 Inf] us - default = 0
- CONTROLPTS (single) sets the value of TGC control points. [0 960] - default = 900
- FASTTGC (int32) enables the fast TGC mode. 0 = classical TGC, 1 = fast TGC - default = 0
- NBHOSTBUFFER (int32) sets the number of host buffer. [1 10] - default = 2
- ORDERING (int32) sets the ELUSEV execution order during the ACMO. 0 = chronological, [1 Inf] = customed order - default = 0

Inherited objects:

- MODE contains REMOTE.MODE instances.
- TGC contains REMOTE.TGC instances.
- DMACONTROL contains REMOTE.DMACONTROL instances.
- ELUSEV contains ELUSEV.ELUSEV instances.

5.3.3 ACMO.ULTRAFAST.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

STRUCT = OBJ.BUILDREMOTE() returns the remote structure STRUCT containing all mandatory remote fields for the ACMO.ULTRAFAST instance.

[OBJ STRUCT] = OBJ.BUILDREMOTE() returns the updated ACMO.ULTRAFAST instance OBJ and the remote structure STRUCT.

5.3.4 Inherited methods

As a subclass of the *acmo* class, the *ultrafast* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *ultrafast* instance,
- the *isParam* method checks if a list of parameters belongs to the *ultrafast* instance,
- the *setParam* method sets a parameter of the *ultrafast* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *ultrafast* instance,
- the *isempty* method checks if all parameters of the *ultrafast* instance (*parameter* instances and *object* instances) are empty.

6 USSE

The *usse* package contains all the classes defining an ULtrasound SEquence, i.e. the usse class and its subclasses contain *tpc* and *acmo* classes and subclasses. All classes of the *usse* package are subclasses of the *common.remoteobj* class. In addition, these classes are all inheriting from the *remoteobj* and *object* classes and follow the same structure:

- The *initialize* method is defining the parameters of the class.
- The *buildRemote* method is formatting the parameters values to build the corresponding REMOTE structure.
- The *setValue* method gives the opportunity to change parameters value.

It should be noted that additional methods are implemented in order to control the remote managed system as well as loading the ultrasound sequence and retrieving data.

Fig. 7 – Structure and hierarchy of the *usse* package.

6.1 The *usse.usse* class (*common.remoteobj* subclass)

6.1.1 USSE.USSE (COMMON.REMOTEOBJ)

Create an USSE.USSE instance.

OBJ = USSE.USSE() creates a generic USSE.USSE instance.

OBJ = USSE.USSE(DEBUG) creates a generic USSE.USSE instance using the DEBUG value (1 is enabling the debug mode).

OBJ = USSE.USSE(NAME, DESC, DEBUG) creates an USSE.USSE instance with its name and description values set to NAME and DESC (character values).

OBJ = USSE.USSE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an USSE.USSE instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- REPEAT (int32) sets the number of USSE repetition. [1 Inf]
- LOOP (int32) enables the loop mode. 0 = single execution sequence, 1 = loop the sequence
- DATAFORMAT (char) sets the format of output data. RF = RF data, BF = beamformed data, FF = final frame data - default = RF
- ORDERING (int32)sets the ACMO execution order during the USSE. 0 = chronological, [1 Inf] = customed order - default = 0

Dedicated objects:

- TPC contains a single REMOTE.TPC instance.
- ACMO contains ACMO.ACMO instances.

Dedicated variables:

- SERVER contains the remote server parameters.
- INFOSTRUCT contains general information on the ultrasound sequence.

Inherited variables:

- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass USSE.USSE.
- BUILDREMOTE builds the associated remote structure.
- BUILDINFO builds the INFOSTRUCT structure.
- CALLBACKS manages GUIs callbacks.
- SELECTHARDWARE selects the system hardware.
- SELECTPROBE selects the probe.
- INITIALIZEREMOTE initializes the remote server.
- QUITREMOTE switches the system to user level.
- STOPSSYSTEM stops the remote host.
- LOADSEQUENCE loads the ultrasound sequence.
- STARTSEQUENCE starts the loaded ultrasound sequence.
- STOPSEQUENCE stops the loaded ultrasound sequence.
- GETDATA waits for data and retrieve them.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass USSE.USSE.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass USSE.USSE.
- GETPARAM retrieves the value/object of a USSE.USSE parameter.
- SETPARAM sets the value of a USSE.USSE parameter.
- ISEMPTY checks if all USSE.USSE parameters are correctly defined.

