MARTe/BaseLib2

Software Design Analysis

Code Development Guide

Antonio Barbalace

June 2009

Contents

List of Figures

List of Tables

Part I

Welcome

Introduction

BaseLib/MARTe are togheter a complete software framework to run a distributed control code on different architectures and operating systems. BaseLib/MARTe are not only a middleware but include also a collection of different components, i.e. computational blocks.

A control system is not more than the interconnection of computational blocks, devices that read signals from the environment and then write them out of the digital system. If a digital control system can be completely designed with blocks included in the BaseLib/MARTe library the implementation is straightforward and can require only a couple of hours.

The whole project can be logically splitted in BaseLib, MARTe and the components collection. BaseLib offers the basic data structures, an architecture and OS independent API, an object accounting and tracing infrastructure, a serialization and reflection faciliy and all key mechanisms to run a control system; MARTe lets the control algorithm run.

The componets collection is a growing set of algorithmic blocks (GAMs), device drivers adaption blocks (IOGAMs) and device drivers (GACQMs). All the code is written in C/C++. Figure 1 is a complete schema of the software framework we are going to analyse in this text.

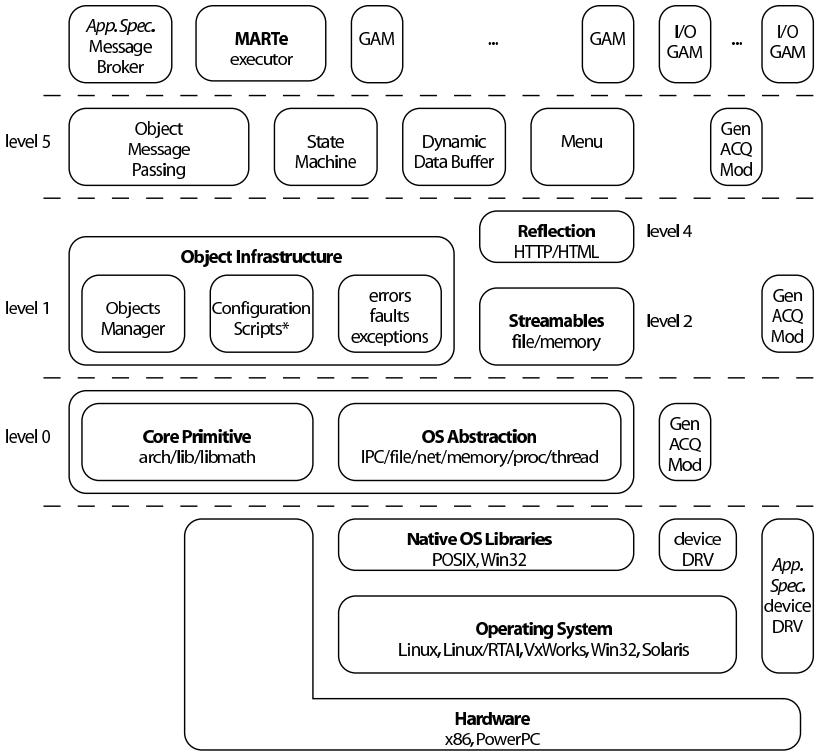

Figure 1: BaseLib and MARTe schema

The cited Figure highlights the supported operating systems, those are: Microsoft Windows© 32bit, Linux, Linux/RTAI, Wind River VxWorks© and Sun Microsystems Solaris©. The supported architectures are: Intel x86 and IBM/Motorola PowerPC (now Freescale), BaseLib/MARTe worked also on SPARC machines but it is not recently tested on. A list of supported I/O and timing devices (ADC and DAC boards) is not available yet.

BaseLib/MARTe is a rapid deve lopment control software that can be suitable to automation industry, robot control and manufaturing processes supervision. In those markets we can find other software products, never used in fusion research, those softwares comes from differents experience but most of them lead to the same functionalities and ideas on which BaseLib/MARTe was built.

The oldest project we can find on Internet is **OSACA** (http://www.osaca.org), i.e. Open System Architecture for Controls within Automation systems, focuses on industrial control, is not open-source and probably not manteined yet. There are two government founded project: **TORERO**, TOtal life cycle web-integrated control design architecture and methodology for distributed control systems in factory automation (http://www.uni-magdeburg.de/iaf/cvs/torero/), and **OMAC**, the Open Modular Architecture Controller (http://www.isd.mel.nist.gov/projects/). Fully open source university founded projects are: **OROCOS** (http://www.orocos.org) the Open RObot COntrol Software; **MCA2** (http://www.mca2.org/) the Modular Controller Architecture (Version 2); **MICROB** (http://www.robotique.ireq.ca/microb/en/index.html) the Modules Intégrés pour le Contrôle de ROBots and **OASYS** (http://www-lar.deis.unibo.it/oasys/) the Open source software for industrial Automation and distributed SYstemS (probably part of the TORERO alliance).

