EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VLT PROGRAMME

VERY LARGE TELESCOPE

LCU Common Software

LCU Server Framework
User Manual

Doc. No.: VLT-MAN-ESO-17210-2252

Issue: 1.0

Date: 2000-10-25

Prepared: P.Duhoux	2000-10-25	
Name	Date	Signature
Approved: B. Gustafsson		
Name	Date	Signature
Released: G. Raffi		
Name	Date	Signature

CHANGE RECORD

ISSUE	DATE	SECTION/PAGE AFFECTED	REASON/INITIATION DOCUMENTS/REMARKS
1.0	2000-10-25	All	First preparation

TABLE OF CONTENTS

1 Introduction

1.1 Purpose & Scope

This document describes the usage of the LCU Server Framework `lsf` from a programmatic point of view. It is intended to provide all the necessary information for an application developer to create, configure, customize and install LCU Server applications. It describes all the necessary steps that must be performed for the implementation of a new Software Device.
The intended readers are application developers of VLT Software who need to develop, test or maintain LCU Server Applications, based on this framework.

The author assumes the reader has a good knowledge of the VLT Standards and Software Engineering practices, especially experience with Real-Time and Object Oriented concepts and UML Notation would be an asset.

Beside the Introduction, the document is built in 3 parts:

- the **User Manual** describes the concepts and usage of the LCU Server Framework
- the **Reference** section provides the necessary man-pages for all the available scripts, commands and code
- the **Installation Guide** explains how to install a new LCU Server Application in the VLT environments

---

Last modified: Thu Oct 26 12:50:45 METDST 2000

1.2 Applicable Documents

The following documents, of the exact issue shown, form part of this document to the extend specified herein. In the event of conflict between the documents referenced herein and the content of this issue, the content of this document shall be considered as a superseding requirement.

Reference	Document Number	Issue	Date	Title
[AD 01]	VLT-SPE-ESO-10000-0011	2.0	30/09/1992	VLT Software Requirements Specifications
[AD 02]	VLT-PRO-ESO-10000-0228	1.0	10/03/1993	VLT Software Programming Standards
[AD 05]	VLT-MAN-ESO-17200-0888	1.0	17/08/1995	VLT Common Software Overview
[AD 06]	VLT-MAN-ESO-17210-0642	1.11	22/11/1999	VLT Common Software Installation Manual
[AD 07]	VLT-MAN-ESO-17210-0855	3.0	30/10/1998	VLT Software Environments Common Configuration User Manual
[AD 08]	VLT-MAN-ESO-17210-0667	1.0	03/12/1996	Guidelines for the Development of VLT Application Software
[AD 09]	VLT-SPE-ESO-17210-2051	1.0	2000-03-10	LCU Common Software - LCU Server Framework Detailed Design

1.3 Reference Documents

The reference documents contain background information required to fully understand the structure of this document, the terminology used, the software environment in which the product shall be integrated and the interface characteristics to the external systems.
VLT Project Software documents are available from the ESO VLT Project Archive or online at the following URL:
http://www.eso.org/projects/vlt/sw-dev/ftp/index.html

Reference	Document Number	Issue	Date	Title
[RDV 01]	VLT-MAN-ESO-17210-0619	1.6	20/10/1999	VLT Central Control Software - User Manual
[RDV 02]	VLT-MAN-SBI-17210-0001	3.5	20/10/1999	LCU Common Software - User Manual
[RDV 03]	VLT-MAN-ESO-17210-0707	1.4	20/10/1999	CCS On-Line Database Loader - User Manual
[RDV 04]	VLT-SPE-ESO-17230-0819	1.3	18/09/1997	Mode Switching - Design Description
[RDV 05]	VLT-MAN-ESO-17210-0600	1.7	02/10/1998	LCC Motor Control Module - User Manual

The following non-VLT documents are referenced in the document:

Reference	Title	Authors	Publisher, Date
[RD 01]	IEEE Std 1012, Standard for Software Verification and Validation Plans		1986
[RD 02]	IEEE Std 830, Recommended Practice for Software Requirements Specifications		1993
[RD 03]	IEE (UK), Guidelines for the Documentation of Software in Industrial Computer Systems		1985
[RD 04]	The Unified Modeling Language Reference Manual	J. Rumbaugh, G. Booch, I. Jacobson	Addison-Wesley, 1998
[RD 05]	The Unified Modeling Language User Guide	G. Booch, J. Rumbaugh, I. Jacobson	Addison-Wesley, 1998
[RD 06]	The Unified Software Development Process	I. Jacobson, G. Booch, J. Rumbaugh	Addison-Wesley, 1999
[RD 07]	Real-Time UML - Developing Efficient Objects For Embedded Systems	B. Douglas	Addison-Wesley, 1998
[RD 08]	Design Patterns - Elements of Reusable Object-Oriented Software	E. Gamma, R. Helm, R. Johnson and J. Vlissides	Addison-Wesley, 1994

1.4 Acronyms

This document employs several abbreviations and acronyms to refer concisely to an item, after it has been introduced. The following list is aimed to help the reader in recalling the extended meaning of each short expression:

[AD *xx*]	Applicable Document #*xx*
CCS	Central Common Software
DB	Database
ESO	European Southern Observatory
I/O	Input/Output
LAN	Local Area Network
LCC	LCU Common Software
LCU	Local Control Unit
MCM	Motor Control Module (part of LCC)
OLDB	On-line Database
OO	Object Oriented
[RD *xx*]	Reference Document #*xx*
RT	Real-Time
TIM	Time Interface Module
UC	Use Case
UML	Unified Modeling Language
VME	Versa Module Eurocard
WS	(Unix) Work Station

1.5 Stylistic Conventions

1.5.1 Text

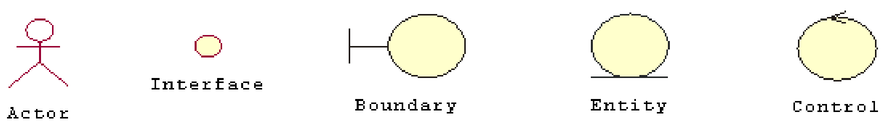
The following styles are used:

- **bold** : in the text, for commands, filenames, pre- and suffixes as they have to be typed.
- *italic* : in the text, for parts that have to be substituted with the real content before typing.
- `teletype` : for examples.
- <name> : in the examples, for parts that have to be substituted with the real content before typing.

1.5.2 Diagrams

All diagrams are based on the UML notation, as described in [RD 04] and given in Appendix.
Besides the classical class notation, three dedicated icons are used throughout the class diagrams for representing the UML standard class stereotypes ([RD 06] pp. 183ff) and shown below:

- <<boundary>>: model interactions between the system and its actors.
- <<entity>>: model information and associated behavior of phenomenon, concept such as a real-life objects; often persistent.
- <<control>>: represent coordination, sequencing, transactions and control of other objects; often used to encapsulate the control related to a use-case.



2 User Manual

2.1 Introduction

In this chapter, the concept of the LCU Server Framework will be described. This section is followed by the practical description of its usage through the whole creation and customization process of a new LCU Server Application.
Naming Conventions: In this chapter, the following naming conventions apply: the generic names *`devtype`* and *`Devtype`* differ only by the case of the first letter (lower/upper). The first case indicates the name of the device type, the latter is the name of the device type when concatenated with the module prefix as in `appDevtype`.

2.2 LCU Server Framework

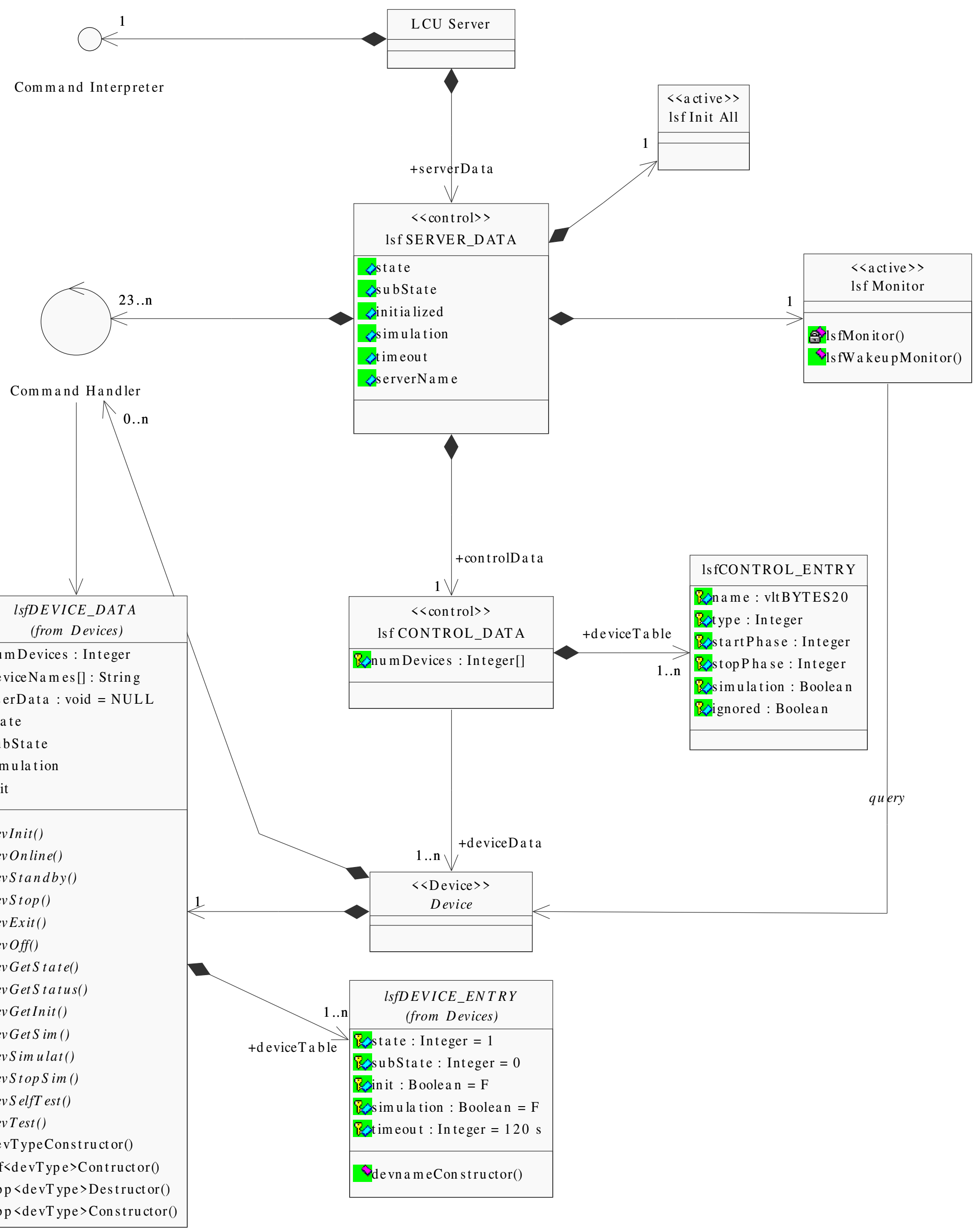
The intent, motivation and consequences of the framework have been discussed in detail in [AD 09]. The framework implements the minimum behavior for all the Standard Commands and offers the expected extensibility, flexibility and changeability while defining a fixed architecture and improving the overall maintainability as most of the code is provided and reused.

2.2.1 lsf: the LCU Server Framework

The framework consists in a collection of reusable components that implement the standard behaviour for a number of devices. It provides a unique and fully re-entrant set of common procedures that need not be duplicated by new applications.
As of this issue, `lsf` supports the following devices:

- I/O signals: digital and analog signals based resp. on the VLT standard Acromag AVME 948x Digital I/O and VMIVME-3111 Analog I/O VME boards
- motors: any kind of motor as supported by MCM
- serial communication links: based on the VxWorks `tyCo` driver and on the VLT standard esd ISER8 Serial Interface Board.
- tasks: any periodic task running in background as part of the Software Device
- software devices: any `lsf`-based sub-Software Device running on the same LCU.

Architecture: The purpose of a framework is to impose the architecture of the applications based on and to provide the implementation of the standard behaviour. The application is consequently responsible for implementing the specialized functionality as of its design.
The architecture is hierarchical: The server object `lsfServer` is responsible for handling the application global data and hosts the commands of the application; these commands might involve no device (pure software command) or devices of various types (combined command). The actual control of the devices is performed from within the object `lsfControl` that dispatches the standard commands to the respective devices. Finally for each device type, an object `lsf`*`DeviceType`* implements the commands of this type of device.

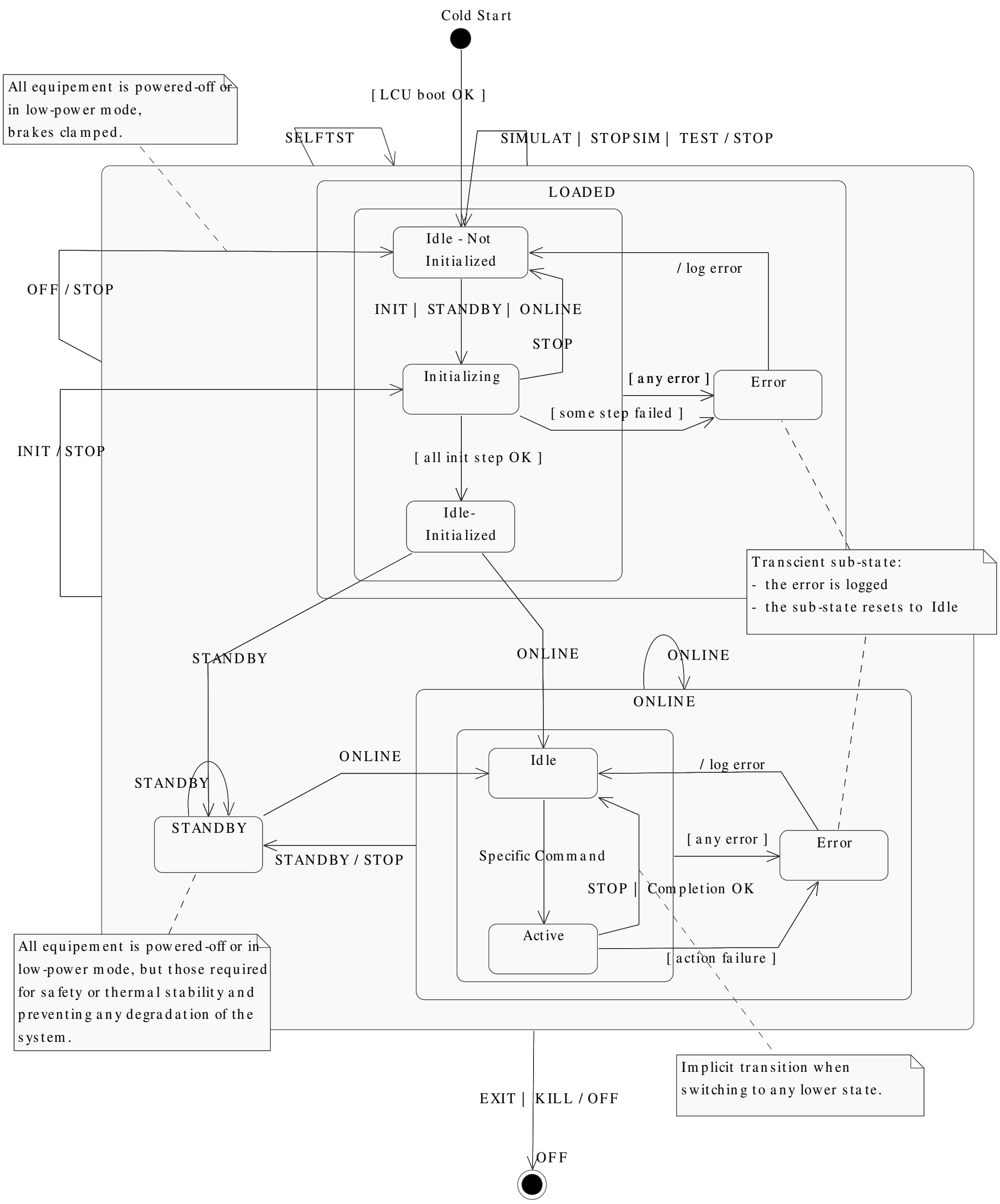


Two additional objects complete this view: the task `lsfInitAll` that creates the application at boot time, and the task `lsfMonitor`, created at boot time, that monitors periodically the state and sub-state of all the devices and updates consequently the state and sub-state of the application.

Devices: Each device is characterized by its type as listed above. All devices of one type are supposed to behave in a similar manner in response to the standard commands. Consequently all the methods associated to one device type perform the same actions on all the devices of this type.

Standard Behaviour: The standard commands are implemented within the framework. They define the standard behaviour of each standard command for each type of device. The standard behaviour is edicted in [RDV 02] for Software Devices and State management. The standard behaviour can not be modified but only extended by means of dedicated hooks functions, which are application specific and implement additional actions required by the application design (e.g. set a particular I/O signal prior to go Online).

States and Sub-States: Each device is characterized by its state and sub-state. The state is defined by LCC ([RDV 02]) and takes the values `OFF`, `LOADED`, `STANDBY` or `ONLINE`.
Each group of devices of a type is as well characterized by a state and sub-state value; the state is set as being the lowest state of all the devices of this group.
The global application state is then the lowest state value of all the devices.