6.1.2 USSE.USSE.INITIALIZE (PROTECTED)

Build the remoteclass USSE.USSE.

OBJ = OBJ.INITIALIZE() returns a generic USSE.USSE instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic USSE.USSE instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an USSE.USSE instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an USSE.USSE instance with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- REPEAT (int32) sets the number of USSE repetition. [1 Inf]
- LOOP (int32) enables the loop mode. 0 = single execution sequence, 1 = loop the sequence
- DATAFORMAT (char) sets the format of output data. RF = RF data, BF = beamformed data, FF = final frame data - default = RF
- ORDERING (int32)sets the ACMO execution order during the USSE. 0 = chronological, [1 Inf] = customed order - default = 0

Dedicated objects:

- TPC contains a single REMOTE.TPC instance.
- ACMO contains ACMO.ACMO instances.

6.1.3 USSE.USSE.SELECTPROBE (PUBLIC)

Selects the probe.

OBJ = OBJ.SELECTPROBE() selects the probe.

6.1.4 USSE.USSE.SELECTHARDWARE (PUBLIC)

Selects the system hardware.

OBJ = OBJ.SELECTHARDWARE() selects the system hardware.

6.1.5 USSE.USSE.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

OBJ = OBJ.BUILDREMOTE() returns the updated USSE.USSE instance OBJ.

[OBJ NBDMA] = OBJ.BUILDREMOTE() returns the updated USSE.USSE instance OBJ and the number of independent acquisitions.

6.1.6 USSE.USSE.BUILDINFO (PROTECTED)

Build the INFOSTRUCT structure.

OBJ = OBJ.BUILDINFO() builds the INFOSTRUCT structure.

6.1.7 USSE.USSE.INITIALIZEREMOTE (PUBLIC)

Initialize the remote server.

OBJ = OBJ.INITIALIZEREMOTE() initializes the remote server characterized by the SERVER variable of the USSE.USSE instance.

OBJ = OBJ.INITIALIZEREMOTE(PARNAME, PARVALUE, ...) initializes the remote server with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- IPADDRESS (char) sets the IP address of the remote server.

6.1.8 USSE.USSE.LOADSEQUENCE (PUBLIC)

Load the ultrasound sequence.

OBJ = OBJ.LOADSEQUENCE() loads the ultrasound sequence of the USSE.USSE instance.

6.1.9 USSE.USSE.STARTSEQUENCE (PUBLIC)

Start the loaded ultrasound sequence.

OBJ = OBJ.STARTSEQUENCE() starts the loaded ultrasound sequence.

OBJ = OBJ.STARTSEQUENCE(PARNAME, PARVALUE, ...) starts the loaded ultrasound sequence with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- WAIT (int32) wait parameters. 0 = start immediatly, 1 = wait for previous sequence to stop

6.1.10 USSE.USSE.CALLBACKS (PROTECTED)

Manages GUIs callbacks.

OBJ.CALLBACKS(SRC, EVENT) manages GUIs callbacks coming from the SRC handles and corresponding to the EVENT event.

6.1.11 USSE.USSE.GETDATA (PUBLIC)

Wait for data and retrieve them.

BUFFER = OBJ.GETDATA() returns the structure BUFFER containing the acquired data as well as several data descriptors.

BUFFER = OBJ.GETDATA(PARNAME, PARVALUE, ...) returns the structure BUFFER with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- REALIGN (int32) enables the realignment of RF data. 0 = no realignment, 1 = realignment

6.1.12 USSE.USSE.STOPSEQUENCE (PUBLIC)

Stop the loaded ultrasound sequence.

OBJ = OBJ.STOPSEQUENCE() stops the loaded ultrasound sequence.