The famous **EPICS** framework (the TANGO framework can be considered as well) doesn't appear in the previous list because it doesn't provide all the basic architectural and design requirement a generic control systems needs to be setup. EPICS, as well as **CORBA** in the Enterprise Market, is a middleware: it just defines a common protocol to lets different machines (equipped with potentially different OS and processor) communicate. Another middleware used by many automation companies on only Microsoft enabled products is **OPC** (http://www.opcfoundation.org) the OLE for Process Control, it exploits Microsoft's OLE/COM/DCOM technologies.

Various instances of BaseLib/MARTe running in a distributed system can communicate using object messaging, application specific message brokers can also be written to handle 3rd party messages (Figure 1). There is an ongoing effort to turn BaseLib/MARTe in an EPICS enabled product.

0.0.1 Project Organization

The whole project is structured in at minimum three packages (directories):

- MakeDefaults
- BaseLib (or BaseLib2)
- MARTe

Components (i.e. GAMs) are allotted between the *BaseLib* directory and the *MARTe* directory, other GAMs's directory can be present in your project. The directory *MakeDefaults* group togheter all files to build the software, it supports the Microsoft build system as well as the GNU Makefile utility. The directory *BaseLib* holds all BaseLib files and subdirectories, obviously the directory *MARTe* contains all MARTe files and subdirs, that are presented at the end of the chapter.

This directories represent the whole software framework, no external library or software is required, only a working supported Operating System. This means no external dependencies, no dependecies

checking and a faster time to a ready to use system. Other projects use differents 3d party libraries and its a pain for a user to put all libraries working togheter. The best solution in this sense is to distribuite the 3d party libraries togheter with the software package, this option can be considered in the future to speed up upgrades of the BaseLib/MARTe code (imagine to add XML support).

To compile the whole project two steps are needed: first you must compile BaseLib and then it is possible to build MARTe. When the libraries are ready you need a configuration file to see something running.

0.0.2 Component Object Model

Most software products nowadays utilize several design patterns; BaseLib/MARTe explicitly uses the Container (Composite), the Iterator, Command and State design patterns (last threes of these are considered by the GAMMA behavioral patterns). Also the Observer, Prototype, Adapter, Bridge and Decorator design patterns are used but the implementation comes naturally, without any previous class's design.

Instead, the development of BaseLib/MARTe was done following the Component Oriented Programming practice (well known as Component Object Model). Following such paradigm all communications between objects use a well known interface, the source code is infact rich of interface definitions.

BaseLib/MARTe is built on different components, these usually lead to be developed in single classes, i.e. one component is implemented within one C++ class. These is especially true for the computational blocks that realize a control system from an algorithmic point of view. A control system can be first designed using a Block Diagram Modeller IDE (Matlab Simulink©, Scicos, NI LabView© or Ptolemy/Kepler) and then converted for execution in BaseLib/MARTe. Each computational block of the diagram will be mapped to a software component called Generic Application Module (GAM); GAMs are then connected via a memory bus component that offers some interfaces but also requires to be compliant to other interfaces. The definition of incoming and outcoming interfaces is usually referred to Connection Oriented Programming and is a particular programming style that is especially useful when aiming for Component Oriented Programming. Figure 2 show how connection can be setup between components.