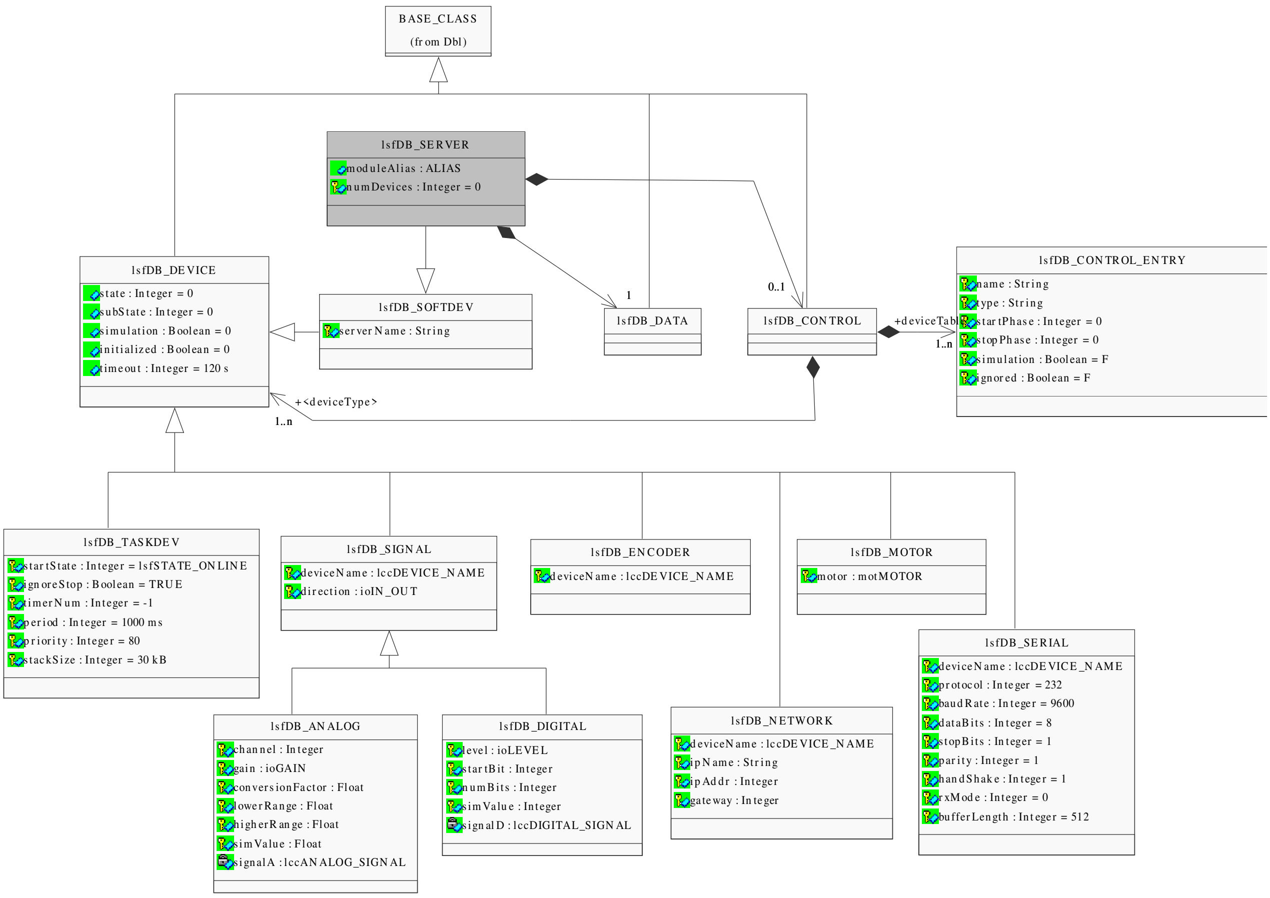
The sub-state is used to indicate the detailed activity of a device, or group of devices; the following sub-states have been introduced:

- `Idle`: the device is inactive
- `Error`: an error has occured
- `Timeout`: the requested action has timed out
- `Initializing`: the device is performing the initialization procedure
- `Monitoring`: monitoring is active in background
- `Moving`: for motors only when performing a motion
- `Active`: any other activity like e.g. Waiting for an event, in a Control Loop etc...

The 2 sub-states `Error` and `Timeout` are transcient states that reset to `Idle` on recovery, or on next request.
The global sub-state of the software device is set to:

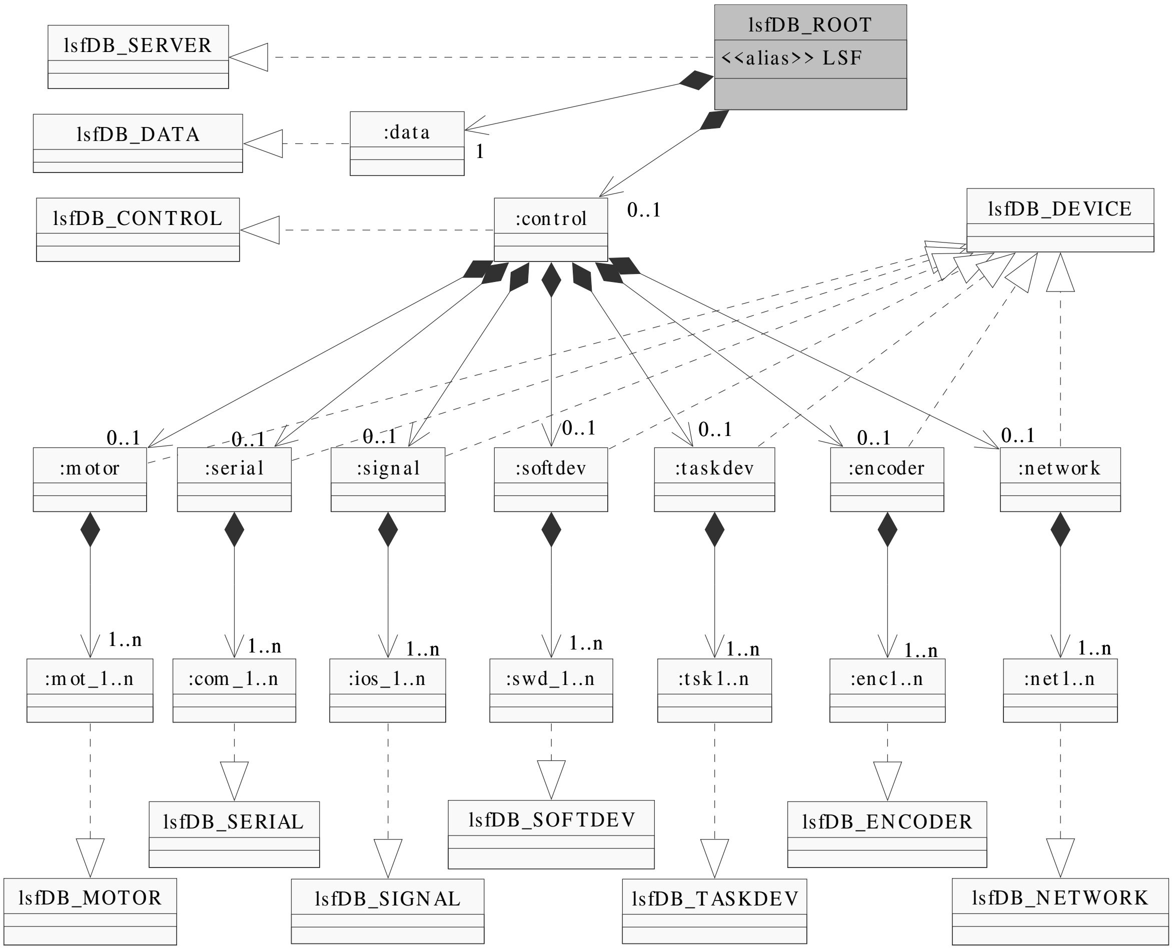
- `Idle`: when all the devices are inactive
- `Error`: if one device went in error.
- `Timeout`: if one device timed out.
- `Initializing`: if one device is initializing
- `Active`: if one device or more is active

Database: All relevant configuration and runtime data is stored in the OLDB of the LCU, which is partly mirrored on the WS so as to enable the use of the scan system.
Each type of device is mapped in the database by a dedicated class. All the device classes are derived from the base abstract class `lsfDB_DEVICE` as depicted in the class hierarchy:



Note that a Software Device is a sub-class of `lsfDB_SOFTDEV`, thus allowing recursive construction: a Software Device can be a sub-Software Device of another one.

The structure of the database is fixed as illustrated in the diagram below:



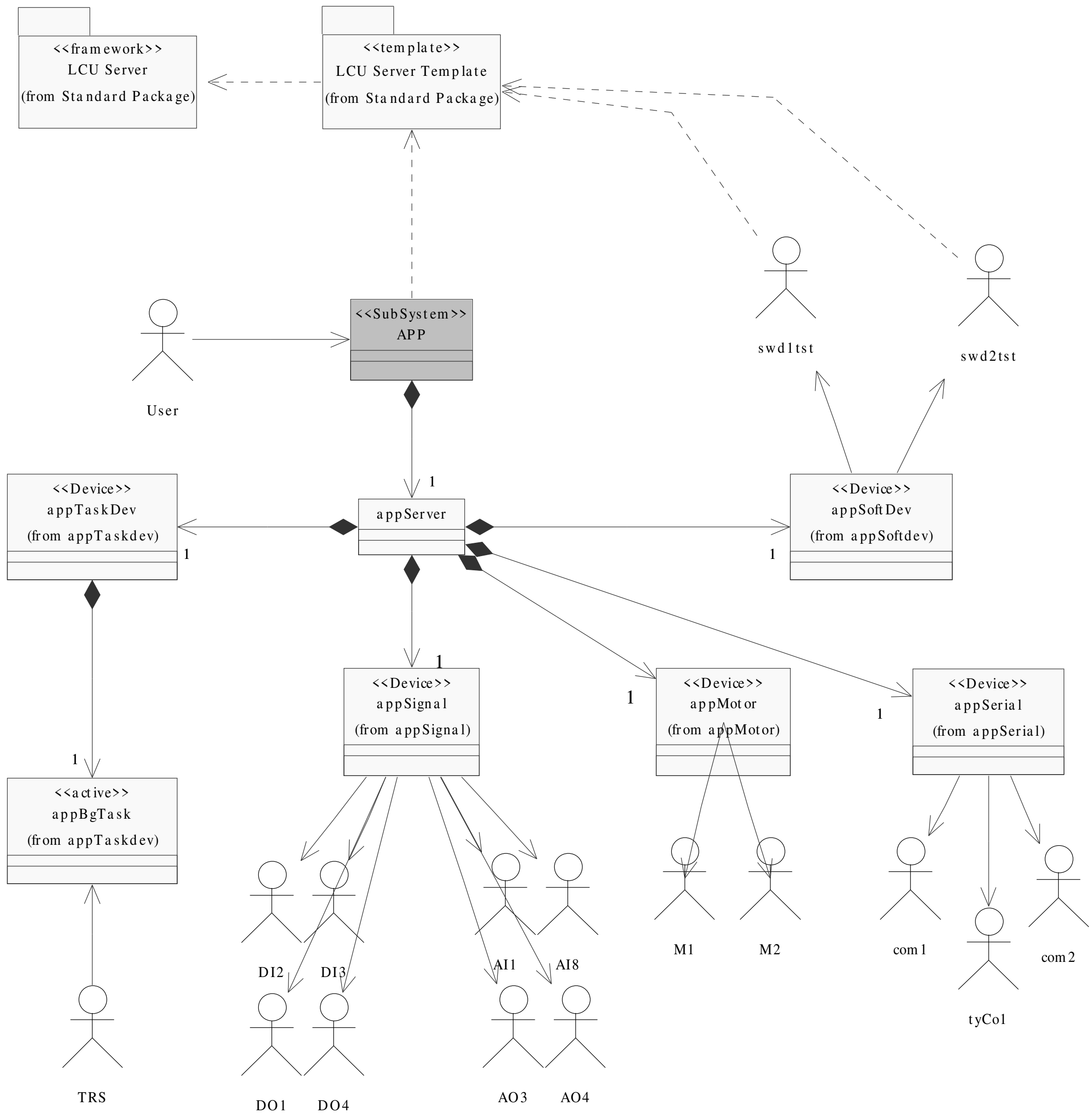
Note the multiplicity of the various instances as a means to scale the database to the exact needs of the application.

Application Data: The class `lsfDB_DATA` is an abstract class. It does not contain any attribute as it is intended to be specialized (by sub-classing) to the needs of the application. This class has been introduced so as to allow the application to store data that shows following characteristics at a known location:

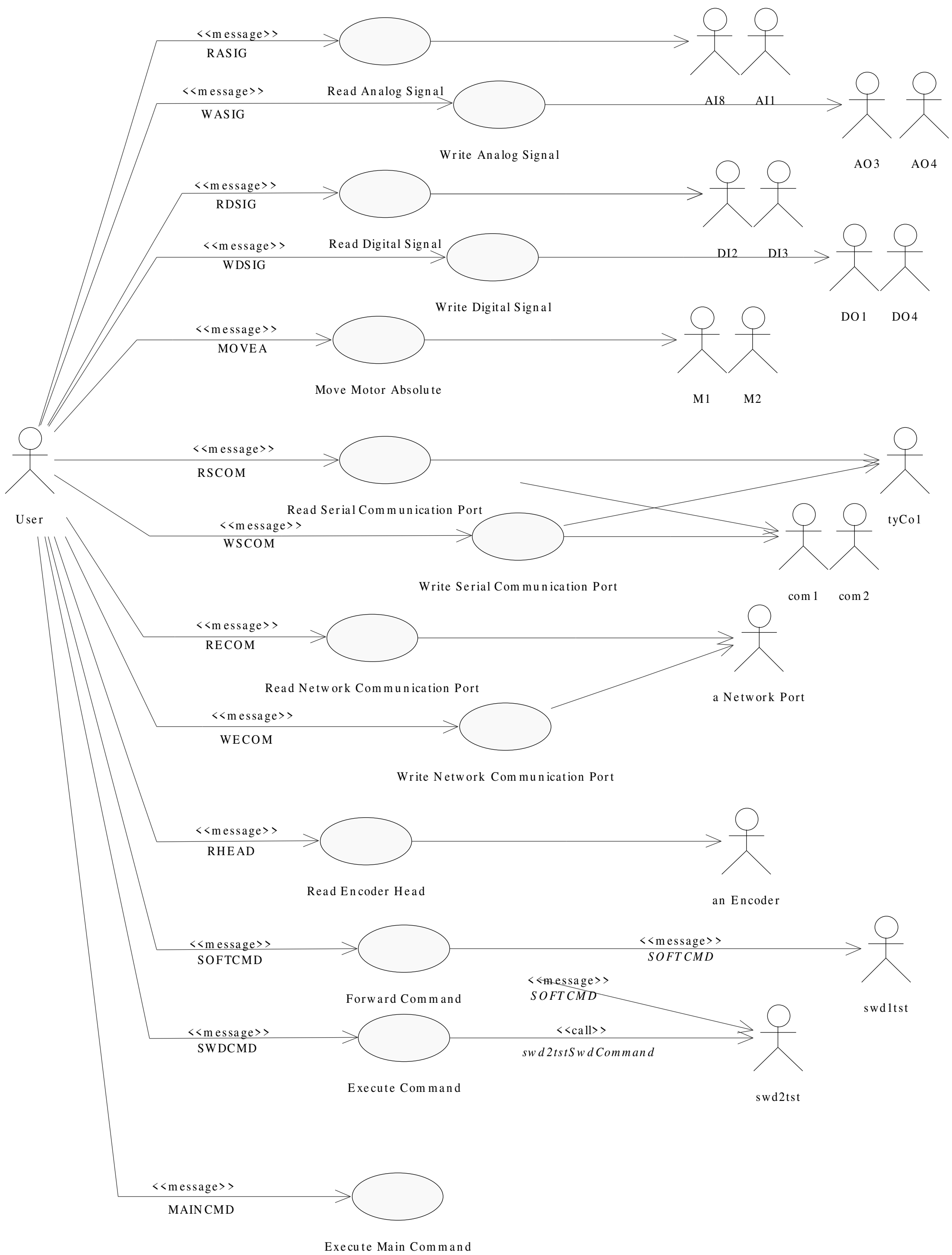
- the data is not strictly device dependent (e.g. temperatures, or telemetry values), or shall be hardware independent (i.e. should a thermometer device be exchanged from a device with digital interface to a one with serial interface)
- the data might be accessed from within another device (e.g. a background task) for further processing
- the data might need to be scanned to the WS independently on the devices producing it.

2.2.2 lsftpl : the LCU Server Application Template

The module `lsftpl` is a template application based on `lsf` that provides an implementation template for all available device types. In particular, it gives examples of ACI and API functions for signals and motors for addressing one or all devices of this type.
The template implements the control of 16 devices: 8 I/O signals (4 analogue and 4 digital where from 2 input and 2 output signals), 2 motors, 3 serial communication links (2 using the ISER 8 board, and one `/tyCo` device), 2 sub-ordinated software devices (`swdtst1` and `swdtst2`) and 1 task device performing periodic monitoring) as illustrated in the class diagram below:



It implements the functionality as shown in the Use Case diagram:



<u>N.B.:</u> as of this version, both the encoder and network devices are not yet supported.
The following section describes all the steps necessary to create a new application from this template.

2.3 LCU Server Application

This section describes how to create a LCU Server Application based on the Framework.
It assumes the detailed analysis of the package is complete: all interacting devices, and commands and interfaces have been identified and described.

2.3.1 Module Creation

The creation of a new LCU Server Application `app` from the template is made by means of the utility `lsfCreate(1)`. It makes a copy of the template module `lsftpl` in its last version available under CMM, and performs all string replacements: all occurences of `lsftpl` are substituted by `app`, as well the occurences of `LSFTPL` are substituted by `APP`.

Example:

```
lsfCreate app
Retrieving module 'lsftpl' ... done
Renaming module 'lsftpl' to 'app' ... done
Renaming all files with prefix 'app' ... done
Replacing pattern 'lsftpl' by 'app' ... done
1.24u 49.41s 1:59.60 42.3%
```

The module `app` is ready. It needs now be configured for the devices and commands.

2.3.2 Module Configuration

The configuration of the application is kept central in the file `app/ws/config/app.cfg`. This file must be edited to the needs of the application. The module configuration contains 3 parts:

- module:
  LSF.MODULE.NAME app
- devices:
  LSF.DEVICE.NUM number of devices *D*
  For each device *d* (from 1 to *D*)
  LSF.DEVICE*d*.NAME: name of the device (19 char. max), unique for the application.
  LSF.DEVICE*d*.TYPE: type of the device:
    - **analog**, **digital**: I/O signal
    - **motor**: motorized axis
    - **serial**: communication link
    - **taskdev**: task
    - **softdev**: sub-Software Device (another LSF-based application to which this application will communicate via the Message Server)

  LSF.DEVICE*d*.CLASS: when applicable the name of the specialized database class appDB_*devtype* as derived from the corresponding generic device class lsfDB_*devtype*. For the motors, the motor specialized class name (motDVAMI etc... see [RDV 05])
  LSF.DEVICE*d*.PARAM: for motors only the dimensions of database tables for the named positions (P), named speeds (V) and initialization sequence (I), following the syntax: `P` *p*`,V` *v*`,I` *i*.

- commands:
  LSF.COMMAND.NUM number of commands *C*
  For each command *c* (from 1 to *C*)
  LSF.COMMAND*c*.NAME: name of the command following the naming recommendations stated in [RDV 02]
  LSF.COMMAND*c*.DEST: device type for which the command is intended (see the list of device types above); in addition, the destination may be set to `server` for any command which either does not involve any device, or commands that involve many devices of various types
  LSF.COMMAND*c*.GROUP: as described in [RDV 02], the associated command group (**PUBLIC**, **MAINTENANCE** or **TEST**)
  LSF.COMMAND*c*.ENTRY: the name of the entry function associated to the command and to be declared in the Command Interpreter Table (CIT). The ACI function name must follow the convention: Iapp<*Devtype*><*Entry*>, where *Devtype* is the device type as declared under LSF.COMMAND*c*.DEST and *Entry* is the name entered under this key-word
  LSF.COMMAND*c*.INVOKE: the invocation mode **FUNCTION** or **TASK**
  LSF.COMMAND*c*.OPTION: for the invocation mode **TASK** only, the task options following the syntax: <*taskName*>,[<*priority*>], [<*task flags*>],[<*stack size*>][,<*options*>].
  where the task name shall be built as tapp<*command name*>
  Note: The option REGISTER must be omitted since it is handled internally (all tasks are registered to allow CCS communications)

Example:

```
#*****************************************************************************
# E.S.O. - VLT project
#
# "@(#) $Id: app.cfg,v 3.4 2000/07/21 09:22:08 vltsccm Exp $"
#
# who       when        what
# --------  ----------  ----------------------------------------------
# pduhoux   2000-04-07  created
#

#
# MODULE
#
LSF.MODULE.NAME     app

#
# DEVICES
#
LSF.DEVICE.NUM      1

LSF.DEVICE1.NAME    NDF
LSF.DEVICE1.TYPE    motor
LSF.DEVICE1.CLASS   motDVAMI
LSF.DEVICE1.PARAM   P 7,V 3,I 6

#
# COMMANDS
#
LSF.COMMAND.NUM       1

LSF.COMMAND1.NAME     SETNDF
LSF.COMMAND1.DEST     motor
LSF.COMMAND1.GROUP    PUBLIC
LSF.COMMAND1.ENTRY    SetFilter
LSF.COMMAND1.INVOKE   TASK
LSF.COMMAND1.OPTION   appSETNDF,,,40000

# ___oOo___
```

The effective configuration of the LCU Server Application `app` is made by means of the utility `lsfConfig(1)`.
It performs following steps:

1. Parse the configuration file `app/ws/config/app.cfg` and display all the devices by type and the commands by group
2. Generate the include file `app/lcu/include/appDeviceList.h` containing the macro definition of all devices
3. Generate the database file `app/lcu/dbl/app.db` for the instantiation of the database point.
4. Generate the makefile `app/lcu/src/Makefile`
5. Generate the CDT partial files `app/lcu/src/appPublic.cdt`, `app/lcu/src/appMaintenance.cdt` and `app/lcu/src/appTest.cdt`. The files are created only if at least one command has been declared in this group.
6. Generate the CIT partial files `app/lcu/src/app<Devtype>.cit` for each referenced device type `Devtype`.
7. Move all the unused files to `app/lcu/tmp`
   (The unused files are the files associated to the control of the device types not referenced in the configuration file. They are moved to this directory prior to be removed by the user when preparing for archive.)
8. Generate the log file `app/ChangeLog` and add an entry for the configuration sequence.