OBJ = OBJ.STOPSEQUENCE(PARNAME, PARVALUE, ...) stops the loaded ultrasound sequence with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- WAIT (int32) wait parameter. 0 = stop immediatly, 1 = wait for the sequence to end

6.1.13 USSE.USSE.QUITREMOTE (PUBLIC)

Switch the system to user level.

OBJ = OBJ.QUITREMOTE() switches the system to user level.

OBJ = OBJ.INITIALIZEREMOTE(PARNAME, PARVALUE, ...) initializes the remote server with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- IMAGINGMODE (char) sets the imaging mode. B = B-mode, COL - color, SWE = elastography, PW = power doppler mode

6.1.14 USSE.USSE.STOPSYSTEM (PUBLIC)

Stop the remote host.

OBJ = OBJ.STOPSYSTEM() stops the remote host.

OBJ = OBJ.STOPSYSTEM(PARNAME, PARVALUE, ...) stops the remote host with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- IPADDRESS (char) sets the IP address of the remote server.

6.1.15 Inherited methods

As a subclass of the *remoteobj* class, the *usse* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *usse* instance,
- the *isParam* method checks if a list of parameters belongs to the *usse* instance,
- the *setParam* method sets a parameter of the *usse* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *usse* instance,
- the *isempty* method checks if all parameters of the *usse* instance (*parameter* instances and *object* instances) are empty.

6.2 The *usse.lte* class (*usse.usse* subclass)

6.2.1 USSE.LTE (USSE.USSE)

Create an USSE.LTE instance.

OBJ = USSE.LTE() creates a generic USSE.LTE instance.

OBJ = USSE.LTE(DEBUG) creates a generic USSE.LTE instance using the DEBUG value (1 is enabling the debug mode).

OBJ = USSE.LTE(NAME, DESC, DEBUG) creates an USSE.LTE instance with its name and description values set to NAME and DESC (character values).

OBJ = USSE.LTE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) creates an USSE.LTE instance with parameters PARSNAME set to PARSVALUE.

Inherited parameters:

- REPEAT (int32) sets the number of USSE repetition. [1 Inf]
- LOOP (int32) enables the loop mode. 0 = single execution sequence, 1 = loop the sequence
- DATAFORMAT (char) sets the format of output data. RF = RF data, BF = beamformed data, FF = final frame data - default = RF
- ORDERING (int32)sets the ACMO execution order during the USSE. 0 = chronological, [1 Inf] = customed order - default = 0

Inherited objects:

- TPC contains a single REMOTE.TPC instance.
- ACMO contains ACMO.ACMO instances.

Inherited variables:

- SERVER contains the remote server parameters - default = 192.168.1.15.
- INFOSTRUCT contains general information on the ultrasound sequence.
- NAME contains the name of the object.
- TYPE contains the type of the object.
- DESC contains a description of the object.
- DEBUG enables the debugging mode.

Dedicated functions:

- INITIALIZE builds the remoteclass USSE.USSE.
- BUILDREMOTE builds the associated remote structure.
- ADDHIFU adds an HIFU block to the sequence.
- ADDIMAGING adds an LTEIMAGING block to the sequence.
- SETVOLTAGE sets the Magna Power voltage.
- GETDATA waits for data and retrieves them.

Inherited functions:

- ISPARAM checks if a parameter (or several parameters) already belongs to the remoteclass USSE.USSE.
- ADDPARAM adds a parameter/object (explicit definition and cell array definition) to the remoteclass USSE.USSE.
- GETPARAM retrieves the value/object of a USSE.USSE parameter.
- SETPARAM sets the value of a USSE.USSE parameter.
- ISEMPTY checks if all USSE.USSE parameters are correctly defined.
- BUILDINFO builds the INFOSTRUCT structure.
- CALLBACKS manages GUIs callbacks.
- SELECTHARDWARE selects the system hardware.
- SELECTPROBE selects the probe.

- INITIALIZEREMOTE initializes the remote server.
- QUITREMOTE switches the system to user level.
- STOPSSYSTEM stops the remote host.
- LOADSEQUENCE loads the ultrasound sequence.
- STARTSEQUENCE starts the loaded ultrasound sequence.
- STOPSEQUENCE stops the loaded ultrasound sequence.