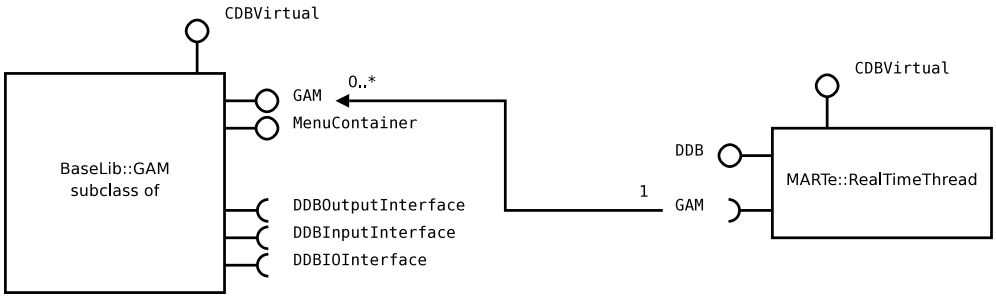

Figure 2: Components connections in BaseLib/MARTe

Such Figure points out the connection between a `MARTe::RealTimeThread` component and many `BaseLib::GAM` subclass components, note the multiplicities at each ends of the association. A `BaseLib::GAM` subclass component, or simply a GAM, holds receptacles for `DDBInputInterface`, `DDBOutputInterface` and `DDBIOInterface`. Such interfaces lets a GAM produce and consume datas on the common memory bus called Dynamic Data Buffer (DDB). The `CDBVirtual` interface lets components load and save its configuration from and to a configuration file (in a human readable format). The `MenuContainer` interface lets components export a menu user interface to enable users to see and change component's parameter in real time.

Most of the code in the project that doesn't extends or doesn't implement a GAM is infrastructural and lets GAMs load, connect between them, run and save is state (persistence), the exception is MARTe that is a scheduling entity of all GAMs. Figure 3 illustrate this concept. Note that this doesn't mean that only GAMs are components, all the software is structured using the Component Object Model.

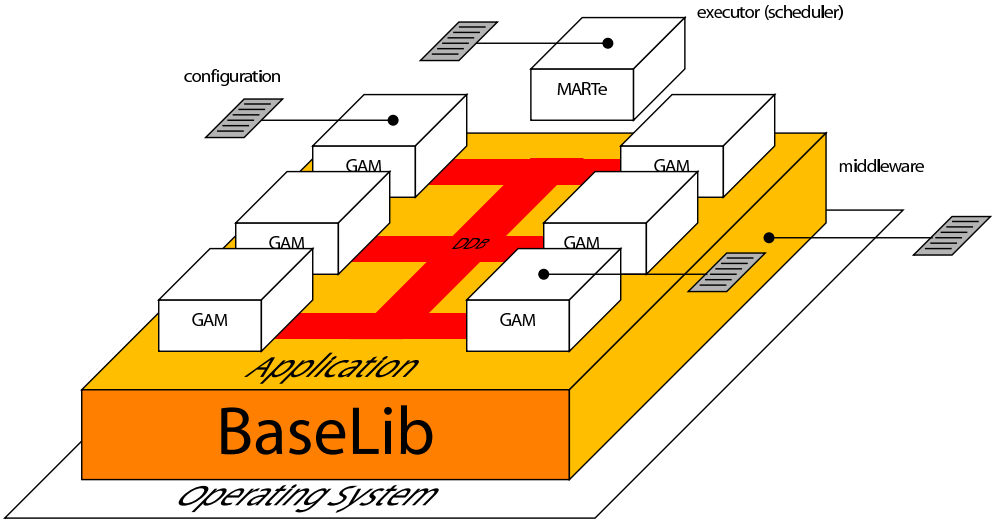

Figure 3: BaseLib and MARTe schema in 3D

Referring to the same picture is simple to understand what a user application running in BaseLib/-MARTe is: **a collection of interconnected components initialised via a configuration file**.

If components for your control system application are just written no understanding of the entire software framework is needed and also no programming skills are required because you can implement the control system using only a human readable/writable configuration file (similar to the Microsoft's `ini` files).

0.0.3 Reflection, Persistence, Garbage Collection and Messages

A big difference between the most used C object oriented version (C++) and the Sun Object Oriented language Java is the lack of reflection, persistence (serialization) and garbage collection of the former. These capabilities enable Java to manage every aspect of objects lifecycle, from the creation to the destruction providing also a runtime inspection. Both C++ and Java supply runtime introspection (C++ by means of RTTI).

Smalltalk and Objective C languages have built-in reflection besides introspection; such languages add Object Messaging. To get an object to do something, you send it a message telling it to apply a method or to do an activity. Programming by messages instead using method calls ensure inter-thread communication safety without holding locks.

BaseLib/MARTe as a C/C++ framework supplies to the user a complete solution that introduce in C/C++ language an esay to use reflection, persistence, garbage collection and object messaging, all those capability work on different compilers and different operating systems. Looking back at Figure 1 there is a block named *Object Infrastructure*, this block is deepened in Figure 4 and implements the

core functionalities for object handling and inspection. Those object level features are not common to a component based software that usually takes into account component-alike properties (this is the behavior of the previous cited OROCOS control framework). Such added capabilities add at minimum another level of complexities to the schema in Figure 3 where for example there is no object message loop.