Important Note: The utility `lsfConfig` shall be called once as its actions are destructive for some files. Any application dependent modification made to the partial CDT files will be lost during this process, as these files are re-created. A posteriori addition of a device of a new type is not supported.

Example:

```
> lsfConfig app
Parse file 'app/ws/config/app.cfg' ... done
Found 1 device:
     1 Motor    : NDF
Found 1 PUBLIC command:
     1 Motor    : SETNDF
Found 0 MAINTENANCE command
Found 0 TEST command
Generate 'app/ws/config/app.dbcfg' ... done
Generate 'app/lcu/include/appDeviceList.h' ... done
Generate 'app/lcu/dbl/app.db' ... done
Generate 'app/lcu/src/Makefile' ... done
Generate 'app/lcu/src/appPublic.cdt' ... done
Generate 'app/lcu/src/appMotor.cit' ... done
Move unused files to ./lcu/tmp ... done
Add change log entry ... done

>>> Remember to edit the CDT files ...

Module configuration for 'app' ... done"
>
```

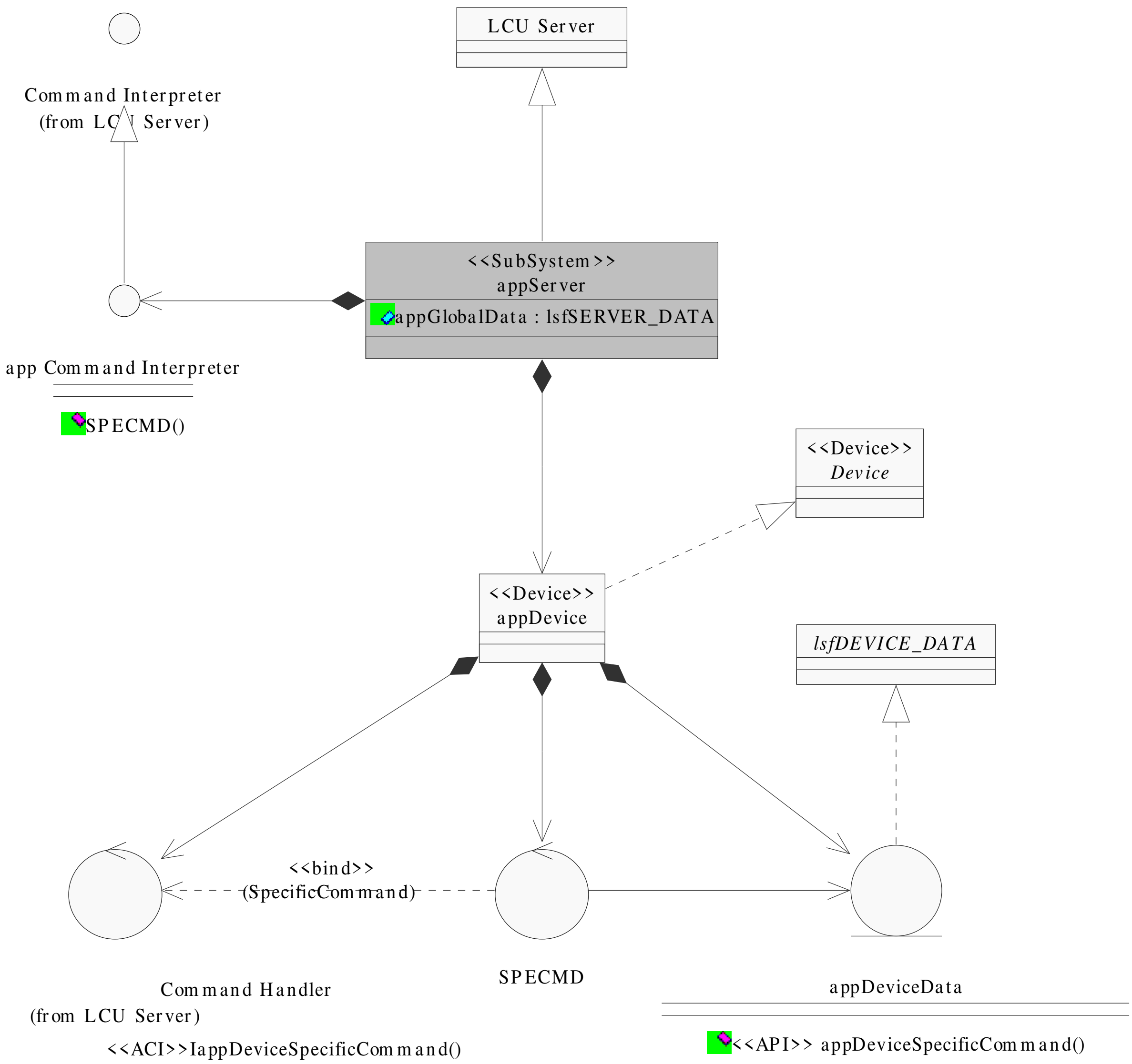
2.3.3 Specific Implementation

After the module has been tailored to the needs of the application, some files need be edited to implement the behaviour and specify the interfaces as of the design.

- `app/lcu/src/appPublic.cdt`,
  `app/lcu/src/appMaintenance.cdt`,
  `app/lcu/src/appTest.cdt`: For each command the parameters and replies must be described.
- `app/lcu/dbl/appDB_DEVICE.class`: Each application specific device class must be defined as a sub-class of the corresponding device type `lsfDB_`*`DEVTYPE`*
- `app/lcu/dbl/appDB_DATA.class`: If any extra data storage is needed in the database by the application, the class `appDB_DATA` must be described in this file
- `app/lcu/include/app`<*Devtype*>`.h`: for each referenced device type, the prototypes of the API functions associated to the commands must be declared. The file `app/lcu/include/appServer.h` is also subject to modification.
  The API function shall be named following the convention for the ACI function names without the leading `I`.
- `app/lcu/src/app`<*Devtype*>`.c`: Each API function must be implemented in the respective file.
- `app/lcu/src/app`<*Devtype*>`Interface.c`: Each ACI function must be implemented in the respective file.

For the sake of clarity, one might group the API functions dedicated to one specific device into one separate file. These files may be included into the device type associated file.

The class diagram shown below depicts the device control structure for a specific command:



2.3.4 Scan Links

For application implementing the WS counterpart of the database, the scan links must be configured. These links are described in the file `app/ws/config/app.scan`.

2.3.5 Database Configuration

At boot-time, the database branch of the module need be configured. The configuration is kept in the file `app/ws/config/app.dbcfg` and restored from `$VLTROOT/config`, resp. from `$INTROOT/config` at startup. This file is mandatory for starting up the server, however the database is then solely basically configured and might not permit to initialize the system. By means of the dedicated engineering tools, the database can be properly configured. Once the initialization is successful, the utility `lsfBackup(1)` creates a file `./app.dbcfg` that contains the relevant database configuration.

<u>Example:</u>

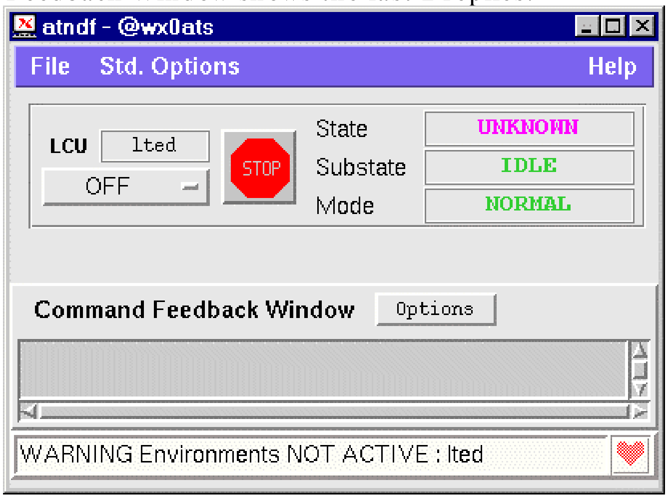
```
> cd app/ws/config
> lsfBackup -e lcuEnv -m app
Generating input file './app.inp' ... done
Performing database backup into  './app.dbcfg' ... done
```

```
>
```

The file `./app.inp` is a temporary file that contains the list of database points and attributes to backup. At backup completion, it is removed.

2.3.6 Default panel & UIF widget

A default panel `appgui` is provided by the template that instantiates the UIF widget `lsfState_uifClass`. This widget shows the associated LCU environment name, a button for sending standard commands, a STOP button, and the status information of the module (state, subState and operational mode).
In addition, the Command Feedback Window shows the last 2 replies.



2.3.7 Application Makefiles

The WS part is built as described in `app/ws/src/Makefile`. It makes the panel `app/ws/src/appgui.pan` and installs the two configuration files `app/ws/config/app.scan` and `app/ws/config/app.dbcfg` under `$XXXROOT/config`.

Example:

```
> cd app/ws/src
> make all install
 ...
 . . . 'all' done
 ...
 . . . installation done
>
```

The LCU part is built as described in `app/lcu/src/Makefile`.
It invokes 3 utilities prior to building the code (target "all"):

- `lsfMakeCIT`: merge all the partial CIT files `app/lcu/src/app*.cit` and the common CIT file `$XXXROOT/vw/CIT/lsfCommon.cit` into `app/lcu/CIT/appServer.cit`;
- `lsfMakeCDT`: merge all the partial CDT files `app/lcu/src/app*.cdt` and the common CDT files `$XXXROOT/CDT/lsf*.cdt` into `app/lcu/CDT/appServer.cdt`;
- `lsfMakeINC`: generate all the files `app/lcu/include/app*Interface.h` from the partial CIT files `app/lcu/src/app*.cit`.

Example:

```
> cd app/lcu/src
> make all
 . . . 'CIT' done
 . . . 'CDT' done
Include file '../include/appMotorInterface.h' done
 . . . 'include files' done
 ...
== Building executable: ../bin/app
```

```
 . . . 'all' done
>
```

Note: The final CDT and CIT files are **deleted** by the Makefile. Remember to perform the modifications in the `./src` directory.

2.3.8 Implementation Rules for the Command Handlers

As stated in the rules applying to a Software Device, the behaviour of each command is implemented in a Command Handler. This handler must show the 2 ACI and API interfaces:

- **ACI function:** the ACI function name is built on the following scheme:

  ```
  ccsCOMPL_STAT Iapp<Devtype><Entry>
                               ( ... , ccsERROR *error )
  ```

  It is responsible for parsing the command parameters, invoking the API function and building the reply buffer as defined in the CDT entry of this command.
- **API function:** the API function name is built on the following scheme:

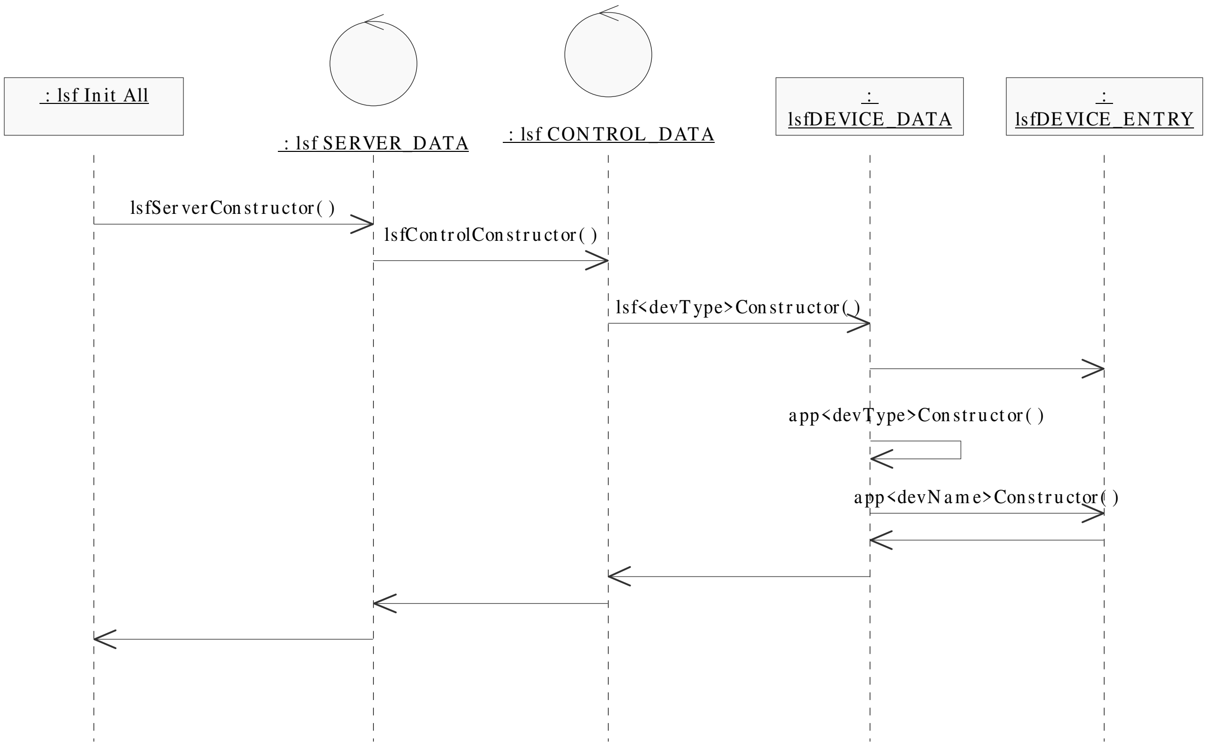
  ```
  ccsCOMPL_STAT app<Devtype><Entry>
                               ( ... , ccsERROR *error )
  ```

  It is the core implementation of the behaviour of the command. **It must be callable from another application.** It receives application specific parameters and the error stack pointer, and returns the completion status.

2.3.9 Hook functions

A hook function is implementing an additional behaviour. Four kinds of hooks are supported:

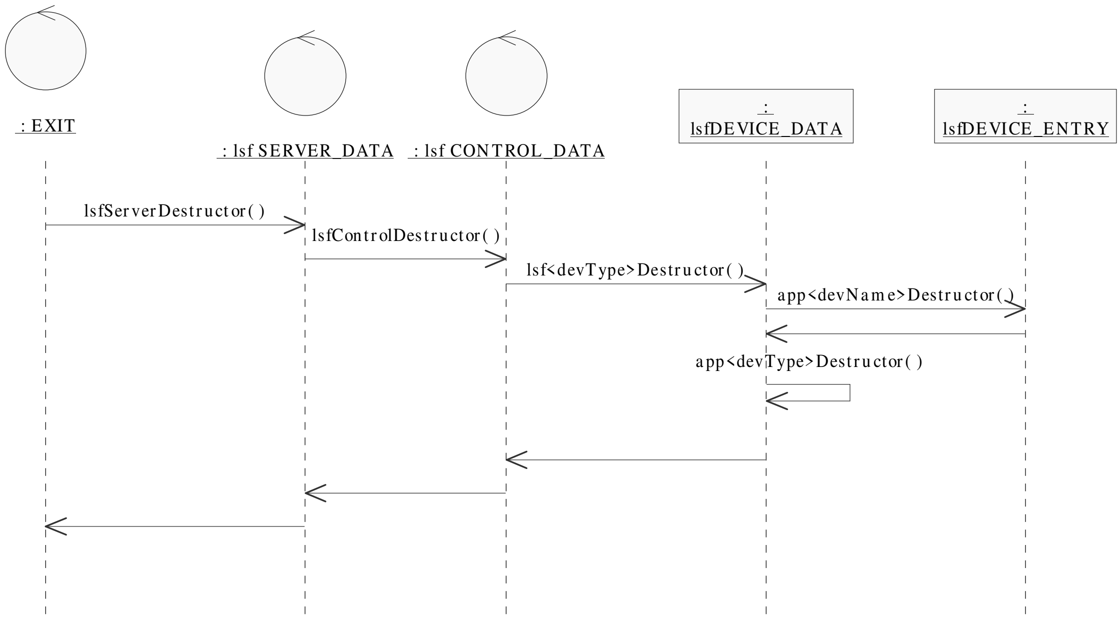
- **Constructors/Destructors:** One application specific constructor and one application specific destructor per device name, resp. device type may be implemented. These functions will be invoked according to the following scheme:
  - At boot time, the object constructors will be invoked as depicted in the following Sequence Diagram.



The application specific constructors have the prototype:

```
ccsCOMPL_STAT app<devType>Constructor ( IN  void       *devData,
                                        IN  const char *devName,
                                        OUT ccsERROR   *error )
ccsCOMPL_STAT app<devName>Constructor ( IN  void       *devData,
                                        IN  const char *devName,
                                        OUT ccsERROR   *error )
```

- At exit time, the object destructors will be invoked as depicted in the following Sequence Diagram.