6.2.2 USSE.LTE.INITIALIZE (PROTECTED)

Build the remoteclass USSE.USSE.

OBJ = OBJ.INITIALIZE() returns a generic USSE.LTE instance.

OBJ = OBJ.INITIALIZE(DEBUG) returns a generic USSE.LTE instance using the DEBUG value (1 is enabling the debug mode).

OBJ = OBJ.INITIALIZE(NAME, DESC, DEBUG) returns an USSE.LTE instance with its name and description values set to NAME and DESC (character values).

OBJ = OBJ.INITIALIZE(NAME, DESC, PARNAME, PARVALUE, ..., DEBUG) returns an USSE.LTE instance with parameters PARSNAME set to PARSVALUE.

Inherited parameters:

- REPEAT (int32) sets the number of USSE repetition. [1 Inf]
- LOOP (int32) enables the loop mode. 0 = single execution sequence, 1 = loop the sequence
- DATAFORMAT (char) sets the format of output data. RF = RF data, BF = beamformed data, FF = final frame data - default = RF
- ORDERING (int32)sets the ACMO execution order during the USSE. 0 = chronological, [1 Inf] = customed order - default = 0

Inherited objects:

- TPC contains a single REMOTE.TPC instance.
- ACMO contains ACMO.ACMO instances.

6.2.3 USSE.LTE.ADDIMAGING (PUBLIC)

Stop the loaded ultrasound sequence.

OBJ = OBJ.ADDIMAGING() adds an LTEIMAGING block to the sequence.

OBJ = OBJ.ADDIMAGING(PARNAME, PARVALUE, ...) adds an LTEIMAGING block to the sequence with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- TRFREQ (single) sets the emitting frequency. [1 15] MHz - default = 1
- NCYCLES (int32) sets the number of cycle. [1 600] - default = 3
- TRANSMITELEMTS (int32) sets the tx channels. [1 system.probe.NbElemts] - default = 1: system.probe.NbElemts.

- DELAYS (single) sets the delays for each channels. [0 1000] us - default = 0
- MAXDUTY (single) sets the waveform duty cycle. [0 1] - default = 0.9
- RXFREQ (single) sets the sampling frequency. [1 60] MHz - default = 45
- ACQDURATION (single) sets the sampling duration. [0 1000] us - default = 0
- RCVELEMTS (int32) sets the receiving elements. 0 = no acquisition, [1 system.probe.NbElemts] - default = 1
- TRIGGERMODE (int32) sets the trigger mode. 0 = none, 1 = trigger in, 2 = trigger out, 3 = trigger in & out - default = 0
- TGCCONTROLPTS (single) sets the TGC control points. [0 960] - default = 950
- REPEAT (int32) sets the number of LTEIMAGING repetition. [1 Inf] - default = 1
- FASTTGC (int32) enables the fast TGC. default = 0

6.2.4 USSE.LTE.ADDHIFU (PUBLIC)

Stop the loaded ultrasound sequence.

OBJ = OBJ.ADDHIFU() adds an HIFU block to the sequence.

OBJ = OBJ.ADDHIFU(PARNAME, PARVALUE, ...) adds an HIFU block to the sequence with parameters PARSNAME set to PARSVALUE.

Dedicated parameters:

- HIFUFREQ (single) sets the emitting frequency. [0.5 4] MHz - default = 1
- FIRINGDURATION (single) sets the HIFU duration. [0.001 1] s - default = 0.1 s
- TRANSMITELEMTS (int32) sets the tx channels. [1 system.probe.NbElemts] - default = 1:system.probe.NbElemts
- PHASE (single) sets the phase offsets. [0 2pi] rad - default = 0
- MAXDUTY (single) sets the waveform duty cycle. [0 0.9] - default = 0.9
- RXFREQ (single) sets the sampling frequency. [1 60] MHz - default = 45
- ACQDURATION (single) sets the sampling duration. 0 = no acquisition, [1e-6 0.1] s - default = 0
- RCVELEMTS (int32) sets the receiving elements. 0 = no acquisition, [1 system.probe.NbElemts] - default = 1
- TRIGGERMODE (int32) sets the trigger mode. 0 = none, 1 = trigger in, 2 = trigger out, 3 = trigger in & out – default = 0
- CONTROLPTS (single) sets the TGC control points. [0 960] - default = 950
- REPEAT (int32) sets the number of LTEIMAGING repetition. [1 Inf] - default = 1
- FASTTGC (int32) enables the fast TGC. 0 = classical TGC, 1 = fast TGC - default = 0

6.2.5 USSE.LTE.BUILDREMOTE (PUBLIC)

Build the associated remote structure.