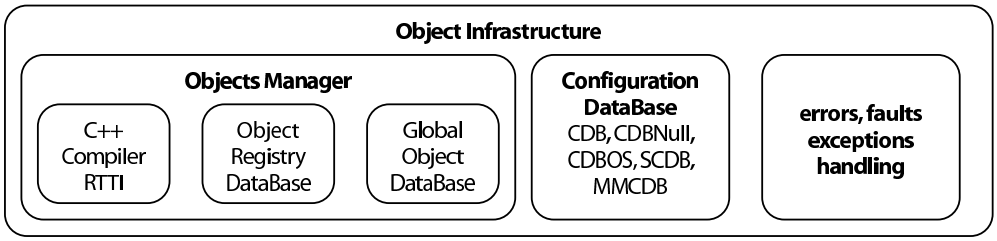

Figure 4: BaseLib Object Infrastructure

Figure 4 highlights the different entity componing the *Object Infrastructure*: an object manager built on top of the C++'s compiler RTTI, an Object Registry DataBase (ORDB), i.e. a list of all objects type currently loaded in the system; a Global Object DataBase (GODB), i.e. a list of all currently instantiated objects; a configuration utility (the CDB) that lets load and save object's attributes (i.e. implement the serialization) and an error management unit.

Exploiting the RTTI facility and language macros the ORDB and the GODB implement reflection and garbage collection. Garbage collection is done by refernce counting. External live reflection capability is offered via an HTTP server at highers level (many industrial boards nowadays lets the user inspect it via a web browser).

The CDB exploiting ORDB and GODB adds serialization. The serialization in BaseLib/MARTe is done by each object, i.e. only the single object is aware of its attributes (or parameters). The serialization is not done on human unreadable format but in a simple format that lets a user write it's own file, a configuration file is a human readable serialization of objects. A whole control system can be written using only a single configuration file without a single line of C/C++.

Object Messaging is achieved by the *Object Message Passing* infrastructure in a higher level of the framework (have a look at Figure 1). This entity will be integrated in the Object Infrastructure.

0.1 BaseLib

BaseLib is a library, i.e. a collection of codes. BaseLib relay on very few global data structures loaded at initialization time, it is the user that loads the framework via a function call that invoke all the startup process.

The startup process (CREATION state) of BaseLib reads a configuration file (also in memory) then creates all objects in such file with the correct attribute's values (exploiting serialization). If the creation finishes without any problem the framework is in the RUNTIME state, in this state MARTe is running. If MARTe chooses to exit move all the system in the DELETION state, the last state before application end. In each state BaseLib can moves to the ERROR state. The whole BaseLib state transitions are depicted in Figure 5; BaseLib doesn't know about its internal state unless it is in ERROR, BaseLib' states are just an abstraction to understand its behaviour.

BaseLib's core source code is partitioned right now in seven different directories, each one is a *level* starting from *level0* up to *level6*. Code in a directory within an higher *level* depends on code in low

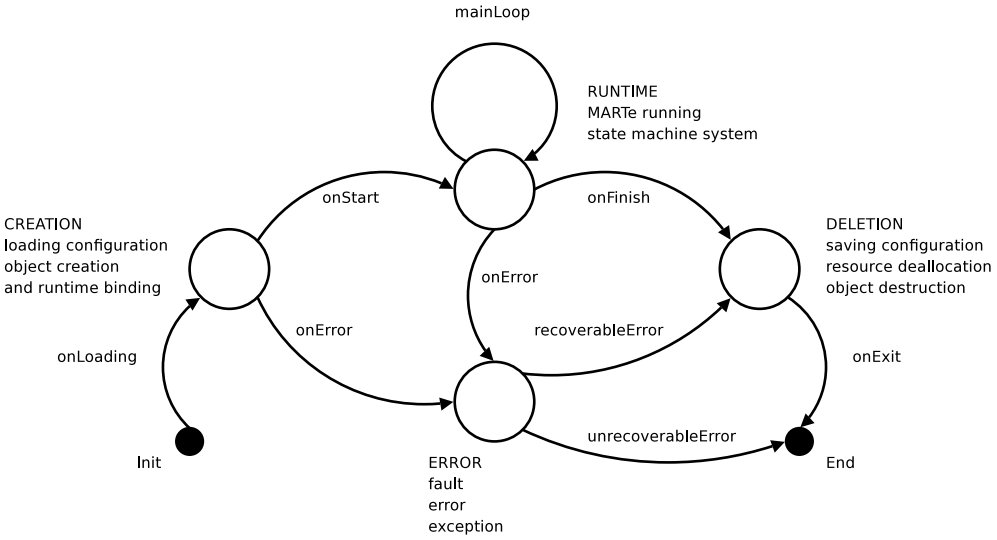

Figure 5: BaseLib's main states

level directories like in an hierarchy relationship.