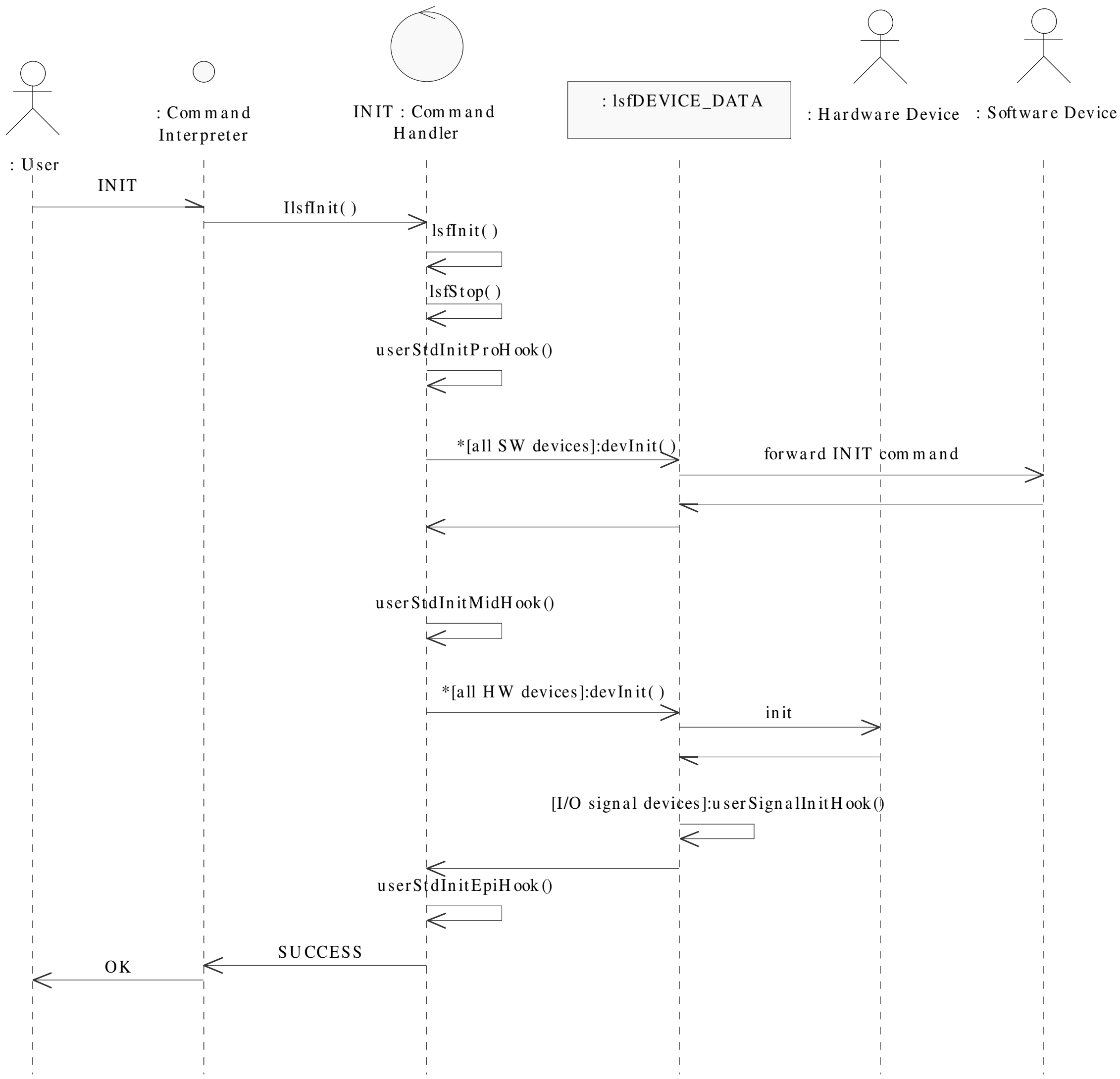
The application specific destructors have the prototype:

```
ccsCOMPL_STAT app<devType>Destructor  ( IN  void *devData )
ccsCOMPL_STAT app<devName>Destructor  ( IN  void *devData )
```

- **Standard Commands:** 3 hooks per standard command may be implemented. The hook function names must follow the prototype convention:

```
ccsCOMPL_STAT appStd<Command><Type>Hook ( IN  lsfSERVER_DATA *serverData,
                                          OUT ccsERROR       *error )
```

where `Command` is the name of the command (`Init`, `Standby`, `Online`, `Stop`, `Off`, `Exit`, `Selftest`, `Test`),
and `Type` is the hook type (`Pro`, `Mid`, `Epi`) resp. for the prologue, the intermediate and the epilogue additional statements. The intermediate hook is invoked only together with at least one sub-Software Device and one Hardware Device.
The hooks are located in the file `appStdHooks.c`. They do not need any prototype declaration.



- **Signals Initialization hook:** a dedicated hook might be implemented that allows the proper setting of the signals at initialization, thus setting the sub-system in a known state. The hook function has the following prototype:

```
ccsCOMPL_STAT appSignalInitHook ( IN  lsfSIGNAL_DATA *signalData,
                                  IN  const char     *signalName,
                                  OUT ccsERROR       *error )
```

- **Task device hooks:** One hook function per task device must be implemented, additional 2 hooks are optional.
  The mandatory hook function implements the core of the task and the prototype is:

  ```
  ccsCOMPL_STAT app<TaskDevName> ( IN  lsfTASKDEV_DATA *taskDevData,
                                   IN  const char      *taskDevName,
                                   OUT ccsERROR        *error )
  ```

  where `TaskDevName` is the name of the task device (starting with upper case).
  The 2 additional hooks are invoked whenever implemented before entering the loop, resp. when terminating. The hook functions must be prototyped:

  ```
  ccsCOMPL_STAT app<TaskDevName>ProHook ( IN  lsfTASKDEV_DATA *taskDevData,
                                          IN  const char      *taskDevName,
                                          OUT ccsERROR        *error )
  ```

  resp.

  ```
  ccsCOMPL_STAT app<TaskDevName>EpiHook ( IN  lsfTASKDEV_DATA *taskDevData,
                                          IN  const char      *taskDevName,
                                          OUT ccsERROR        *error )
  ```

2.3.10 Miscellaneous

- **Adding source files:** There is no limitation in adding new source or include files to the module repository. These files must be listed in the makefile either under the list `app_OBJECTS` for a source file, or under `INCLUDES` for an include file to be installed, see also `Makefile(5)`.

Last modified: Thu Oct 26 14:23:30 METDST 2000

3 Reference

In this section, the man-pages of the utilities, objects anf database classes are provided. The public include files are appended.

Utility	Code	Database Class	Include File
lsf(1)		lsfTemplate.db	lsfDefines.h
lsfCreate(1)			
lsfConfig(1)			
lsfBackup(1)			
	lsfServer(3)	lsfDB_SERVER(5)	lsfServer.h
	lsfStandard(3)		
	lsfHandleBreak(3) lsfHandleKill(3)		
	lsfControl(3)	lsfDB_CONTROL(5) lsfDB_DATA(5)	lsfControl.h
	lsfDevice(3)	lsfDB_DEVICE(5)	lsfDevice.h
	lsfSignal(3)	lsfDB_SIGNAL(5)	lsfSignal.h
	lsfMotor(3)	lsfDB_MOTOR(5)	lsfMotor.h
	lsfSerial(3)	lsfDB_SERIAL(5)	lsfSerial.h
	lsfTaskDev(3)	lsfDB_TASKDEV(5)	lsfTaskDev.h
	lsfSoftDev(3)	lsfDB_SOFTDEV(5)	lsfSoftDev.h

Last modified: Fri Aug 11 08:21:18 METDST 2000

3.1 Utilities

3.1.1 lsf(1)

NAME

```
lsf - LCU Server Framework
```

SYNOPSIS

```
> lsfCreate <modName>
> emacs <modName>/ws/config/<modName>.cfg
> lsfConfig <modName>
> emacs <modName>/lcu/src/<modName>Public.cdt
> cd <modName>/ws/src  ; make clean all man install ; cd -
> cd <modName>/lcu/src ; make clean all man install ; cd -
```

DESCRIPTION

```
lsf is a framework for developing LCU Server Applications. It
consists of a WS part dedicated to the generation and
configuration/customization  of the target application module and
of a LCU part providing the application architecture and implementing
the minimum behaviour for the standard commands.

1 - The WS part:
  1.1 Script lsfCreate(1): creates a software module compliant with
      the VLT Standards from a template.
  1.2 Script lsfConfig(1): customizes the software module according
```

```
    to the software device description file.
1.3 The description file: is an ASCII file containing the definition
    of all devices (hardware and software) to be controlled by the
    application and of all the commands to be accepted by the
    application.
    The devices are grouped in 7 types:
    - signal (analog or digital) based on VLT AIO resp. ACRO boards
    - motor (all types as supported by MCM)
    - serial communication link (RS232, RS422 or RS485) based on
      VxWorks tyCo driver or VLT ISER8 board.
    - software device (any lsf-based application to be controlled
      from this software device)
    - tasks (any periodic background process)
    - encoder [not supported yet]
    - network [not supported yet]
    The commands are grouped in 3 types:
    - public for normal operations
    - maintenance for software/hardware verification work
    - test with restricted access to technical staff
1.4 The database is a partial mirror (based on the scan system) of
    the LCU database

2 - The LCU part:
  2.1 Code: the binary code lsf is to be downloaded to the LCU prior
      to loading the application software. It implements the minimum
      behaviour of the supported devices for the standard commands.
  2.2 Database: the database architecture is provided.
      The device classes may be customized by sub-classing
      to the needs of the application.

3 - Creating a new application
  The creation of a new module is made by the utility lsfCreate.
  It receives the name of the module <mod> as argument.
  The module name shall not be registered in the CMM archive.

  Example:
    From the parent directory of <mod>
    > lsfCreate mod
    Retrieving module 'lsftpl' ... done
    Renaming module 'lsftpl' to 'mod' ... done
    Renaming all files with prefix 'mod' ... done
    Replacing pattern 'lsftpl' by 'mod' ... done
    1.24u 49.41s 1:59.60 42.3%
    >

4 - Customizing the software device
  The customizatin of a new module is made by the utility lsfConfig.
  The file <mod>/ws/config/<mod>.cfg must be edited and its content
  modified to the needs of the application <mod>.

  LSF.MODULE.NAME      <mod>

  LSF.DEVICE.NUM       << the number of devices to be controlled >>

  Mandatory for each device <d>:
  LSF.DEVICEd.NAME     << the name of the device,
                          it defines the name of a database point >>
  LSF.DEVICEd.TYPE     << the type of the device:
                          analog, digital: for signals
                          serial: for serial communication links
```

```
                         motor: for motorized axis
                         taskdev: for a periodic background process
                         softdev: for a sub-software device,
                         it defines the type of control, and the
                         associated database class >>
The following 2 entries are optional:
LSF.DEVICEd.CLASS   << the specialized database class name mapping
                       the device. For motors, the motor class
                       name (e.g. motDVAMI) >>
LSF.DEVICEd.PARAM   << for motors only, the number of entries in
                       the named position (P), named speed (V),
                       initialization sequence (I) and unit conversion
                       methods (U), following the syntax:
                       "P <p>,V <v>,I <i>,U <u>" >>

LSF.COMMAND.NUM     << the number of commands to be accepted,
                       in addition to the standard commands >>

Mandatory for each command <c>:
LSF.COMMANDc.NAME   << the name of the command as specified for
                       the CDT (7 char.) >>
LSF.COMMANDc.DEST   << signal, motor, serial, softdev or server
                       where "server" indicates that the command
                       is to be processed internally (no device
                       interaction).
                       NB: task devices do not process commands >>
LSF.COMMANDc.GROUP  PUBLIC, MAINTENANCE or TEST

LSF.COMMANDc.ENTRY  << API Funtion name <Fct> following the rule:
                       <mod><Devtype><Fct>()
                       The ACI function name is built automatically
                       as I<mod><Devtype><Fct>() in the corresponding
                       CIT file. Where <Devtype> is the destination
                       device type (e.g. "Signal") >>
LSF.COMMANDc.INVOKE TASK or FUNCTION
LSF.COMMANDc.OPTION << for TASK invocation only, the parameters
                       as for the CIT entries without REGISTER >>

Example:
  From the parent directory of <mod>
  > lsfConfig mod
  Parse file 'mod/ws/config/mod.cfg' ... done
  Found 11 devices:
       2 Motor    : M1 M2
       8 Signal   : AI1 AO3 AO4 AI8 DO1 DI2 DI3 DO4
       1 Taskdev  : bgTask
  Found 6 PUBLIC commands:
       1 Motor    : MOVEA
       4 Signal   : RASIG WASIG RDSIG WDSIG
       1 Server   : MAINCMD
  Found 0 MAINTENANCE command
  Found 0 TEST command
  Generate 'mod/ws/config/mod.dbcfg' ... done
  Generate 'mod/lcu/include/modDeviceList.h' ... done
  Generate 'mod/lcu/dbl/mod.db' ... done
  Generate 'mod/lcu/src/Makefile' ... done
  Generate 'mod/lcu/src/modPublic.cdt' ... done
  Generate 'mod/lcu/src/modServer.cit' ... done
  Generate 'mod/lcu/src/modMotor.cit' ... done
  Generate 'mod/lcu/src/modSignal.cit' ... done
```

```
  Move unused files to ./lcu/tmp ... done
  Add change log entry ... done

  >>> Remember to edit the CDT files ...

  Module configuration for 'mod' ... done
  >

5 - Application specific code
  The following files must be edited:
  - <mod>/lcu/src/<mod>*.cdt: describe all the command parameters
    and reply
  - <mod>/lcu/dbl/<mod>DB_DEVICE.class: define the sub-classes
    of lsfDB_DEVICE (or derived) which have been referenced in the
    device description file <mod>/ws/config/<mod>.cfg
  - <mod>/lcu/src/<mod><Devtype>.c: for each referenced device type
    <devtype>, implement the API functions
  - <mod>/lcu/include/<mod><Devtype>.h: declare the API function
    prototypes
  - <mod>/lcu/src/<mod><Devtype>Interface.c: for each command,  in
    implement the ACI functions

6a - Extend the behaviour of the standard commands
  For each of the standard commands, up to 3 hook functions might be
  defined following the prototype:
    ccsCOMPL_STAT <mod>Std<cmd><typ>Hook
                                ( IN  lsfCONTROL_DATA *controlData,
                                  OUT ccsERROR        *error)
  where
    <mod> is the module name,
    <cmd> is the standard command name (Init, Online etc...)
    <typ> is the hook index (Pro,Mid,Epi)
       Pro: hook is invoked when entering the method,
       Mid: hook is invoked between Software and Hardware device control
       Epi: hook is invoked before leaving the method.
  The first (Pro) and last (Epi) are called from within the standard
  object regardless on the number of devices. The middle hook (Mid) is
  only invoked from within this object.
  The prototypes to these functions do not need be declared. The hook
  functions shall be located in <mod>/lcu/src/<mod>Standard.c

  In addition, for the signals only the optional hook function
    ccsCOMPL_STAT <mod>SignalInitHook
                                ( IN  lsfSIGNAL_DATA *signalData,
                                  IN  const char     *signalName,
                                  OUT ccsERROR       *error )
    might be implemented in <mod>/lcu/src/<mod>Signal.c

6b - Extend the behaviour of the device constructors/destructors
  For each device type and each device, a specific additional constructor
  resp. destructor might be implemented. These functions must be defined
  following the prototype:
    ccsCOMPL_STAT <mod><devName>Constructor ( IN  void       *devData,
                                              IN  const char *devName,
                                              OUT ccsERROR   *error)
    void          <mod><devName>Destructor  ( IN  void       *devData )
  The user specific constructors are invoked at completion of the
  standard constructor for each device type, then for each device of
  this type;
  the user specific destructors for each device of each type followed
```

```
  by theuser specific destructor for this device type are invoked
  prior to the standard destructor.

7 - Task devices
  The task devices are scheduled according to the state of the
  Software Device and is configurable in the database.
  The periodic loop associated to a task device is implemented within
  the framework. The core function of the task is mandatory and must
  be prototyped:
    ccsCOMPL_STAT <mod><Devname> ( IN  lsfTASKDEV_DATA *taskDevData,
                                   IN  const char      *taskDevName,
                                   OUT ccsERROR        *error )
  Two optional hooks might be implemented that are invoked when
  starting resp. when terminating the task.
  The function name syntax follows:
    ccsCOMPL_STAT <mod><Devname>ProHook() for the prologue function, and
    ccsCOMPL_STAT <mod><Devname>EpiHook() for the epilogue function,
      same parameters as the core function.

  The method lsfTaskDevTrigger() is provided that triggers
  asynchronously the task, if running.

8 - Building the code
  On both <mod>/ws/src and <mod>/lcu/src, invoke:
    make clean all man install

9 - Building the environments
  The instantiation of the database point associated to the Software
  Device is realized from within the file:
    $VLTDATA/ENVIRONMENTS/<env>/dbl/DATABASE.db

    #define <mod>DB_ROOT "<< database absolute path >>"
    #include "<mod>.db"

  On the LCU environment, the file 'bootScript' shall be updated
  that contains the modules: lcc, cai, scan, too, mcm, lsf and <mod>
  The file 'devicesFile' shall be updated as well with the entry
  corresponding to the new Software Device:
    "<mod>" "<mod>Server" 1 0 0 1
  NB: any Software Device to be controlled by another Software Device
      MUST NOT be declared in this list.
10 - Database configuration file
  The default database configuration file <mod>/ws/config/<mod>.dbcfg
  shall be updated to store the complete final device configuration.
  The backup of the database configuration is made by the utility
  lsfConfig.

  Example:
    From the directory of <mod>/ws/config
    > lsfBackup -e <lcuEnv> -m mod
    Generating input file './mod.inp' ... done
    Performing database backup into  './mod.dbcfg' ... done
    1.24u 49.41s 1:59.60 42.3%
    >
```

SEE ALSO

lsfCreate(1), lsfConfig(1), lsfBackup(1)

3.1.2 lsfCreate(1)

NAME

```
lsfCreate - Create a new module from lsftpl template
```

SYNOPSIS

```
lsfCreate [-f] <modName>
```

DESCRIPTION

```
This script creates a module directory <modName> from the
template module ’lsftpl’.
It checks that the new module is not already existing in the
CMM Archive.
It renames all files with the new prefix.
It replaces all occurences of ’lsftpl’ by ’modName’ in lower
and upper case.
Finally it creates the file <modName>/ChangeLog with one entry.

Options:
-f : the module is created anyway
     the existing module ’modName’ is renamed ’modName-old’
     the module is created even if already in the CMM archive
```

ENVIRONMENT

```
CMM
```

RETURN VALUES

```
0 on SUCCESS
1 on Failure
```

CAUTIONS

```
The module is not created if it already exists under the CMM archive
or if the directory ’modName’ already exists. Option -f bypasses these
checks.
```

EXAMPLES

```
> lsfCreate app
Copying template module ’lsftpl’ to ./app ... done
Renaming all files with prefix ’app’ ... done
Replacing pattern ’lsftpl’ by ’app’ ... done
1.24u 49.41s 1:59.60 42.3%

Module creation ’app’ ... done

>
```

SEE ALSO

```
lsf(1), cmmCopy
```

3.1.3 lsfConfig(1)

NAME

```
lsfConfig - Configure the application
```

SYNOPSIS

```
lsfConfig <modName>
```

DESCRIPTION

```
This script configures the module <modName> according to the
description file <modName>/ws/config/<modName>.cfg

It generates the files:
- <modName>/ws/config/<modName>.dbcfg
- <modName>/lcu/include/<modName>DeviceList.h
- <modName>/lcu/dbl/<modName>.db
- <modName>/lcu/src/Makefile

It removes the unused files.

Finally it adds a log entry in <modName>/ChangeLog.

The syntax of the configuration file must follow:
LSF.MODULE.NAME     <modName>

LSF.DEVICE.NUM      <numDevices>

LSF.DEVICEi.NAME    <devName>
LSF.DEVICEi.TYPE    analog | digital | motor | serial | taskdev | softdev
LSF.DEVICEi.CLASS   <dbClass>
LSF.DEVICEi.PARAM   <parameter value>[, <parameter value>]

LSF.COMMAND.NUM     <numCommands>

LSF.COMMANDi.NAME   <cmdName>
LSF.COMMANDi.DEST   server | analog | digital | motor | serial | taskdev |
softdev
LSF.COMMANDi.GROUP  PUBLIC | MAINTENANCE | TEST
LSF.COMMANDi.ENTRY  <entryFct>
LSF.COMMANDi.INVOKE FUNCTION | TASK
LSF.COMMANDi.OPTION <task parameters> (without REGISTER option)
```