OBJ = OBJ.BUILDREMOTE() returns the updated USSE.LTE instance OBJ.

[OBJ NBDMA] = OBJ.BUILDREMOTE() returns the updated USSE.LTE instance OBJ and the number of independent acquisitions.

6.2.6 USSE.LTE.GETDATA (PUBLIC)

Wait for data and retrieve them.

BUFFER = OBJ.GETDATA() returns the structure BUFFER containing the acquired data as well as several data descriptors.

6.2.7 USSE.LTE.SETVOLTAGE (PUBLIC)

Sets the Magna Power voltage.

OBJ = OBJ.SETVOLTAGE(VOLTAGE) sets the Magna Power voltage to VOLTAGE.

6.2.8 Inherited methods

As a subclass of the *usse* class, the *lte* class inherits several methods:

- the *addParam* method adds a *parameter* instance (or an *object* instance) to the *lte* instance,
- the *isParam* method checks if a list of parameters belongs to the *lte* instance,
- the *setParam* method sets a parameter of the *lte* instance to a new value or add a new value to an existing container,
- The *getParam* method returns the value of a parameter of the *lte* instance,
- the *isempty* method checks if all parameters of the *lte* instance (*parameter* instances and *object* instances) are empty.

6.3 USSE.CONTROLSEQUENCE (PUBLIC)

Control if a sequence has been manually stopped.

STOP = CONTROLSEQUENCE() returns 1 if a sequence has been stopped manually, i.e. the STOP button of the dialog box has been pressed.

7 SYSTEM

7.1 The *system.hardware* class

7.1.1 SYSTEM.HARDWARE

Class containing hardware constants.

Constants:

- NAME corresponds to the name of the hardware.
- NBDAB corresponds to the number of DAB.
- NBADB corresponds to the number of ADB.
- NBTXPADB corresponds to the number of TX channels per ADBs.
- NBRXPADB corresponds to the number of RX channels per ADBs.

- NBTXCHAN corresponds to the number of TX channels.
- NBRXCHAN corresponds to the number of RX channels.
- CLOCKFREQ corresponds to the clock frequency.
- ADFILTER corresponds to the accepted decimation rates for the AD input filter.
- ADRATE corresponds to the accepted AD sampling frequencies.
- MINNNOOP corresponds to the minimal duration of the noop variable.
- MAXNSAMPLES corresponds to the maximum number of samples per waveform.

7.1.2 Aixplorer

Constants:

- NAME = Aixplorer.
- NBDAB = 2.
- NBADB = 16.
- NBTXPADB = 8.
- NBRXPADB = 4.
- NBTXCHAN = 256.
- NBRXCHAN = 128.
- CLOCKFREQ = 180.
- ADFILTER = [1, 2, 3, 4].
- ADRATE = [3, 4, 5, 6, 7, 8, 9].
- MINNNOOP = 5.
- MAXNSAMPLES = 1024.

7.1.3 Liver Therapy

Constants:

- NAME = Liver Therapy.
- NBDAB = 2.
- NBADB = 16.
- NBTXPADB = 8.
- NBRXPADB = 4.
- NBTXCHAN = 256.
- NBRXCHAN = 128.
- CLOCKFREQ = 180.
- ADFILTER = [1, 2, 3, 4].
- ADRATE = [3, 4, 5, 6, 7, 8, 9].
- MINNNOOP = 5.
- MAXNSAMPLES = 400.