BaseLib contains also other code, it holds a collection of GAMs (common and wide used ones), some testing applications and useful debugging and logging tools. All that in four different directories: *LoggerService*, *GAMs*, *BaseLibTests*, *BaseLibTools*.

In this work only *level* directories will be analised. At the end of reading it will be easy to understand that the subdivision by compilation dependence doesn't ease the understanding of the project (also if can be a good choice for debugging the project). Table 1 summarizes which component it is possible to find in each level. The CDB's code for example is spreaded between *level0*, *level1*, *level3* and *level6*. Same work must be done in these direction to ease code comprehension and modularity.

level 0	OS Abstraction Core Primitives
level 1	Garbage Collection ObjectRegistryDataBase (ORDB) GlobalObjectDataBase (GODB) ConfigurationDataBase (CDB)
level 2	streams ConfigurationDataBase (CDB)
level 3	ConfigurationDataBase (CDB) ConfigurationDataBase OutputStream (CDBOS) parser
level 4	HTTP, HTML protocol libraries
level 5	Dynamic Data Buffer (DDB) Generic Acquisition Module (GAM) Messages State Machine Menu
level 6	Memory Mapped CDB (MMCDB) maths objects

Table 1: BaseLib levels

0.2 MARTe

MARTe library is a collection of GAMs, IOGAMs, GACQMs and an executor (or dispatcher). Such executor is a thread that run consecutively a set of GAMs and IOGAMs; the set that is currently running depends on the state of the plant system and the controller machine. There are basically four sets: *initialising*, *offline*, *online* and *safety*. In Figure 6 the states of the plant system are depicted; usually the *initialising* set is executed only during the INIT state, the *online* set is executed during PREPULSE, PULSE and POSTPULSE; *offline* set is executed in IDLE and WAITINGPRE states. When some error occurs the currently running set is substitued with the *safety* set.

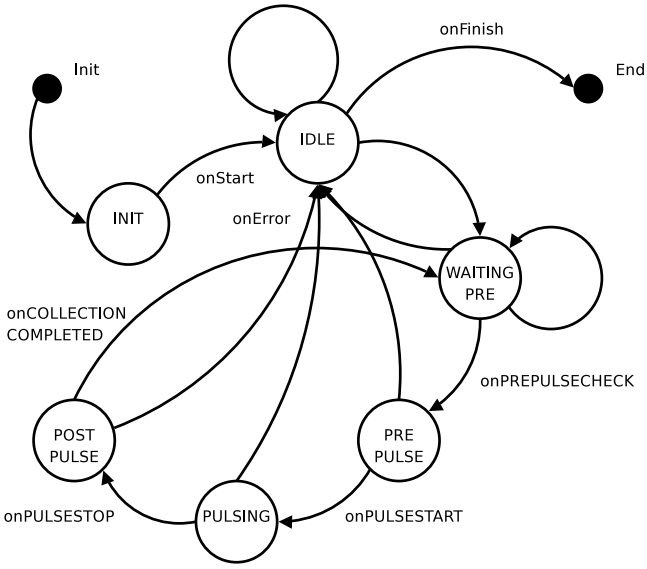

Figure 6: MARTe States

MARTe originates as the first application built on BaseLib library. Some components, like the *RealTime Thread* (the executor), were added because not in BaseLib, MARTe is also the configuration file that request the instantiation of a precise set of components. Figure 7 depict the different componets MARTe require to be instantiated.

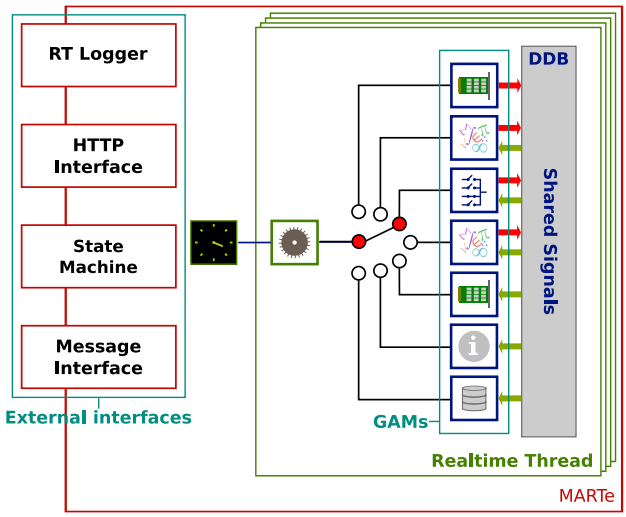

Figure 7: MARTe Components

In such Figure there are at least three different components that come from BaseLib: the *HTTP server*, the *Messaging* infrastructure and an *external state machine* that communicate with the internal one. In the Figure is not depicted the communication library that connects MARTe to plant system rendering the application a part of the whole distributed system. The communication library is included in MARTe, right now there is only CODASLib that supports the JET communication protocol; forthcoming libraries will be MDSplusLib and EPICSLib.