FILES

```
<modName>/ws/config/<modName>.cfg
```

RETURN VALUES

```
0 on SUCCESS
1 on Failure
```

CAUTIONS

```
The script shall be run only once from the parent directory.
The deleted files can not be restored.
```

EXAMPLES

```
> lsfConfig app
Parse file 'app/ws/config/app.cfg' ... done
Found 16 devices:
     2 Motor    : M1 M2
     8 Signal   : AI1 AO3 AO4 AI8 DO1 DI2 DI3 DO4
     3 Serial   : tyCo1 com1 com2
     2 Softdev  : swd soft
     1 Taskdev  : bgTask
Found 10 PUBLIC commands:
     1 Motor    : MOVEA
     4 Signal   : RASIG WASIG RDSIG WDSIG
     2 Serial   : RSCOM WSCOM
     2 Softdev  : SOFTCMD SWDCMD
     1 Server   : MAINCMD
Found 0 MAINTENANCE command
Found 0 TEST command
Generate 'app/ws/config/app.dbcfg' ... done
Generate 'app/lcu/include/appDeviceList.h' ... done
Generate 'app/lcu/dbl/app.db' ... done
Generate 'app/lcu/src/Makefile' ... done
Generate 'app/lcu/src/appPublic.cdt' ... done
Generate 'app/lcu/src/appServer.cit' ... done
Generate 'app/lcu/src/appMotor.cit' ... done
Generate 'app/lcu/src/appSignal.cit' ... done
Generate 'app/lcu/src/appSerial.cit' ... done
Generate 'app/lcu/src/appSoftdev.cit' ... done
Move unused files to ./lcu/tmp ... done
Add change log entry ... done

>>> Remember to edit the CDT files ...

Module configuration for 'app' ... done"

>
```

SEE ALSO

```
lsf(1), lsfCreate(1)
```

BUGS

3.1.4 lsfBackup(1)

NAME

```
lsfBackup - Save LCU database configuration into file
```

SYNOPSIS

```
lsfBackup -e <lcuEnv> -m <modName> [-o <fileName>]
```

DESCRIPTION

```
This script saves the content of the LCU database to a file.
It generates a ".dbcfg" type file containing all the configuration
attributes of the module.

Arguments:
lcuEnv  : LCU environment name
modName : module name (alias of the database top point)
fileName: output file name (default to "./<modName>.dbcfg"
```

FILES

ENVIRONMENT

RETURN VALUES

```
0 on SUCCESS
1 on Failure
```

CAUTIONS

EXAMPLES

```
> lsfBackup -e lcuEnv -m app
Generating input file './app.inp' ... done
Performing database backup into  './app.dbcfg' ... done
1.24u 49.41s 1:59.60 42.3%
>
```

SEE ALSO

```
dbBackup
```

BUGS

3.2 Code

3.2.1 lsfServer(3)

NAME

lsfServer - Top level control of the application

SYNOPSIS

```
#include "lsfServer.h"
```

DESCRIPTION

This class provides the methods for the handling of the server process.

PUBLIC METHODS

```
ccsCOMPL_STAT lsfInitAll
                    ( IN  const char      *swdName,
                      OUT lsfSERVER_DATA **pServerData )
    This method is the constructor of the application server <swdName>.
    It shall be invoked at boot time from the bootScript(5) as part
    of the application file <swdName>.boot(3).
    It receives the address of the pointer to the application global
    data <pServerData>.
    The method loads the database from the file <swdName>.dbcfg
    located under VLTROOT/config or $INTROOT/config.
    It invokes then the server constructor and finally spawns the
    command interpreter with the associated CDT and CIT files.

vltLOGICAL    lsfServerGetInit
                    ( IN lsfSERVER_DATA *serverData )
    This method returns the initialization status of the device.

vltLOGICAL    lsfServerGetSim
                    ( IN  lsfSERVER_DATA *serverData )
    This method returns the simulation status of the device.
lccDEV_MODE   lsfServerGetState
                    ( IN  lsfSERVER_DATA *serverData )
const char   *lsfServerGetStateName
                    ( IN  lsfSERVER_DATA *serverData )
    These methods return the state and state’s name name of the device.

vltINT32      lsfServerGetSubState
                    ( IN  lsfSERVER_DATA *serverData )
const char   *lsfServerGetSubStateName
                    ( IN  lsfSERVER_DATA *serverData )
    These methods return the subState and subState’s name of the device.

ccsCOMPL_STAT lsfServerCheckOnline
                    ( IN  lsfSERVER_DATA *serverData,
                      OUT vltLOGICAL     *online,
                      OUT ccsERROR       *error )
    This method returns the flag <online> indicating if the device
    is in state ONLINE when True.
```

```
lsfSERVER_DATA *lsfServerGetGlobalData
                   ( IN  const char     *swdName,
                     OUT ccsERROR       *error )
    This method returns the pointer to the global data structure of
    the device.

ccsCOMPL_STAT lsfServerGetSwDeviceName
                   ( OUT char           *swdName,
                     OUT ccsERROR       *error )
    This method returns the name of the device from the task name of
    the commmand interpreter process.

vltLOGICAL    lsfServerIsALccDevice
                   ( IN  lsfSERVER_DATA *serverData )
    This method returns the flag indicating if the device is registered
    in the LCC Device Table.
    Any Software Device controlled by another Software Device
    must NOT be registered in LCC.
```

PROTECTED METHODS

```
void          lsfServerDestructor
                   ( IN  lsfSERVER_DATA **pServerData )
    This method is the object destructor. All resources are released.

void          lsfServerGetHooks
                   ( IN  lsfSERVER_DATA  *serverData )
    This method resolves the addresses of all existing hook functions
    provided by the application. It is invoked once at Init time.
    The function entry points are retrieved from the global
    symbol table.

ccsCOMPL_STAT lsfServerCallHook
                   ( IN  lsfSERVER_DATA  *serverData,
                     IN  lsfHOOKFUNC_IDX  cmd,
                     IN  lsfHOOKTYPE_IDX  typ,
                     OUT ccsERROR        *error )
    This method resolves the hook function <typ> [Pro,Mid or Epi]
    associated to the command indexed by <cmd>, if resolved.

ccsCOMPL_STAT lsfServerReadConfig
                   ( IN  lsfSERVER_DATA *serverData,
                     OUT ccsERROR       *error )
    This method reads the device configuration from the database.

ccsCOMPL_STAT lsfServerSetInit
                   ( IN  lsfSERVER_DATA *serverData,
                     IN  vltLOGICAL      value,
                     OUT ccsERROR       *error )
    These methods set/return the initialization status of the device.

ccsCOMPL_STAT lsfServerSetSim
                   ( IN  lsfSERVER_DATA *serverData,
                     IN  vltLOGICAL      value,
                     OUT ccsERROR       *error )
    This method sets the simulation status of the device.
```

```
ccsCOMPL_STAT lsfServerTakeInitSem
                   ( IN  lsfSERVER_DATA *serverData )
void          lsfServerGiveInitSem
                   ( IN  lsfSERVER_DATA *serverData )
    These methods control the initialization semaphore associated
    to the device. It prevents multiple invocation.

ccsCOMPL_STAT lsfServerTakeCmdSem
                   ( IN  lsfSERVER_DATA *serverData )
void          lsfServerGiveCmdSem
                   ( IN  lsfSERVER_DATA *serverData )
    These methods control the command semaphore associated
    to the device. It prevents multiple invocation.

void          lsfServerWakeupMonitor
                   ( IN lsfSERVER_DATA *serverData  )
    This method triggers the monitor task for state update

vltINT32      lsfServerGetCmdTimeout
                   ( IN  lsfSERVER_DATA *serverData )
void          lsfServerSetCmdTimeout
                   ( IN  lsfSERVER_DATA *serverData,
                     IN  vltINT32        timeout )
    These methods set/get the timeout value associated to the commands.

ccsCOMPL_STAT lsfServerSwitchState
                   ( IN  lsfSERVER_DATA *serverData,
                     IN  lccDEV_MODE     newDevState,
                     OUT ccsERROR       *error )
    This method instructs LCC of the new state <newDevState> of the
    device.

int           lsfStartupTask
                   ( IN  char           *name,
                     IN  vltINT32        priority,
                     IN  vltINT32        stackSize,
                     IN  FUNCPTR         entryPoint,
                     IN  void           *param,
                     OUT ccsERROR       *error )
    This method spawns the task <name> with the appropiate parameters,
    and checks the task has synchronized with the caller. The task entry
    function must comply with the prototype lsfTASK_FUNCTION.
      typedef void lsfTASK_FUNCTION ( IN void          *parameter,
                                      IN SEM_ID        *syncSem,
                                      IN ccsCOMPL_STAT *status )
    The first argument is application dependent. The semaphore <syncSem>
    must be given by the task within 1 second, and the task status
    <status> shall be set accordingly to SUCCESS or FAILURE.
```

PRIVATE METHODS

```
static ccsCOMPL_STAT lsfServerUpdateState
                   ( IN  lsfSERVER_DATA  *serverData,
                     OUT ccsERROR        *error )
static ccsCOMPL_STAT lsfServerUpdateSubState
                   ( IN  lsfSERVER_DATA  *serverData,
                     OUT ccsERROR        *error )
    These 2 methods are responsible for updating the state and subState
    of the device. They query the individual states and subStates of all
    registered devices.
    The device state is set to the lower state of the devices.
    The subState is set to :
    - lsfSUBSTATE_ERROR if any device is in that subState.
    - lsfSUBSTATE_INITIALIZING if any device is in that subState.
    - lsfSUBSTATE_IDLE  if all devices are in that subState
    - lsfSUBSTATE_ACTIVE if any device is not lsfSUBSTATE_IDLE and
         the device subState is not lsfSUBSTATE_IDLE
    - lsfSUBSTATE_<ACTION> if a device is in that subState and
         the device subState is lsfSUBSTATE_IDLE
    The following subStates are dedicated:
    - lsfSUBSTATE_MOVING for a motor when moving
    - lsfSUBSTATE_MONITORING for a device monitoring a piece of hardware
    In addition, the subState lsfSUBSTATE_TIMEOUT indicates a timeout
    occured.
    Both lsfSUBSTATE_TIMEOUT and lsfSUBSTATE_ERROR are transient
    subStates that reset to lsfSUBSTATE_IDLE with the next action.

static ccsCOMPL_STAT lsfServerInitDb
                   ( IN  const char      *swdName,
                     OUT ccsERROR        *error )
    This method is invoked at boot time by lsfInitAll(). It restores
    the content of the device database branch from the file
    <swdName>.dbcfg located under $VLTROOT/config or $INTROOT/config.

static void lsfServerMonitor
                   ( IN  void            *pData,
                     IN  SEM_ID          *pSyncSem,
                     IN  ccsCOMPL_STAT   *pStatus )
    This method is invoked as a task (see lsfStartupTask) at construction
    time. It is responsible for updating the device global state and
    subState at the given rate (see lsfSERVER(5)).
    Note: be aware that a high polling rate is CPU consuming and might
    impact on the overall behaviour of the system.

static ccsCOMPL_STAT lsfServerConstructor
                   ( IN  const char      *swdName,
                     OUT lsfSERVER_DATA **pServerData,
                     OUT ccsERROR        *error )
    This method is the constructor of the software device. It invokes
    the control contructor (see lsfControl(3)) and spawns the monitor.
```

PRIVATE DATA MEMBERS

```
typedef struct
{
  void         *ctrlData;     Ptr to Control Data structure
  lccDEV_MODE  state;         Module state
  vltINT32     subState;      Module sub-state
  vltLOGICAL   init;          True if initialized
  vltLOGICAL   simulation;    True if in simulation
  vltINT32     numDevices;    number of controlled devices (registered)

  SEM_ID       cmdSem;        semaphore for sending commands
  vltINT32     cmdTimeout;    timeout for sending commands
  SEM_ID       initSem;       semaphore for init command
  vltLOGICAL   lccDevice;     SW device is a LCC device
  int          monTid;        Monitoring Task Id
  SEM_ID       monSem;        semaphore for monitor task period
  int          monPeriod;     Monitoring Task Period (ticks)
  ccsERROR     error;         error stack for cmdInit()

  char         *attrTable;    cai DB attribute table

  vltBYTES20   swdName;       Name of the SW device
  void         *fctHook[8][3]; Hook function addresses
} lsfSERVER_DATA;
```

FILES

```
$INTROOT/config/<modName>.dbcfg
```

DATABASE

The class lsfDB_SERVER is a sub-class of lsfDB_SOFTDEV.
It is partially mapped by the data structure lsfSERVER_DATA.

SEE ALSO

lsfControl(3), lsfDevice(3), lsfDB_DEVICE(5)

3.2.2 lsfStandard(3)

NAME

```
lsfStandard - Standard commands
```

SYNOPSIS

```
#include "lsfStandard.h"
```

DESCRIPTION

```
This class provides the methods implementing the high level default
behaviour of the standard commands.
For each of the standard commands, up to 3 hook functions might be
defined following the syntax:
  <mod>Std<cmd><typ>Hook(lsfSERVER_DATA *serverData, ccsERROR *error)
where
  <mod> is the module name,
  <cmd> is the standard command name (Init, Online etc...)
  <typ> is the hook index (Pro,Mid,Epi)
       Pro: hook is invoked when entering the method,
       Mid: hook is invoked between Software and Hardware device control
       Epi: hook is invoked before leaving the method.
The first (Pro) and last (Epi) are called regardless on the number
of devices. The middle hook (Mid) is only invoked from within the control
object.
```

PUBLIC METHODS

```
ccsCOMPL_STAT lsfStdInit
ccsCOMPL_STAT lsfStdStandby
ccsCOMPL_STAT lsfStdOnline
ccsCOMPL_STAT lsfStdStop
ccsCOMPL_STAT lsfStdExit
ccsCOMPL_STAT lsfStdStartSim
ccsCOMPL_STAT lsfStdStopSim
ccsCOMPL_STAT lsfStdSelfTest
ccsCOMPL_STAT lsfStdTest
                   ( IN  lsfSERVER_DATA *serverData,
                     IN  const char     *devName,
                     OUT ccsERROR       *error )
    These methods implement the high level behaviour associated
    to the standard commands. They invoke the associated control
    method for the specified device <devName>.
    If <devName> is set to the software device name, the method
    affects all devices registered in that software device, otherwise
    the action is restricted to the one device.
    At completion, the methods update the global state accordingly.

ccsCOMPL_STAT lsfStdGetInit
                   ( IN  lsfSERVER_DATA *serverData,
                     OUT vltLOGICAL     *init,
                     OUT ccsERROR       *error )
    This method returns the initialization status.

ccsCOMPL_STAT lsfStdGetSimulation
                   ( IN  lsfSERVER_DATA *serverData,
                     OUT vltLOGICAL     *simulation,
                     OUT ccsERROR       *error )
```

```
    This method returns the value of the simulation flag.

ccsCOMPL_STAT lsfStdGetVersion
                   ( OUT char           *version,
                     OUT ccsERROR       *error )
    This method returns the software version of the module LSF.
```

PRIVATE METHODS

```
static const char *lsfCheckDevice
                   ( IN  lsfSERVER_DATA *serverData,
                     IN  const char     *devName,
                     OUT ccsERROR       *error )
    This method checks if the given device is known:
    - name of the software device or "all", it returns "all"
    - name of a controlled device, it returns the name if defined.
    - unknown name, it returns NULL.
```

PRIVATE DATA MEMBERS

```
These methods use the data structure <serverData>.
```

COMMANDS

```
The public methods are the API functions associated to the
ACI for the Standard Commands.
```

RETURN VALUES

```
SUCCESS if everything ok
FAILURE, the error structure is updated with:
    lsfERR_DEVICE           : unknown device name or
                              module name mismatch
    lsfERR_MULTIPLE_COMMAND : multiple invocation not allowed
```

SEE ALSO

```
lsfServer(3)
```

3.2.3 lsfSignalHandlers(3)

NAME

lsfHandleBreak, lsfHandleKill - signal handler for BREAK and KILL signals

SYNOPSIS

```
#include "lsfSignalHandlers.h"

void lsfHandleKill(void);
void lsfHandleBreak(void);
```

DESCRIPTION

lsfHandleBreak() and lsfHandleKill() are the 2 signal handler methods installed at boot time by lsfInitAll(). These methods retrieve the name of the affected software device from the server task name before performing the corresponding actions (STOP, resp. EXIT).