7.1.4 Brain Therapy

Constants:

- NAME = Brain Therapy.
- NBDAB = 4.
- NBADB = 16.
- NBTXPADB = 8.
- NBRXPADB = 4.
- NBTXCHAN = 512.
- NBRXCHAN = 256.
- CLOCKFREQ = 180.
- ADFILTER = [1, 2, 3, 4].
- ADRATE = [3, 4, 5, 6, 7, 8, 9].
- MINNNOOP = 5.
- MAXNSAMPLES = 400.

7.2 The *system.probe* class

7.2.1 SYSTEM.PROBE

Class containing probe constants.

Constants:

- NAME corresponds to the name of probe.
- TYPE corresponds to the type of probe.
- NBELEMTS corresponds to the number of elements.
- PITCH corresponds to size of the pitch (mm).
- WIDTH corresponds to width between 2 elements (mm).
- RADIUS corresponds to curvature radius (mm).

7.2.2 VC-192

Constants:

- NAME = VC-192.
- TYPE = curved.
- NBELEMTS = 192.
- PITCH = 0.3.
- WIDTH = 0.3.
- RADIUS = 60.

7.2.3 VL-256

Constants:

- NAME = VL-256.

- TYPE = linear.
- NBELEMTS = 256.
- PITCH = 0.2.
- WIDTH = 0.2.
- RADIUS = Inf.

7.2.4 Stanford

Constants:

- NAME = Stanford.
- TYPE = custom.
- NBELEMTS = 60.
- PITCH = 0.2.
- WIDTH = 0.2.
- RADIUS = Inf.

8 Tutorials

8.1 A flat elusev class

9 Appendixes

9.1 Optimization of the number of events for HIFU sequences

9.1.1 Preliminary actions

9.1.2 Event duration greater than 5 ms

9.1.3 Event duration shorter than 2 ms

10 List of figures, tables and codes

10.1 List of figures

Fig. 1 – Hierarchy of generic objects to define an ultrasound sequence. 14

Fig. 2 – Structure and hierarchy of the *common* package. 15

Fig. 3 – Structure and hierarchy of the *remote* package. 25
Fig. 4 – TGC curve built out of the *Duration* and *ControlPts* parameters. 41
Fig. 5 – Structure and hierarchy of the *elusev* package. 59
Fig. 6 – Structure and hierarchy of the *acmo* package. 82
Fig. 7 – Structure and hierarchy of the *usse* package. 91

10.2 List of tables

Tab. 2.1 – List of variables of the *constants* class. 15
Tab. 2.2 – List of variables of the *legHAL* class. 16
Tab. 2.3 – List of the variables of the *parameter* class. 17
Tab. 2.4 – List of the methods of the *parameter* class. 17
Tab. 2.5 – List of the variables of the *object* class. 20
Tab. 2.6 – List of the methods of the *object* class. 20
Tab. 2.7 – List of the methods of the *remoteobj* class. 23
Tab. 3.1 – List of the parameters of the *dmacontrol* class. 26
Tab. 3.2 – List of the parameters of the *event* class. 28
Tab. 3.3 – List of the parameters of the *fc* class. 31
Tab. 3.4 – List of the parameters of the *hvmux* class. 32
Tab. 3.5 – List of the parameters of the *mode* class. 34
Tab. 3.6 – List of the parameters of the *rx* class. 37
Tab. 3.7 – List of the parameters of the *tgc* class. 39
Tab. 3.8 – List of the parameters of the *tpc* class. 42
Tab. 3.9 – List of the parameters of the *tw* class. 44
Tab. 3.10 – List of the parameters of the *tw_arbitrary* class. 47
Tab. 3.11 – List of the parameters of the *tw_pulse* class. 49
Tab. 3.12 – List of the parameters of the *tx* class. 51
Tab. 3.13 – List of the parameters of the *tx_flat* class. 55
Tab. 3.14 – List of the parameters of the *tx_focus* class. 57
Tab. 4.1 – List of the parameters of the *elusev* class. 60
Tab. 4.2 – List of the parameters of the *arbitrary* class. 63

10.3 List of codes

Code 1 – Example of a class declaration within the *Classname.m* m-file. 12
Code 2 – Example of superclass and subclass declarations. 13