MARTeSupportLib	
GAMs	EventCollection, WaveformCollection, DataCollection WebStatistics, Statistics Hysteresis, Noise, CurrentControl
IOGAMs	ATCA, MPV956, MPV922 ForeHE, Interphase, LinuxSocket (UNIX), WinSock WinTimer, LinuxTimer (UNIX), VxWorksTimer, HRT
CODASLib **MDSplusLib*** **EPICSLib***	

Table 2: MARTe directory tree

0.3 Remarks

TODO

Part II

BaseLib

Chapter 1

BaseLib Level 0

BaseLib Level0 is a mixture of different classes that must provide the developer with a complete system abstraction. In fact such abstraction stops at the operating system level without offering device driver's API. Each device driver must be developed separately for a specific architecture and OS.

To better understand what services there are in BaseLib Level 0 the following argomentation subdivid it in a logical way extrapolating these sections:

- Architectures
- Lists
- IPC
- Network
- Files, Streams
- Errors, Exceptions
- Memory
- Processes, Threads
- Mathematic

BaseLib2 was already ported to the following OS: OS/2©, VxWorks©, Linux, Linux/RTAI, Solaris© and MS Windows©.

1.1 Architectures

This group of classes (Figure 1.1) adds the very low level support to the library, addressing architecture and hardware support, i.e. processor assembly instructions and special processor register functions. The Architecture group include the following classes:

- Processor
- ProcessorType
- LoadableLibrary
- RTAILoader
- Atomic

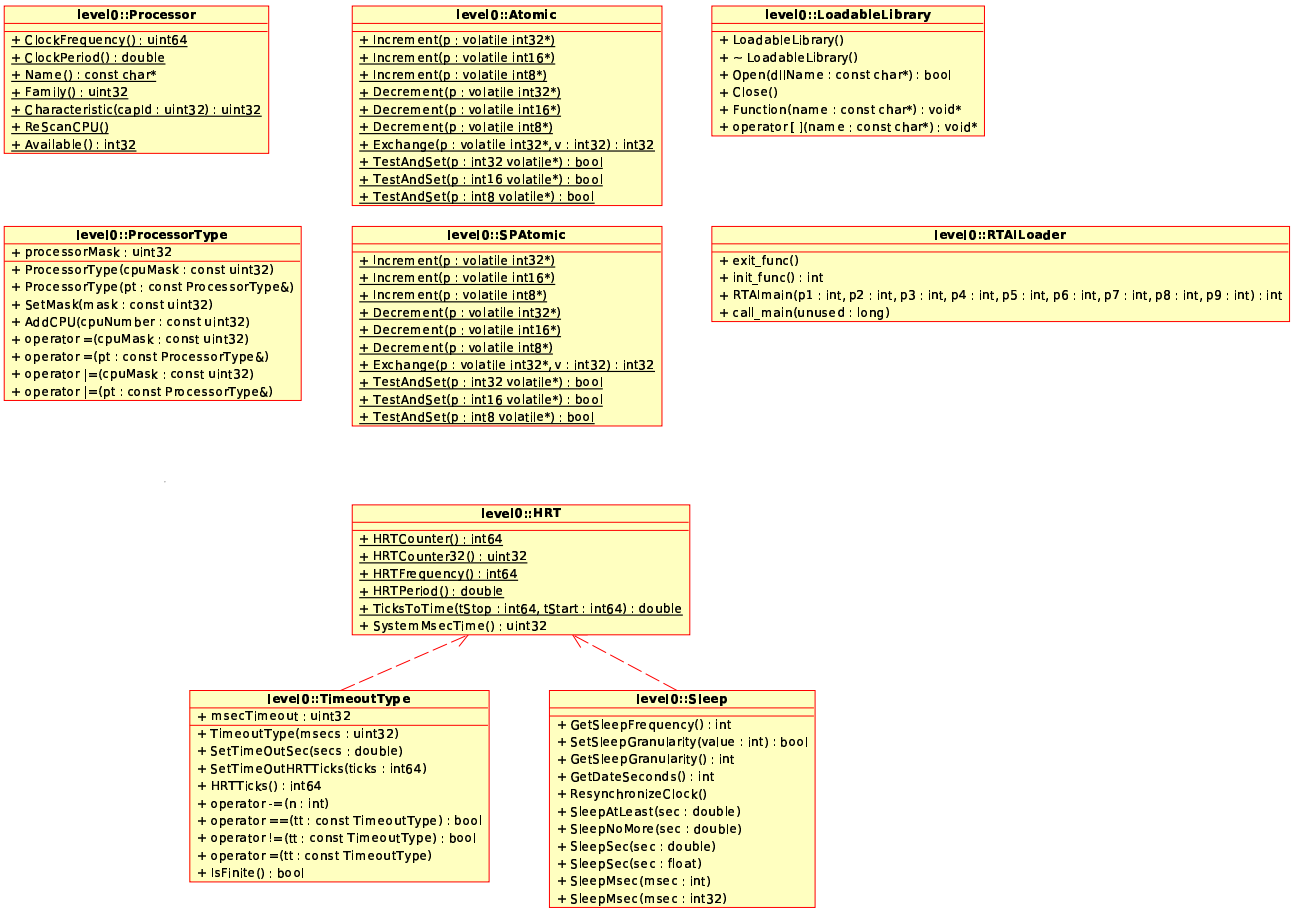

Figure 1.1: BaseLib Level0 Architecture classes

- SPAtomic
- HRT (High Resolution Timer)
- TimeoutType
- Sleep

Processor

`[Processor.h, Processor.cpp]`
The Processor class lets you discover how many processor there are on the machine and which kind of processor are and it's characteristics like the clock and the family name.

```
public:
   static uint64 ClockFrequency();
   static double ClockPeriod();

   static const char *Name();
   static uint32 Family();
   static uint32 Characteristic(uint32 capId);

   static int32 Available();
   static void ReScanCPU();
```

All methods are static; `ClockFrequency` return the CPU or the HRT frequency, `ClockPeriod` return the CPU period (the inverse of the frequency). `Name` return the processor name or type, `Family`

return the brand of the processor, `Characteristic` return some capabilities of the CPU, `Available` return the number of available cpus on the system. Th function `ReScanCPU` reinitialize the static processor object and redo a scan of the system to count the number of CPU.

In *Processor.cpp* there is also a `HRTCalibrator` class that calibrate the CPU clock. Such class has only a constructor and no other methods or attributes. The `HRTCalibrator` class is statically instantiated with the name `hrtCalibrator`; it just sets some global variables at the start up executing the code in its contructor.

ProcessorType

`[ProcessorType.h]`
The `ProcessorType` class defines the processors where a particular task should run. There is a unique mask argument as a `uint32`, the `processorMask`, this is a bitmask and allow the maximum presence of 32 CPU on a system. The interface let the user initialize the class with a bitmask and adding CPUs but never remove one or more. There are some operator overlaoding.

For example, if we have a task is that will run on cpu 2 and 3 the mask is: 0x06 (00000110).

```
public:
   uint32 processorMask;

   ProcessorType(const uint32 cpuMask = 0xFE);
   ProcessorType(const ProcessorType &pt);

   void SetMask(const uint32 mask);
   void AddCPU(const uint32 cpuNumber);

   void operator=(const uint32 cpuMask);
   void operator=(const ProcessorType &pt);
   void operator|=(const uint32 cpuMask);
   void operator|=(const ProcessorType &pt);
```

LoadableLibrary

`[LoadableLibrary.h]`
The class `LoadableLibrary` lets BaseLib, that is a component oriented framework, load its components by means of loading external libraries. On different Operating Systems libraries have different extensions.

GENERAL ACQUISITION MODULE	`*.gam`
HIGH LEVEL DRIVER	`*.drv`
STD NAME FOR DLL	`*.dll`
LINUX NAME FOR DLL	`*.so`

Table 1.1: Loadable libraries extensions

Atomic

`[Atomic.h, Atomic.cpp]`
The class is build up only of static methods. Such routines are performed by the CPU without interruption; these are slower than those in *SPAtomic.h*, but are granted to work on multiprocessor machines.