RETURN VALUES

None

SEE ALSO

lsfStandard(3), lsfServer(3)

3.2.4 lsfControl(3)

NAME

```
lsfControl - Device control
```

SYNOPSIS

```
#include "lsfControl.h"
```

DESCRIPTION

```
This class provides the methods dedicated to the control
of the devices. It is driven by the deviceTable. This table is read
at init time.
For each of the standard commands, the associated control method
invokes the device methods for each type of device assuming devices of
that type are registered and not ignored.
First the sub-software devices are invoked, then the hardware
devices whereby the digital and analogue signals are processed first,
followed by the motors.
For each of the standard commands, up to 3 hook functions might be
defined following the syntax:
  <mod>Std<cmd><typ>Hook(lsfCONTROL_DATA *controlData, ccsERROR *error)
where
  <mod> is the module name,
  <cmd> is the standard command name (Init, Online etc...)
  <typ> is the hook index (Pro,Mid,Epi)
       Pro: hook is invoked when entering the method,
       Mid: hook is invoked between Software and Hardware device control
       Epi: hook is invoked before leaving the method.
The first (Pro) and last (Epi) are called from within the standard
object regardless on the number of devices. The middle hook (Mid) is
only invoked from within this object.
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfControlConstructor
                      ( IN  const char      *swdName,
                        IN  lsfDEVICE_DATA  *serverData,
                        OUT ccsERROR        *error )
    This method is the object constructor. It invokes the constructors
    of all devices registered in the device table.
void lsfControlDestructor
                      ( IN lsfDEVICE_DATA *serverData )
    This method is the object destructor. It invokes the destructors
    of all registered devices. All resources are released.
const char *lsfControlGetSwdName
                      ( IN  void *pData )
    This method returns the name of the software device (= module name)
void *lsfControlGetDevData
                      ( IN lsfCONTROL_DATA *controlData,
                        IN int devType )
    This method returns a pointer to the device data structure
    of the given device type.
int lsfControlGetDevType
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName )
    This method returns the type of the given device.
int lsfControlGetDevNamesOfType
```

```
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  int              devType,
                        IN  const char      *devNames[] )
    This method returns the number of devices of the given type.
    The array <devNames[]> contains the names of all the matching
    devices.
ccsCOMPL_STAT lsfControlInit
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error)
ccsCOMPL_STAT lsfControlStandby
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
ccsCOMPL_STAT lsfControlOnline
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
ccsCOMPL_STAT lsfControlStop
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error)
ccsCOMPL_STAT lsfControlOff
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error)
ccsCOMPL_STAT lsfControlExit
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
ccsCOMPL_STAT lsfControlSimulat
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
ccsCOMPL_STAT lsfControlStopsim
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
ccsCOMPL_STAT lsfControlSelftest
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
ccsCOMPL_STAT lsfControlTest
                      ( IN  lsfCONTROL_DATA *controlData,
                        IN  const char      *devName,
                        OUT ccsERROR        *error )
    These methods perform the actions corresponding to the standard
    behaviour.
```

PRIVATE METHODS

```
static int lsfDecodeDeviceType
                      (char *type)
    This method returns the associated device type associated to
    the device type name.
static ccsCOMPL_STAT lsfControlReadDevTbl
                      ( IN  lsfCONTROL_DATA *controlData,
                        OUT ccsERROR        *error )
    This method reads the database attribute control:deviceTable
    and sets the private data member <deviceTable>.
```

PRIVATE DATA MEMBERS

```
The data structure lsfCONTROL_ENTRY maps the database attribute
:control.deviceTable.
typedef struct
{
  vltBYTES20 name;                Name of the device
  vltINT32   type;                Type of the device
  vltINT32   startPhase;          Sequence order for increasing/
  vltINT32   stopPhase;             decreasing the state
  vltLOGICAL simulation;          True when device is in Simulation
  vltLOGICAL ignored;             True when device is ignored
} lsfCONTROL_ENTRY;

The data structure lsfCONTROL_DATA holds all the necessary information
for the control of the devices.
typedef struct
{
  void             *ctrlData;      Ptr to self
  void             *srvrData;      Ptr to parent structure
  int               numDevices[lsfDEVICE_NUM_TYPES+1];
                                   Number of devices of each type
  lsfCONTROL_ENTRY *deviceTable;
                                   Device table (mirror of DB)
  lsfDEVICE_DATA   *deviceData[lsfDEVICE_NUM_TYPES+1];
                                   Device data
  char             *attrTable;     LCC cai internal
  void             *fctHook[8][3]; Hook function addresses
} lsfCONTROL_DATA;
```

DATABASE

```
The class lsfCONTROL is the class holding the description of the
device. It consists of the deviceTable attribute that stores the
description of each device to be controlled:
    vltBYTES20 name;               Name of the device
    vltBYTES20 type;               Type of the device
    vltINT32   startPhase;         Sequence order for increasing/
    vltINT32   stopPhase;            decreasing the state
    vltLOGICAL simulation;         Device is in Simulation
    vltLOGICAL ignored;            Device is ignored
For each kind of device a point is instanciated from the class lsfDEVICE
that holds the information of that kind of devices (state, subState
etc...). These points are instanciated dynamically at build time
depending on the number of devices of a kind.
```

RETURN VALUES

```
The control sequence is broken on the first failure.
The method returns FAILURE and an error is logged.
```

CAUTIONS

```
Start/Stop Phase handling mechanism is not yet implemented.
```

SEE ALSO

```
lsfCONTROL(5)
```

3.2.5 lsfDevice(3)

NAME

```
lsfDevice - Common device handling methods
```

SYNOPSIS

```
#include "lsfDevice.h"
```

DESCRIPTION

```
This class provides methods dedicated to the handling of devices.
The allows other objects to query device private information and
to set the device state/subState.
```

PUBLIC METHODS

```
vltLOGICAL    lsfDeviceIsDefined   ( IN  void       *pData,
                                     IN  const char *devName );
vltLOGICAL    lsfDeviceIsIgnored   ( IN  void       *pData,
                                     IN  const char *devName );
vltLOGICAL    lsfDeviceIsSimulated ( IN  void       *pData,
                                     IN  const char *devName );
vltLOGICAL    lsfDeviceIsInState   ( IN  void       *pData,
                                     IN  vltINT32    state );
    These 4 methods query the device status.
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfDeviceSetStatus   ( IN  void       *pData,
                                     IN  vltINT32    devIdx,
                                     IN  vltINT32    state,
                                     IN  vltINT32    subState,
                                     IN  vltINT32    init,
                                     OUT ccsERROR   *error );
    This method sets the device status to the given values for:
    the <state>, the <subState> and the <init> flag.
    The index <devIdx> refers to the selected device when > 0
    or to the device group when zero.
ccsCOMPL_STAT lsfDeviceSetState    ( IN  void       *pData,
                                     IN  vltINT32    devIdx,
                                     IN  vltINT32    state,
                                     OUT ccsERROR   *error );
    Idem dito for the <state> only.
ccsCOMPL_STAT lsfDeviceSetSubState ( IN  void       *pData,
                                     IN  vltINT32    devIdx,
                                     IN  vltINT32    subState,
                                     OUT ccsERROR   *error );
    Idem dito for the <subState> only.
ccsCOMPL_STAT lsfDeviceSetInit     ( IN  void       *pData,
                                     IN  vltINT32    devIdx,
                                     IN  vltLOGICAL  init,
                                     OUT ccsERROR   *error );
    Idem dito for the <init> flag only.
```

PRIVATE DATA MEMBERS

```
typedef struct
{
  vltINT32         state;            State of the device
  vltINT32         subState;         Associated sub-state
  vltLOGICAL       init;             True when Initialized
  vltLOGICAL       simulation;       True when in Simulation
  vltINT32         timeout;          Device action timeout in seconds
} lsfDEVICE_ENTRY;

typedef struct
{
  void            *ctrlData;         Ptr to the CONTROL_DATA structure
  void            *userData;         Ptr to User data
  vltINT32         state;            State, SubState and Init/Sim flag of
  vltINT32         subState;           all the devices of this type
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;          Timeout in seconds
  vltINT32         numDevices;       Number of devices of this type
  vltBYTES20      *deviceNames;      Names of the devices
  char            *attrTable[lsfMAX_DEVICES+1]; LCC cai internal
  lsfDEVICE_ENTRY *deviceTable[lsfMAX_DEVICES]; Device individual data
} lsfDEVICE_DATA;
```

DATABASE

The class lsfDB_DEVICE is the generic class for device description:
It is partially mapped by the data structure lsfDEVICE_ENTRY.
Each type of device is represented in the database branch of the
Software Device by a point instanciated from lsfDB_DEVICE
and located under :control.
Each device of a kind is represented in the database by a point
instanciated from a sub-class of lsfDB_DEVICE and
located under :control:<devType>.

SEE ALSO

lsfControl(3), lsfDB_DEVICE(5)

3.2.6 lsfSignal(3)

NAME

```
lsfSignal - Control digital and analogue signals
```

SYNOPSIS

```
#include "lsfSignal.h"
```

DESCRIPTION

```
This class provides the methods for the control of digital and
analogue signals. It implements the default behaviour for all
the standard commands.
A user-defined hook function is foreseen to be invoked at
completion of the signal initialization.
This hook shall follow the syntax:
  ccsCOMPL_STAT <mod>SignalInitHook(lsfSIGNAL_DATA *signalData,
                                    const char     *signalName,
                                    ccsERROR       *error)
In addition, it offers the 4 methods for reading and setting
digital and analogue signals as a whole or individually.
```

PUBLIC METHODS

```
ccsCOMPL_STAT lsfSignalReadAnalog
                             ( IN  lsfSIGNAL_DATA *signalData,
                               IN  const char     *signalName,
                               OUT char           *names[],
                               OUT vltFLOAT       *value,
                               OUT ccsERROR       *error )
    This method reads the specified analog signal value(s) and
    stores it/them in the array pointed to by <value>. The names
    of the signals are listed and returned to the caller via the
    array <names[]>.
    If <signalName> is NULL or "all", all input analogue signals
    will be read.
ccsCOMPL_STAT lsfSignalWriteAnalog
                             ( IN  lsfSIGNAL_DATA *signalData,
                               IN  const char     *signalName,
                               IN  vltFLOAT       *value,
                               OUT ccsERROR       *error )
    This method sets the specified analog signal <signalName>
    to the value pointed to by <value>.
    If <signalName> is NULL or "all", all output analogue signals
    will be set to the values pointed to by <value> in the order
    they are registered.
ccsCOMPL_STAT lsfSignalReadDigital
                             ( IN  lsfSIGNAL_DATA *signalData,
                               IN  const char     *signalName,
                               OUT char           *names[],
                               OUT vltUINT32      *value,
                               OUT ccsERROR       *error )
    This method reads the specified digital signal value(s) and
    stores it/them in the array pointed to by <value>. The names
    of the signals are listed and returned to the caller via the
    array <names[]>.
    If <signalName> is NULL or "all", all input digital signals
    will be read.
```

```
ccsCOMPL_STAT lsfSignalWriteDigital
                             ( IN  lsfSIGNAL_DATA *signalData,
                               IN  const char     *signalName,
                               IN  vltUINT32      *value,
                               OUT ccsERROR       *error )
    This method sets the specified digital signal <signalName>
    to the value pointed to by <value>.
    If <signalName> is NULL or "all", all output digital signals
    will be set to the values pointed to by <value> in the order
    they are registered.
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfSignalConstructor
                             ( IN  const char      *swdName,
                               IN  lsfCONTROL_DATA *controlData,
                               IN  const char      *signalNames[],
                               OUT ccsERROR        *error )
    This is the constructor of the signal object. It affects all the
    signals given in the list <signalNames[]> (NULL terminated).
void          lsfSignalDestructor
                             ( IN  lsfCONTROL_DATA *controlData )
    This method is the object destructor. All resources are released.

ccsCOMPL_STAT lsfSignalInit
ccsCOMPL_STAT lsfSignalStandby
ccsCOMPL_STAT lsfSignalOnline
ccsCOMPL_STAT lsfSignalStop
ccsCOMPL_STAT lsfSignalOff
ccsCOMPL_STAT lsfSignalExit
ccsCOMPL_STAT lsfSignalSimulat
ccsCOMPL_STAT lsfSignalStopsim
ccsCOMPL_STAT lsfSignalSelftest
ccsCOMPL_STAT lsfSignalTest
                             ( IN  lsfSIGNAL_DATA *signalData,
                               IN  const char     *signalName,
                               OUT ccsERROR       *error )
    These methods implement the behaviour of the standard commands.
    If <signalName> is NULL, or "all" all the signals will be
    affected, otherwise only the specified one.
```

PRIVATE METHODS

```
static ccsCOMPL_STAT lsfSignalCallHook
                             ( IN  lsfSIGNAL_DATA *signalData,
                               IN  const char     *signalName,
                               IN  const char     *signalFct,
                                OUT ccsERROR       *error )
    This method invokes the user-defined hook function
    <mod>Signal<signalFct>Hook(<signalData>,<signalName>,error).
    It is invoked at Init completion only.
```

PRIVATE DATA MEMBERS

```
The data stucture lsfSIGNAL_ENTRY is a sub-class of lsfDEVICE_ENTRY.
typedef struct
{
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
                               Additional data members
  ioTYPE           ioType;        Type of signal (Digital/Analog)
  ioIN_OUT         direction;     Signal direction (Input/Output)
  dbSYMADDRESS     dbAddr;        DB path to Signal point
  ioDIRADDRESS     ioAddr;        IO Direct address
} lsfSIGNAL_ENTRY;

The data stucture lsfSIGNAL_DATA is a sub-class of lsfDEVICE_DATA.
typedef struct
{
  void            *ctrlData;
  void            *userData;
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
  vltINT32         numSignals;                   Number of signals

  vltBYTES20       *signalNames;                 Signal names
                                                (dynamic allocated)
  char             *attrTable[lsfMAX_DEVICES+1];
  lsfSIGNAL_ENTRY  *signalTable[lsfMAX_DEVICES];  Signal data
} lsfSIGNAL_DATA;
```

DATABASE

```
The class lsfDB_SIGNAL is a sub-class of lsfDB_DEVICE.
It is partially mapped by the data structure lsfSIGNAL_ENTRY.
Each signal is represented in the database by a point under the
main signal point :control:signal, which is partially mapped
by the data structure lsfSIGNAL_DATA.
The 2 sub-classes lsfDB_ANALOG and lsfDB_DIGITAL are specialized classes
for Analogue, resp. Digital signals. They are instanciating the LCC
classes lccANALOG_SIGNAL and lccDIGITAL_SIGNAL resp.
```

SEE ALSO

```
lsfDB_SIGNAL(5)
```

3.2.7 lsfMotor(3)

NAME

```
lsfMotor - Control motors
```

SYNOPSIS

```
#include "lsfMotor.h"
```

DESCRIPTION

```
This class provides the methods for the control of motors.
It implements the default behaviour for all the standard commands.
In addition, it offers 2 methods for performing absolute motions
on one or all motors in parallel.
```

PUBLIC METHODS

```
vltINT32 lsfMotorGetState    ( IN  lsfMOTOR_DATA *motorData,
                               IN  const char    *motorName )
vltINT32 lsfMotorGetSubState ( IN  lsfMOTOR_DATA *motorData,
                               IN  const char    *motorName )
    These 2 methods return the state, resp. sub-state of the motors
    specified by <motorName>.

ccsCOMPL_STAT lsfMotorMoveAbs
                             ( IN  lsfMOTOR_DATA *motorData,
                               IN  const char    *motorName,
                               IN  vltINT32       target[],
                               IN  const char    *unit[],
                               IN  vltLOGICAL     wait,
                               OUT ccsERROR      *error )
    This methods initiates an absolute motion to the position
    <target> for all the motors specified by <motorName>. The
    position unit is specified by <unit> in the corresponding order.
    The flag <wait> indicates whether the function shall wait for
    the motion completion, or not.

ccsCOMPL_STAT lsfMotorWaitMove
                             ( IN  lsfMOTOR_DATA *motorData,
                               IN  const char    *motorName,
                               OUT char          *names[],
                               OUT vltDOUBLE     *position,
                               OUT char          *units[],
                               OUT ccsERROR      *error )
    This method is based on the API function motWaitMove() that
    waits until all motors specified by <motorName> have completed
    their motion. It returns on the first error.
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfMotorConstructor
                             ( IN  const char      *swdName,
                               IN  lsfCONTROL_DATA *controlData,
                               IN  const char      *motorNames[],
                               OUT ccsERROR        *error )
    This is the constructor of the motor object. It affects all the
    motors given in the list <motorNames[]> (NULL terminated).
void          lsfMotorDestructor
                             ( IN  lsfCONTROL_DATA *controlData )
    This method is the object destructor. It deinstalls all
    of all registered motors. All resources are released.

ccsCOMPL_STAT lsfMotorInit
ccsCOMPL_STAT lsfMotorStandby
ccsCOMPL_STAT lsfMotorOnline
ccsCOMPL_STAT lsfMotorStop
ccsCOMPL_STAT lsfMotorOff
ccsCOMPL_STAT lsfMotorExit
ccsCOMPL_STAT lsfMotorSimulat
ccsCOMPL_STAT lsfMotorStopsim
ccsCOMPL_STAT lsfMotorSelftest
ccsCOMPL_STAT lsfMotorTest   ( IN  lsfMOTOR_DATA  *motorData,
                               IN  const char     *motorName,
                               OUT ccsERROR       *error )
    These methods implement the behaviour of the standard commands.
    If <motorName> is NULL, or "all" all the motors will be
    affected, otherwise only the specified one.

ccsCOMPL_STAT lsfMotorWaitInit
                             ( IN  lsfMOTOR_DATA *motorData,
                               IN  const char    *motorName,
                               OUT ccsERROR      *error )
    This method is based on the API function motWaitInit() that
    waits until all motors specified by <motorName> have completed
    their initialization sequence. It returns on the first error.

ccsCOMPL_STAT lsfMotorSetSubState
                             ( IN  lsfMOTOR_DATA *motorData,
                               OUT ccsERROR      *error )
    This method sets the sub-states of all registered motors as a
    function of their resp. initStatus and motionStatus.
```

PRIVATE DATA MEMBERS

```
The data stucture lsfMOTOR_ENTRY is a sub-class of lsfDEVICE_ENTRY.
typedef struct
{
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
                               Additional data members
  motSTATUS       *motorStatus;   Ptr to motorStatus array
                                  from lsfMOTOR_DATA
} lsfMOTOR_ENTRY;
```

```
The data stucture lsfMOTOR_DATA is a sub-class of lsfDEVICE_DATA.
typedef struct
{
  void            *ctrlData;
  void            *userData;
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
  vltINT32         numMotors;                   Number of motors

  vltBYTES20      *motorNames;                  Motor names
                                                (dynamic allocated)
  char            *attrTable[lsfMAX_DEVICES+1];
  lsfMOTOR_ENTRY  *motorTable[lsfMAX_DEVICES];   Motor data
                              Additional data members
  motHANDLE        motorHandle[lsfMAX_DEVICES+1]; List of motor
                                                 handles [0 terminated]
  motSTATUS       *motorStatus;                 Array of motor status
                                                (dynamic allocated)
} lsfMOTOR_DATA;
```

DATABASE

```
The class lsfDB_MOTOR is a sub-class of lsfDB_DEVICE.
It is partially mapped by the data structure lsfMOTOR_ENTRY.
Each motor is represented in the database by a point under the
main  warning: imsignal point :control:motor, which is partially mapped
by the data structure lsfMOTOR_DATA.
```

SEE ALSO

```
lsfDB_MOTOR(5)
```

3.2.8 lsfSerial(3)

NAME

lsfSerial - Control serial communication links

SYNOPSIS

```
#include "lsfSerial.h"
```

DESCRIPTION

This class provides the methods for the control of serial communication links. It implements the default behaviour for all the standard commands.

PUBLIC METHODS

```
ccsCOMPL_STAT lsfSerialRead  ( IN  lsfSERIAL_DATA *serialData,
                               IN  const char     *serialName,
                               OUT char           *names[],
                               OUT char           *buffer[],
                               OUT ccsERROR       *error)
    This method reads data from the given serial communication
    link <serialName> into the buffer pointed to by <buffer>.
    If <serialName> is NULL or "all", all communication links
    will be read in the order they are registered. The names of
    devices read are returned in the array <names[]>. The data
    buffers are ordered identically.

ccsCOMPL_STAT lsfSerialWrite ( IN  lsfSERIAL_DATA *serialData,
                               IN  const char     *serialName,
                               IN  char           *buffer[],
                               OUT ccsERROR       *error )
    This method writes data pointed to by <buffer> to the given
    serial communication link <serialName>.
    If <serialName> is NULL or "all", all communication links
    will be written in the order they are registered.
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfSerialConstructor
                             ( IN  const char      *swdName,
                               IN  lsfCONTROL_DATA *controlData,
                               IN  const char      *serialNames[],
                               OUT ccsERROR        *error )
    This is the constructor of the serial object. It affects all the
    serials given in the list <serialNames[]> (NULL terminated).
void          lsfSerialDestructor
                             ( IN  lsfCONTROL_DATA *controlData )
    This method is the object destructor. All resources are released.

ccsCOMPL_STAT lsfSerialInit
ccsCOMPL_STAT lsfSerialStandby
ccsCOMPL_STAT lsfSerialOnline
ccsCOMPL_STAT lsfSerialStop
ccsCOMPL_STAT lsfSerialOff
ccsCOMPL_STAT lsfSerialExit
ccsCOMPL_STAT lsfSerialSimulat
ccsCOMPL_STAT lsfSerialStopsim
```

```
ccsCOMPL_STAT lsfSerialSelftest
ccsCOMPL_STAT lsfSerialTest
                             ( IN  lsfSERIAL_DATA *serialData,
                               IN  const char     *serialName,
                               OUT ccsERROR       *error )
    These methods implement the behaviour of the standard commands.
    If <serialName> is NULL, or "all" all the serials will be
    affected, otherwise only the specified one.
```

PRIVATE METHODS

```
static ccsCOMPL_STAT lsfSerialCallHook
                             ( IN  lsfSERIAL_DATA *serialData,
                               IN  const char     *serialName,
                               IN  const char     *serialFct,
                               OUT ccsERROR       *error )
    This method invokes the user-defined hook function
    <mod>Serial<serialFct>Hook(<serialData>,<serialName>,error).
    It is invoked at Init completion only.
```

PRIVATE DATA MEMBERS

```
The data stucture lsfSERIAL_ENTRY is a sub-class of lsfDEVICE_ENTRY.
typedef struct
{
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
                               Additional data members
  vltINT32         fd;            File descriptor
  vltLOGICAL       isTyCo;        True when TyCo device
} lsfSERIAL_ENTRY;

The data stucture lsfSERIAL_DATA is a sub-class of lsfDEVICE_DATA.
typedef struct
{
  void            *ctrlData;
  void            *userData;
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
  vltINT32         numSerials;                  Number of serials

  vltBYTES20       *serialNames;                Serial names
                                               (dynamic allocated)
  char             *attrTable[lsfMAX_DEVICES+1];
  lsfSERIAL_ENTRY  *serialTable[lsfMAX_DEVICES];  Serial data
} lsfSERIAL_DATA;
```

DATABASE

```
     The class lsfDB_SERIAL is a sub-class of lsfDB_DEVICE.
     It is partially mapped by the data structure lsfSERIAL_ENTRY.
     Each serial communication link is represented in the database by a
     point under the main serial point :control:serial, which is partially
     mapped by the data structure lsfSERIAL_DATA.
     The sub-classes lsfDB_RS232, lsfDB_RS422 and lsfDB_RS485 derive from
lsfDB_SERIAL
     whereby the protocol and handshake are overloaded.
```

SEE ALSO

```
     lsfDB_SERIAL(5)
```

3.2.9 lsfTaskDev(3)

NAME

```
lsfTaskDev - Control Task Devices
```

SYNOPSIS

```
#include "lsfTaskDev.h"
```

DESCRIPTION

```
This class provides the methods for the control of task devices.
It implements the default behaviour for all the standard commands.
Function of the state, the task devices are activated/stopped.
```

PUBLIC METHODS

```
ccsCOMPL_STAT lsfTaskDevTrigger
                             ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  const char      *taskDevName,
                               OUT ccsERROR        *error )
    This method triggers the given task asynchronously.
    If the timer is based on TIM, the activation occurs asynchronously
    within the period; otherwise, the periodic delay is restarted
    after activation.

STATUS        lsfTaskDevVerify
                             ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  const char      *taskDevName,
                               OUT ccsERROR        *error )
    This method returns the status of the given task device.
    OK if running; ERROR if not. see taskIdVerify()
    If <taskDevName> is "all" the return value is OK if all tasks
    are running otherwise ERROR;
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfTaskDevConstructor
                             ( IN  const char      *swdName,
                               IN  lsfCONTROL_DATA *controlData,
                               IN  const char      *taskDevNames[],
                               OUT ccsERROR        *error )
    This is the constructor of the taskDev object. It affects all the
    task devices given in the list <taskDevNames[]> (NULL terminated).
void          lsfTaskDevDestructor
                             ( IN  lsfCONTROL_DATA *controlData )
    This method is the object destructor. All resources are released.

ccsCOMPL_STAT lsfTaskDevInit
ccsCOMPL_STAT lsfTaskDevStandby
ccsCOMPL_STAT lsfTaskDevOnline
ccsCOMPL_STAT lsfTaskDevStop
ccsCOMPL_STAT lsfTaskDevOff
ccsCOMPL_STAT lsfTaskDevExit
ccsCOMPL_STAT lsfTaskDevSimulat
ccsCOMPL_STAT lsfTaskDevStopsim
ccsCOMPL_STAT lsfTaskDevSelftest
ccsCOMPL_STAT lsfTaskDevTest ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  const char      *taskDevName,
```

```
                              OUT ccsERROR        *error )
    These methods forward the standard commands to the task
    devices given in the list <taskDevName>.
    If <taskDevName> is NULL, or "all" all the task devices will
    be affected, otherwise only the specified one.
    It updates the state and subState of the group.
```

PRIVATE METHODS

```
static ccsCOMPL_STAT lsfTaskDevLoop
                             ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  int              tskIdx,
                               OUT ccsCOMPL_STAT   *tskStat,
                               IN  SEM_ID          *tskSync,
                               OUT ccsERROR        *error )
    This method is the core of the task. Therefore the structure is
    imposed. It invoked an optional prologue user function
    <mod><taskDevName>ProHook(), then enters the infinite loop
    in which it invokes the user core function <mod><taskDevName>()
    When the task is told to terminate, it invoked the optional
    epilogue user function <mod><taskDevName>EpiHook().
    All the task hook functions must comply the prototype:

      typedef ccsCOMPL_STAT lsfTASKDEV_HOOK
                             ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  const char      *taskDevName,
                               OUT ccsERROR        *error)
    The function names must follow the syntax:
      <moduleName><taskDevName>["ProHook" | "" | "EpiHook"]()
    The core function <moduleName><taskDevName>() is mandatory.

static ccsCOMPL_STAT lsfTaskDevActivate
                             ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  int              tskIdx,
                               OUT ccsERROR        *error )
    This method activates the task indexed <tskIdx>.

static ccsCOMPL_STAT lsfTaskDevSchedule
                             ( IN  lsfTASKDEV_DATA *taskDevData,
                               IN  const char      *taskDevName,
                               IN  vltINT32         state,
                               OUT ccsERROR        *error )
    This method schedules the tasks depending on their start state.
    If the <state> is higher or equal the start state, the task is
    activated, otherwise it is terminated.
```

PRIVATE DATA MEMBERS

The data stucture lsfTASKDEV_ENTRY is a sub-class of lsfDEVICE_ENTRY.

```
typedef struct
{
  vltINT32          state;
  vltINT32          subState;
  vltLOGICAL        init;
  vltLOGICAL        simulation;
  vltINT32          timeout;
                                Additional data members
  vltINT32      startState;        Miniumum state for schedule
  vltLOGICAL    ignoreStop;        True when stop is ignored
  vltLOGICAL    first;             True when starting
  vltINT32      tcb,tid;           Task TCB for lccTaskLib
  void         *fctHook[3];        Addresses of function hooks
} lsfTASKDEV_ENTRY;
```

The data stucture lsfTASKDEV_DATA is a sub-class of lsfDEVICE_DATA.

```
typedef struct
{
  void              *ctrlData;
  void              *userData;
  vltINT32           state;
  vltINT32           subState;
  vltLOGICAL         init;
  vltLOGICAL         simulation;
  vltINT32           timeout;
  vltINT32           numTaskDevs;         Number of tasks

  vltBYTES20        *taskDevNames;        Names of the tasks
                                                  (dynamic allocated)
  char              *attrTable[lsfMAX_DEVICES+1];
  lsfTASKDEV_ENTRY *taskDevTable[lsfMAX_DEVICES];   TaskDev data
} lsfTASKDEV_DATA;
```

DATABASE

The class lsfDB_TASKDEV is a sub-class of lsfDB_DEVICE.
It is partially mapped by the data structure lsfTASKDEV_ENTRY.
Each task is represented in the database by a point under the
main point :control:taskDev, which is partially mapped by the data
structure lsfTASKDEV_DATA.

SEE ALSO

lsfDB_DEVICE(5)

3.2.10 lsfSoftDev(3)

NAME

```
lsfSoftDev - Control underlying Software Devices
```

SYNOPSIS

```
#include "lsfSoftDev.h"
```

DESCRIPTION

```
This class provides the methods for the control of software devices.
It implements the default behaviour for all the standard commands.
The standard commands are forwarded to the underlying software devices.
```

PUBLIC METHODS

```
ccsCOMPL_STAT lsfForwardCommand
                             ( IN  lsfSOFTDEV_DATA *softDevData,
                               IN  const char      *softDevName,
                               IN  const char      *command,
                               IN  const char      *param,
                               OUT char            *reply,
                               OUT ccsERROR        *error )
    This method forward a command to the software devices specified
    by <softDevName>. If <softDevName> is NULL, or "all" all the
    software devices will be affected, otherwise only the specified one.
    The method returns the completion status of the command, thus waits
    for all specified software devices to reply.
```

PROTECTED METHODS

```
ccsCOMPL_STAT lsfSoftDevConstructor
                             ( IN  const char      *swdName,
                               IN  lsfCONTROL_DATA *controlData,
                               IN  const char      *softDevNames[],
                               OUT ccsERROR        *error )
    This is the constructor of the softDev object. It affects all the
    software devices given in the list <softDevNames[]> (NULL terminated).
void          lsfSoftDevDestructor
                             ( IN  lsfCONTROL_DATA *controlData )
    This method is the object destructor. All resources are released.

ccsCOMPL_STAT lsfSoftDevInit
ccsCOMPL_STAT lsfSoftDevStandby
ccsCOMPL_STAT lsfSoftDevOnline
ccsCOMPL_STAT lsfSoftDevStop
ccsCOMPL_STAT lsfSoftDevOff
ccsCOMPL_STAT lsfSoftDevExit
ccsCOMPL_STAT lsfSoftDevSimulat
ccsCOMPL_STAT lsfSoftDevStopsim
ccsCOMPL_STAT lsfSoftDevSelftest
ccsCOMPL_STAT lsfSoftDevTest ( IN  lsfSOFTDEV_DATA *softDevData,
                               IN  const char      *softDevName,
                               OUT ccsERROR        *error )
    These methods forward the standard commands to the software
    devices given ib the list <softDevName>.
    If <softDevName> is NULL, or "all" all the software devices will
    be affected, otherwise only the specified one.
```

It updates the state and subState of the group.

PRIVATE METHODS

```
static ccsCOMPL_STAT lsfSendCommand
                             ( IN  const char *srvName,
                               IN  const char *command,
                               IN  const char *param,
                               OUT ccsERROR   *error )
    This method sends the command <command> with the
    parameters <param> to the server process <srvName> of the
    software device.
static ccsCOMPL_STAT lsfWaitReply
                             ( IN  const char *srvName,
                               IN  const char *command,
                               IN  char       *reply,
                               IN  int         timeout,
                               OUT ccsERROR   *error )
    This method waits for the last or error reply to the command
    <command> from the server process <srvName> of the software device.
    The reply is stored in <reply>.
```

PRIVATE DATA MEMBERS

```
The data stucture lsfSOFTDEV_ENTRY is a sub-class of lsfDEVICE_ENTRY.
typedef struct
{
  vltINT32         state;
  vltINT32         subState;
  vltLOGICAL       init;
  vltLOGICAL       simulation;
  vltINT32         timeout;
                               Additional data members
  vltBYTES20  serverName;          Name of the server process
} lsfSOFTDEV_ENTRY;

The data stucture lsfSOFTDEV_DATA is a sub-class of lsfDEVICE_DATA.
typedef struct
{
  void             *ctrlData;
  void             *userData;
  vltINT32          state;
  vltINT32          subState;
  vltLOGICAL        init;
  vltLOGICAL        simulation;
  vltINT32          timeout;
  vltINT32          numSoftDevs;         Number of software devices

  vltBYTES20       *softDevNames;        Names of the software devices
                                                 (dynamic allocated)
  char             *attrTable[lsfMAX_DEVICES+1];
  lsfSOFTDEV_ENTRY *softDevTable[lsfMAX_DEVICES];   SoftDev data
} lsfSOFTDEV_DATA;
```

DATABASE

```
The class lsfDB_SOFTDEV is a sub-class of lsfDB_DEVICE.
It is partially mapped by the data structure lsfSOFTDEV_ENTRY.
Each softDev is represented in the database by a point under the
main point :control:softDev, which is partially mapped by the data
structure lsfSOFTDEV_DATA.
```

SEE ALSO

```
lsfDB_DEVICE(5), lsfDB_SOFTDEV(5)
```

3.3 Database Classes

3.3.1 lsfTemplate.db

```
// ***************************************************************************
// * E.S.O. - VLT project
// *
// * "@(#) $Id: lsfTemplate.db,v 1.27 2000/10/24 08:32:13 vltsccm Exp $"
// *
// * who       when        what
// * --------  ----------  ----------------------------------------------
// * pduhoux   2000-05-18  created
// *

// **************************************************************************
// *
// * This file has been generated by a utility
// *
// *  !!!!!!!!!!!  DO NOT MANUALLY EDIT THIS FILE  !!!!!!!!!!!
// *
// *---------------------------------------------------------------------
// */

//
// Points needed by the application 'pfx'
//
#include "pfxDefines.h"
#include "pfxDeviceList.h"
#include "lsfDB_SERVER.class"
#include "pfxDB_DEVICE.class"
#include "pfxDB_DATA.class"

#ifdef lsfNUM_DEVICES
#if lsfNUM_DEVICES > 0
#include "lsfDB_CONTROL.class"

CLASS "lsfDB_CONTROL" "pfxDB_CONTROL"
BEGIN
    ATTRIBUTE Table deviceTable (lsfNUM_DEVICES)

#ifdef lsfNUM_SIGNAL
#if lsfNUM_SIGNAL > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_SIGNAL_STR
#endif
#endif
#ifdef lsfNUM_MOTOR
#if lsfNUM_MOTOR > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_MOTOR_STR
#endif
#endif
#ifdef lsfNUM_SERIAL
#if lsfNUM_SERIAL > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_SERIAL_STR
#endif
#endif
#ifdef lsfNUM_ENCODER
#if lsfNUM_ENCODER > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_ENCODER_STR
#endif
#endif
```

```
#ifdef lsfNUM_NETWORK
#if lsfNUM_NETWORK > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_NETWORK_STR
#endif
#endif
#ifdef lsfNUM_SOFTDEV
#if lsfNUM_SOFTDEV > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_SOFTDEV_STR
#endif
#endif
#ifdef lsfNUM_TASKDEV
#if lsfNUM_TASKDEV > 0
    ATTRIBUTE  "lsfDB_DEVICE"  lsfDEVTYPE_TASKDEV_STR
#endif
#endif

END
#endif
#endif

CLASS "lsfDB_SERVER" "pfxDB_SERVER"
BEGIN
#ifdef lsfNUM_DEVICES
    ATTRIBUTE int32     numDevices lsfNUM_DEVICES
#if lsfNUM_DEVICES > 0
    ATTRIBUTE "pfxDB_CONTROL" control
    BEGIN
        ATTRIBUTE Table deviceTable (lsfNUM_DEVICES)
    END
#endif
#else
    ATTRIBUTE int32     numDevices 0
#endif
    ATTRIBUTE "pfxDB_DATA" data
END

POINT "pfxDB_SERVER" pfxDB_ROOT
BEGIN
    ALIAS pfxMODULE_NAME
//
// Instanciate all devices
//
#ifdef lsfNUM_DEVICES
#if lsfNUM_DEVICES > 0
//  ATTRIBUTE "<lsfDB_CLASS>"   control:<devType1>:<devName1>
//  ATTRIBUTE "<pfxDB_CLASS>"   control:<devType2>:<devName2>
#endif
#endif

END

// END OF FILE
// ============================================================
```

3.3.2 lsfDB_SERVER(5)

NAME

```
lsfDB_SERVER - Base class for software device application
```

SYNOPSIS

```
#include "lsfDB_SERVER.class"
POINT lsfDB_SERVER ":<myPath>:<myMod>"
BEGIN
  ALIAS "myMod"
END
```

PARENT CLASS

```
lsfDB_DEVICE <-- lsfDB_SOFTDEV <-- lsfDB_SERVER
```

DESCRIPTION

```
The class lsfDB_SERVER is the generic definition of the top point of
a software device. It holds the minimum information required
by "lsf" for the software device management.
```

ATTRIBUTES

```
In addition to those defined in the parent class lsfDB_SOFTDEV:
  monPeriod   : period in ms of the associated monitor task
  numDevices  : number of controlled devices
```

3.3.3 lsfDB_CONTROL(5)

NAME

```
lsfDB_CONTROL - Base class for application control point
```

SYNOPSIS

```
#include "lsfDB_CONTROL.class"

ATTRIBUTE lsfDB_CONTROL control
```

PARENT CLASS

```
BASE_CLASS <-- lsfDB_CONTROL
```

DESCRIPTION

```
The class lsfDB_CONTROL is the generic definition of the
control point.
It contains the attribute <deviceTable> holding the information
relevant to the control of the devices.
The application control point is instanciated from a sub-class
of this generic class. In particular it overloads the number of
devices and adds a sub-point per device type.
```

ATTRIBUTES

```
- TABLE deviceTable(1)
    deviceName : name of the device
    deviceType : type of the device
    startPhase : sequence order when increasing the device state
    stopPhase  : sequence order when decreasing the device state
    simulation : true if device shall be controlled in simulation
    ignored    : true is device shall be ignored
```

CAUTIONS

```
The two fields <startPhase> and <stopPhase> are ignored.
```

3.3.4 lsfDB_DATA(5)

NAME

```
lsfDB_DATA - Base class for application data
```

SYNOPSIS

```
#include "lsfDB_DATA.class"

ATTRIBUTE lsfDB_DATA data
```

PARENT CLASS

```
BASE_CLASS <-- lsfDB_DATA
```

DESCRIPTION

```
This class is the generic definition of the data point.
It is provided to hold the application specific data used e.g.
for monitoring purpose.
The application data point is instanciated from a sub-class
of this generic class. Its definition is application dependent.
```

ATTRIBUTES

```
None
```

3.3.5 lsfDB_DEVICE(5)

NAME

```
lsfDB_DEVICE - Base class for devices
```

SYNOPSIS

```
#include "lsfDB_DEVICE.class"

ATTRIBUTE lsfDB_DEVICE  <aGenericDevice>
```

PARENT CLASS

```
BASE_CLASS <-- lsfDB_DEVICE
```

DESCRIPTION

```
The class lsfDB_DEVICE is the generic definition of a device.
It holds the minimum information required by "lsf" for the
device management. For each type of device a specialized
sub-class is provided that adds specific attributes.
```

ATTRIBUTES

```
lsfDB_DEVICE
  state       : state of the device (= LCC states)
  substate    : sub-state of the device
  initialized : true if device is initialized
  simulation  : true if device shall be controlled in simulation
  timeout     : timeout in seconds for command execution
```

3.3.6 lsfDB_SIGNAL(5)

NAME

```
lsfDB_SIGNAL - Base classes for analogue and digital signals
```

SYNOPSIS

```
#include "lsfDB_SIGNAL.class"

ATTRIBUTE lsfDB_ANALOG  analog
ATTRIBUTE lsfDB_DIGITAL digital
```

PARENT CLASS

```
                                |-- lsfDB_ANALOG
lsfDB_DEVICE <-- lsfDB_SIGNAL <--+
                                |-- lsfDB_DIGITAL
```

DESCRIPTION

```
The class lsfDB_SIGNAL is the generic definition of a signal
(analogue or digital) device. It contains the attribute holding
the information relevant to the control of these devices.
```

ATTRIBUTES

```
The following attributes are mapping for Analog and Digital
signals the configuration parameters expexted by the LCC
functions ioConfigAnalog(3), resp. ioConfigDigital(3).

lsfDB_SIGNAL
  In addition to those defined in the parent class lsfDB_DEVICE:
    deviceName : name of the device "/acroN" for Digital signals
                                    "/aioN"  for Analogue signals
    direction  : signal direction (Input/Output)

lsfDB_ANALOG
    deviceName : overloaded to "/aio"
  In addition to those defined in the parent class lsfDB_SIGNAL:
    channel    : channel number
    gain       : amplifier gain
    conversionFactor: floating point value
    lowerRange : signal range
    higherRange:
    simValue   : simulation value
    signalA    : associated database point (lccANALOG_SIGNAL)

lsfDB_DIGITAL
    deviceName : overloaded to "/acro"
  In addition to those defined in the parent class lsfDB_SIGNAL:
    level      : active Low/High
    startBit   : lower bit [0-63]
    numBits    : number of bits mapping the signal
    simValue   : simulation value
    signalD    : associated database point (lccDIGITAL_SIGNAL)
```

SEE ALSO

```
LCC User Manual for Signal Handling
ioConfigAnalog(3), ioConfigDigital(3)
Digital I/O Board User Manual, and acro driver User Manual
Analog  I/O Board User Manual, and aio  driver User Manual
```

3.3.7 lsfDB_MOTOR(5)

NAME

```
lsfDB_MOTOR - Base class for motor
```

SYNOPSIS

```
#include "lsfDB_MOTOR.class"

ATTRIBUTE lsfDB_MOTOR <aMotorDevice>
```

PARENT CLASS

```
lsfDB_DEVICE <-- lsfDB_MOTOR
```

DESCRIPTION

```
The class lsfDB_MOTOR is the specialized class for motors of
any kind.
At instanciation, the class of the attribute <motor> shall be
overloaded with the sub-class corresponding to the motor.
```

ATTRIBUTES

```
In addition to those defined in the parent class lsfDB_DEVICE:
  motor : motor branch (from motMOTOR)
```

SEE ALSO

```
motor.db(5), motMOTORS(5)
```

3.3.8 lsfDB_SERIAL(5)

NAME

lsfDB_SERIAL - Base classes for serial communication links

SYNOPSIS

```
ATTRIBUTE lsfDB_SERIAL com
ATTRIBUTE lsfDB_RS232  com1
ATTRIBUTE lsfDB_RS422  com2
ATTRIBUTE lsfDB_RS485  com3
```

PARENT CLASS

```
                                 |-- lsfDB_RS232
lsfDB_DEVICE <-- lsfDB_SERIAL <--+-- lsfDB_RS422
                                 |-- lsfDB_RS485
```

DESCRIPTION

```
The class lsfDB_SERIAL is the generic definition of a serial
communication device. It contains the attribute holding the
information relevant to the control of these devices.
The 3 sub-classes lsfDB_RS232, lsfDB_RS422 and lsfDB_RS485 are
specialized classes for the control of RS232, RS422 and RS485 resp.
communication links based on these protocols.
```

ATTRIBUTES

```
 lsfDB_SERIAL
   In addition to those defined in the parent class lsfDB_DEVICE:
   deviceName  : name of the device "/iser0-9" or "/tyCo/1-3"
   protocol    : number of the RS protocol [232, 422 or 485]
   for the following attributes, see iserDrv(3) and VxWorks
   manuals for tyCo devices
   baudRate    :
   dataBits    :
   stopBits    :
   parity      :
   handShake   :
   rxMode      :
   bufferLength:

lsfDB_RS232, lsfDB_RS422 and lsfDB_RS485: these sub-classes overload
  the attributes ’protocol’ and ’handShake’.
```

SEE ALSO

iserDrv(3), VxWorks tyLib(1), ttyLib(1) and related

3.3.9 lsfDB_TASKDEV(5)

NAME

```
lsfDB_TASKDEV - Base classes for task devices
```

SYNOPSIS

```
#include "lsfDB_TASKDEV.class"

ATTRIBUTE lsfDB_TASKDEV <aTaskDevice>
```

PARENT CLASS

```
lsfDB_DEVICE <-- lsfDB_TASKDEV
```

DESCRIPTION

```
The class lsfDB_TASKDEV is the specialized class for tasks
devices to be controlled by the application.
```

ATTRIBUTES

```
lsfDB_TASKDEV
  In addition to those defined in the parent class lsfDB_DEVICE:
  startState  : minimum state for task activation
  timerNum    : timer number for period
                   -1 : use system call taskDelay()
                1 - 4 : use TIM timer
  period      : task period in milli-seconds
  priority    : task priority
  stackSize   : task stack size in kilo-bytes [kB]
```

3.3.10 lsfDB_SOFTDEV(5)

NAME

lsfDB_SOFTDEV - Base classes for software devices

SYNOPSIS

```
#include "lsfDB_SOFTDEV.class"

ATTRIBUTE lsfDB_SOFTDEV <aSoftwareDevice>
```

PARENT CLASS

```
lsfDB_DEVICE <-- lsfDB_SOFTDEV
```

DESCRIPTION

The class lsfDB_SOFTDEV is the specialized class for Software devices to be controlled by the application.

ATTRIBUTES

```
lsfDB_SOFTDEV
  In addition to those defined in the parent class lsfDB_DEVICE:
  serverName  : name of the software device server
```

3.4 Include Files

3.4.1 lsfDefines.h

```
#ifndef LSF_DEFINES_H
#define LSF_DEFINES_H
/*******************************************************************************
* E.S.O. - VLT project
*
* "@(#) $Id: lsfDefines.h,v 1.27 2000/10/24 08:32:16 vltsccm Exp $"
*
* who       when        what
* --------  ----------  ----------------------------------------------
* pduhoux   2000-04-10  created
*/

/************************************************************************
 *
 *----------------------------------------------------------------------
 */

/*
 * Constants
 */
#define _eccs_tostr(a) #a
#define _eccs_tostr_pass2(a) _eccs_tostr(a)

#define __FILE_LINE__  __FILE__ ":" _eccs_tostr_pass2(__LINE__)

#define lsfMODULE_ID    "lsf" /* module name */
#define lsfMODULE_NAME   lsfMODULE_ID
#define lsfMODULE_TITLE  "LCU Server Framework"
#define lsfLOG_ID        100

#define lsfMAX_DEVICES          32

/*
 * Database
 */
#define lsfDB_CONTROL_POINT    ":control"
#define lsfDB_DATA_POINT       ":data"

#define lsfDB_SIGNAL_POINT     ":signal"
#define lsfDB_MOTOR_POINT      ":motor"
#define lsfDB_SERIAL_POINT     ":serial"
#define lsfDB_ENCODER_POINT    ":encoder"
#define lsfDB_NETWORK_POINT    ":network"
#define lsfDB_SOFTDEV_POINT    ":softdev"
#define lsfDB_TASKDEV_POINT    ":taskdev"

#define lsfALL_DEVICES         "all"

#define lsfDEVTYPE_SIGNAL_STR     "signal"
#define   lsfDEVTYPE_ANALOG_STR   "analog"
#define   lsfDEVTYPE_DIGITAL_STR  "digital"
#define lsfDEVTYPE_MOTOR_STR      "motor"
#define lsfDEVTYPE_SERIAL_STR     "serial"
#define   lsfDEVTYPE_TYCO_STR     "tyCo"
#define   lsfDEVTYPE_RS232_STR    "rs232"
#define   lsfDEVTYPE_RS422_STR    "rs422"
```

```
#define   lsfDEVTYPE_RS485_STR    "rs485"
#define lsfDEVTYPE_ENCODER_STR    "encoder"
#define   lsfDEVTYPE_IK320_STR    "ik320"
#define lsfDEVTYPE_NETWORK_STR    "network"
#define   lsfDEVTYPE_ETHERNET_STR "ethernet"
#define   lsfDEVTYPE_NET01_STR    "net01"
#define   lsfDEVTYPE_ATM_STR      "atm"
#define lsfDEVTYPE_SOFTDEV_STR    "softdev"
#define lsfDEVTYPE_TASKDEV_STR    "taskdev"

#define lsfDEVICE_NUM_TYPES 7
#define lsfDEVICE_SIGNAL    1
#define lsfDEVICE_MOTOR     2
#define lsfDEVICE_SERIAL    3
#define lsfDEVICE_ENCODER   4
#define lsfDEVICE_NETWORK   5
#define lsfDEVICE_SOFTDEV   6
#define lsfDEVICE_TASKDEV   7

/* lsfDEVTYPE_<device> = (0x100 << (lsfDEVICE_<type>-1)) + <kind> */
#define lsfDEVTYPE_MASK       0xFF00
#define lsfDEVTYPE_UNKNOWN    0
#define lsfDEVTYPE_SIGNAL     0x0100
#define   lsfDEVTYPE_ANALOG   0x0101      /* = lsfDEVTYPE_SIGNAL | ioANALOG */
#define   lsfDEVTYPE_DIGITAL  0x0102      /* = lsfDEVTYPE_SIGNAL | ioDIGITAL */
#define lsfDEVTYPE_MOTOR      0x0200
#define lsfDEVTYPE_SERIAL     0x0400
#define   lsfDEVTYPE_RS232    0x0401
#define   lsfDEVTYPE_RS422    0x0402
#define   lsfDEVTYPE_RS485    0x0404
#define lsfDEVTYPE_ENCODER    0x0800
#define   lsfDEVTYPE_IK320    0x0801
#define lsfDEVTYPE_NETWORK    0x1000
#define   lsfDEVTYPE_ETHERNET 0x1001
#define   lsfDEVTYPE_NET01    0x1002
#define   lsfDEVTYPE_ATM      0x1004
#define lsfDEVTYPE_SOFTDEV    0x2000
#define lsfDEVTYPE_TASKDEV    0x4000

#define lsfSTATE_OFF     1
#define lsfSTATE_LOADED  2
#define lsfSTATE_STANDBY 3
#define lsfSTATE_ONLINE  4

#define lsfUNCHANGED -99

#define lsfSUBSTATE_IDLE         0
#define lsfSUBSTATE_ERROR        1
#define lsfSUBSTATE_TIMEOUT      2
#define lsfSUBSTATE_INITIALIZING 3
#define lsfSUBSTATE_ACTIVE       4
#define lsfSUBSTATE_MONITORING   5
#define lsfSUBSTATE_MOVING       6
#define lsfSUBSTATE_WAITING      7

#define lsfATTR_NAMES \
    "state","substate","simulation","initialized","timeout"

#define lsfDB_STATE          0
#define lsfDB_SUBSTATE       1
```

```
#define lsfDB_SIMULATION     2
#define lsfDB_INITIALIZED    3
#define lsfDB_TIMEOUT        4
#define lsfDB_LAST           5

#endif /*!LSF_DEFINES_H*/
```

4 Installation Guide

This chapter is a guide for the installation of Software Devices in the VLT environments. It is built in 2 sections resp. for the WS and the LCU parts. The installation of the Software Device on the Workstation environment depends on the requirements, namely whether the database must be scanned from the LCU environment. In that case, the database shall contain the branch mirroring the LCU database branch for scanning.
It is assumed that the WS environment ***wsEnv*** and LCU environment ***lcuEnv*** have been successfully created and configured for ***lcuEnv*** to report to ***wsEnv***.

4.1.1 WS Environment

The configuration of the WS environment is made in 3 steps:

- Edit the file $VLTDATA/ENVIRONMENTS/*wsEnv*/dbl/DATABASE.db
  Add the following 2 lines outside any BEGIN ... END block:

  ```
  #define appDB_ROOT "<absolute path>"
  #include "app.db"
  ```

- Edit the file $VLTDATA/ENVIRONMENTS/*wsEnv*/dbl/Makefile
  Add the following directive:

  ```
  USER_INC = -I${INTROOT}/vw/include
  ```

- Generate the database, initiatialize and start the environment:

  ```
  > cd $VLTDATA/ENVIRONMENTS/wsEnv/dbl
  > make db
  > vccEnvInit -e wsEnv
  > vccEnvStart -e wsEnv
  >
  ```

4.1.2 LCU Environment

The configuration of the LCU environment is made in 4 steps:

- Edit the file $VLTDATA/ENVIRONMENTS/*lcuEnv*/dbl/DATABASE.db
  Add the following 2 lines outside any BEGIN ... END block:

  ```
  #define appDB_ROOT "<absolute path>"
  #include "app.db"
  ```

- Generate the database:

  ```
  > cd $VLTDATA/ENVIRONMENTS/lcuEnv/dbl
  > make db
  >
  ```

- Edit the file $VLTDATA/ENVIRONMENTS/*lcuEnv*/devicesFile
  9. Increment the number of devices in the line:

     ```
     <ATTRIBUTE>: deviceTable <TYPE>: Table <REC>: 0 - N <FIELDS>: 0 - 5
     ```

     where *N* must be incremented
  10. Add the following entry:

     ```
     "app"  "appServer" 1 0 0 1
     ```

  <u>IMPORTANT NOTE:</u> Any Software Device that is to be controlled from another Software Device <u>must not</u> be registered in this list; the respect of this rule is essential for the system to behave correctly: since the standard commands are forwarded from the co-

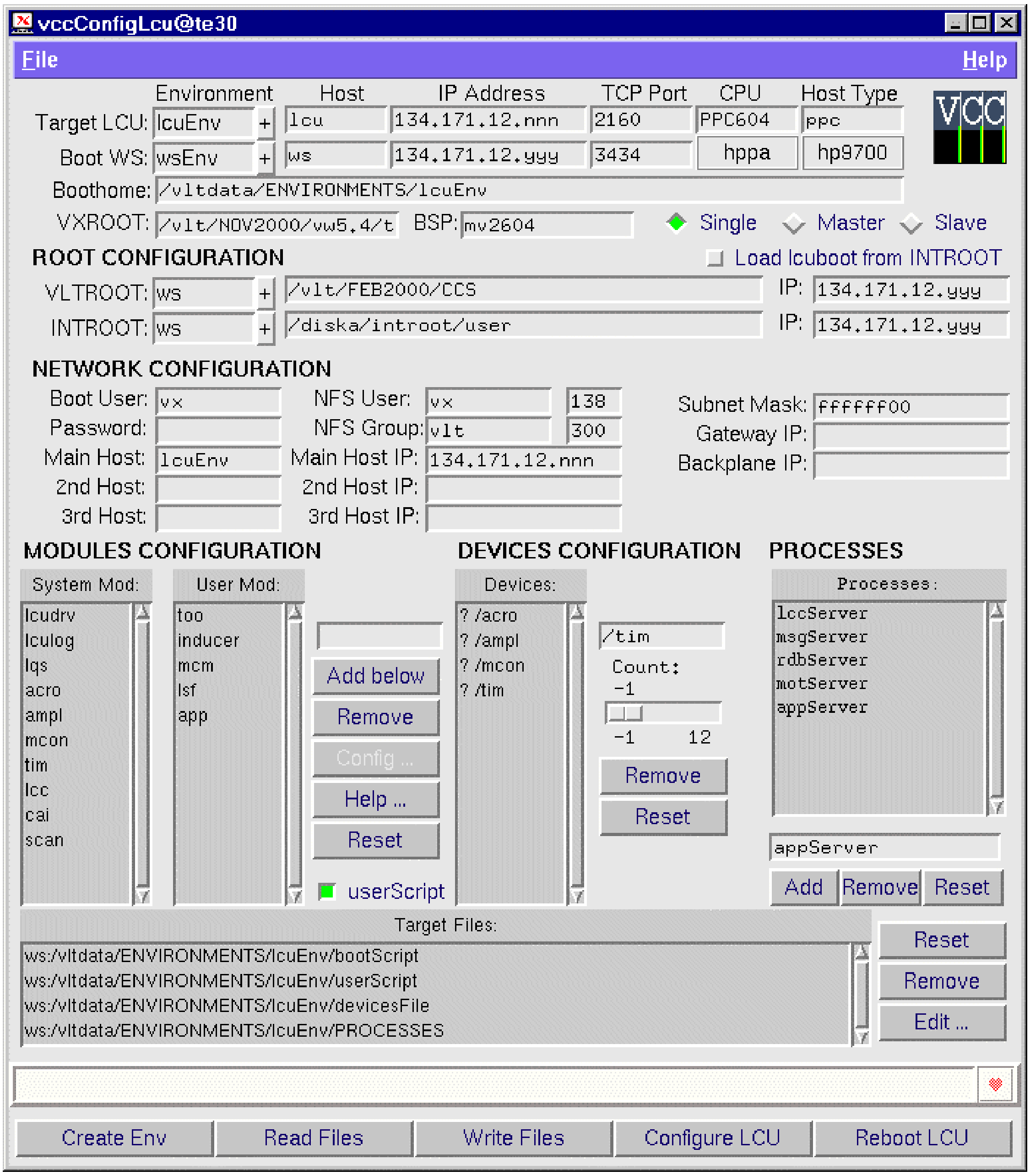
ordinating Software Device to the sub-ordinated ones, any standard command sent to LCC (process `lccServer`) would be forwarded to all the registered Software Devices found in this list, but as well from the co-ordinating Software Devices to the sub-ordinated ones resulting in non predictable effects.

- Configure the LCU bootScript and reboot the LCU:

```
> vccConfigLcu lcuEnv &
>
```

In the LCU Configuration panel:

1. Update the user module list that shall contain:
   `lsf` `and` **`app`** in this precedence order.
   For applications that do use motors, the module **`mcm`** must be added.
2. Add the required driver devices,
3. Add the process **`appServer`**.
4. Click on the button **`Write Files`** to save the configuration
5. Click on the button **`Reboot LCU`** to boot the LCU



4.1.3 Scan Links

After both environments have been successfully started, the scan links can be configured. Edit the file `app/ws/config/app.scan` to add all the links required by the application.

<u>Example:</u>

```
> cd app/ws/config
> vi app.scan
  ...
> cd ../src
> make install
  ...
  . . . installation done
>
> scanLinks -f $INTROOT/config/app.scan -l wsEnv -e lcuEnv -c -E -r <path>
>
```

where *<path>* is the absolute path of the database root point of the application on the WS.

The WS database will now be kept aligned with the LCU database by means of the scan system. In particular, the module state and sub-state are updated on change (mode SRBX).

4.1.4 Verification

On the Workstation, invoke the panel `appgui`.
The fields `State`, `Substate` and `Mode` shall not be shaded (indication for bad data quality, hence scan troubles). They shall indicate `LOADED` (in Orange), `IDLE` (in Green) and `NORMAL` (in Green) resp.
Send the command `VERSION`.
The Software Device is ready for use, however it might be necessary to tune the configuration of the devices (e.g. motors or signals).

4.1.5 Database configuration

This last section is dedicated to the handling of the database configuration file of the application (`app/ws/config/app.dbcfg`).
As mentioned above, this file is generated by the utility `lsfConfig(1)`. It contains then the minimum necessary configuration attributes for the software device to startup. After the tuning of the hardware has been completed, i.e. all devices are properly configured and initialize and perform as required, the database shall be saved into the file `app/ws/config/app.dbcfg` by means of the utility `lsfBackup(1)`.

Example:

```
> cd app/ws/config
> lsfBackup -e lcuEnv -m app
Generating input file './app.inp' ... done
Performing database backup into 'app.dbcfg' ... done
> cd ../src
> make install
 ...
 . . . installation done
>
```

Last modified: Thu Sep 28 10:32:15 METDST 2000

___oOo___