The Solaris port is quite old and is the only one that is not coded with assembly instructions.

```
public:
   static inline void Increment (volatile int32 *p );
   static inline void Increment (volatile int16 *p);
   static inline void Increment (volatile int8 *p);

   static inline void Decrement (volatile int32 *p);
   static inline void Decrement (volatile int16 *p);
   static inline void Decrement (volatile int8 *p);

   static inline int32 Exchange (volatile int32 *p, int32 v);

   static inline bool TestAndSet(int32 volatile *p);
   static inline bool TestAndSet(int16 volatile *p);
   static inline bool TestAndSet(int8  volatile *p);
```

The class offer methods to atomically increment, decrement, exchange (swap) and test and set 8bit, 16bit and 32bit variables, 64bit variables are not supported yet.

SPAtomic

`[SPAtomic.h, SPAtomic.cpp]`
Single Processor Atomic operation support. The interface of exported methods is identical to the `Atomic` class, the only thing that differ is the source code: atomic operations are inherently processor dependent, not all CPU implements atomic instruction in the same way protecting sections to be exectued concurrently.

This routine are not processor safe, not to be executed in multiprocessor environments.

```
public:
    static inline void Increment (volatile int32 *p );
    static inline void Increment (volatile int16 *p);
    static inline void Increment (volatile int8 *p);

    static inline void Decrement (volatile int32 *p);
    static inline void Decrement (volatile int16 *p);
    static inline void Decrement (volatile int8 *p);

    static inline int32 Exchange (volatile int32 *p, int32 v);

    static inline bool TestAndSet(int32 volatile *p);
    static inline bool TestAndSet(int16 volatile *p);
    static inline bool TestAndSet(int8  volatile *p);
```

HRT

`[HRT.h, HRT.cpp]`
The HRT or High Resolution Timer class exhibit a series of public static methods. Methods let the user query the system High Resolution Timer about the current time count, in machine ticks. In a Pentium class processor the offered methods (`HRTCounter`, `HRTCounter32`) are perform a `rdtsc` assembly instruction respectively returning a 64bit and a 32bit number. The `rdtsc` x86 assembly instruction read the Time Stamp Counter register of the processor that is a 64bit register present on all x86 processors since the Pentium. It counts the number of ticks since reset.

```
public:
   static inline int64 HRTCounter();
   static inline uint32 HRTCounter32();
```

To convert the readed value to a more significant value there are some other methods. The method `HRTFrequency` return the CPU frequency, `HRTPeriod` the period of the CPU clock and `TicksToTime`

simply convert CPU ticks in time (seconds).

```
static inline int64 HRTFrequency();
static inline double HRTPeriod();
static inline double TicksToTime(int64 tStop,int64 tStart=0);

uint32 SystemMsecTime();
```

Last method is not static and require an object to be instantiated. Is really strange that this method is static, looking at the implementation it doesn't require object attributes. It count the ms from reset.

1.1.1 Design Notes

HRT's last method must be static. Infact no class methods and attributes are called. The current code follow.

```
uint32 HRTSystemMsecTime(){
   int64 count      =  HRTRead64();
   double msecTime  =  HRTClockCycle() * 1000.0;
   msecTime = msecTime * count;
   return (uint32) msecTime;
}
```

`HRTCalibrator` class must be moved and integrated in `HRT` class; now it is declared in Processor.cpp, not well designed.

ProcessorType class is not scalable. Scalability and modularity are central concepts of Object Oriented Design. It is not a good choice to dress a `uint32` as a class with only one real method (`Add`). The first step is to add at least a `Remove` method. The second step is to extend the `ProcessorType` class to support a generic number of processors; instead of an uint32 we have to add a bitmap, in this way we can define a set of class that work on bitmaps and support a generic number of processors.

Another required extension (that could be handle in a generic way) is the support of atomic operation with generic length variables starting with the 64bit support in the `Atomic` and `SPAtomic` classes.

1.2 Lists

The list abstraction is the most used abstraction in BaseLib. On top of a linked list it's build each BaseLib's DataBase; the SemNameDataBase (SNDB), GlobalObjectDataBase (GODB), ObjectRegistryDataBase (ORDB), ConfigurationDataBase (CDB) and many others.

Lists are mainly divided in two categories: linked list and static list. Figure 1.2 graph all classes involved in this group, those classes can be further splitted in linked and static list and an the *directory* abstraction that is a simple implementation of a single linked list. Static lists are basically vectors.

1.2.1 Linked Lists

Figure 1.3 depict classes involved in the linked list abstraction offered in BaseLib, the `directory` abstraction is not presented again. The basic building block of a linked list is a `LinkedListable` element that is typedefined to became a `Stackable` and a `Queueable` object, but is still the same. The modules that you are going to explore are:

- LinkedListable
- LinkedListHolder

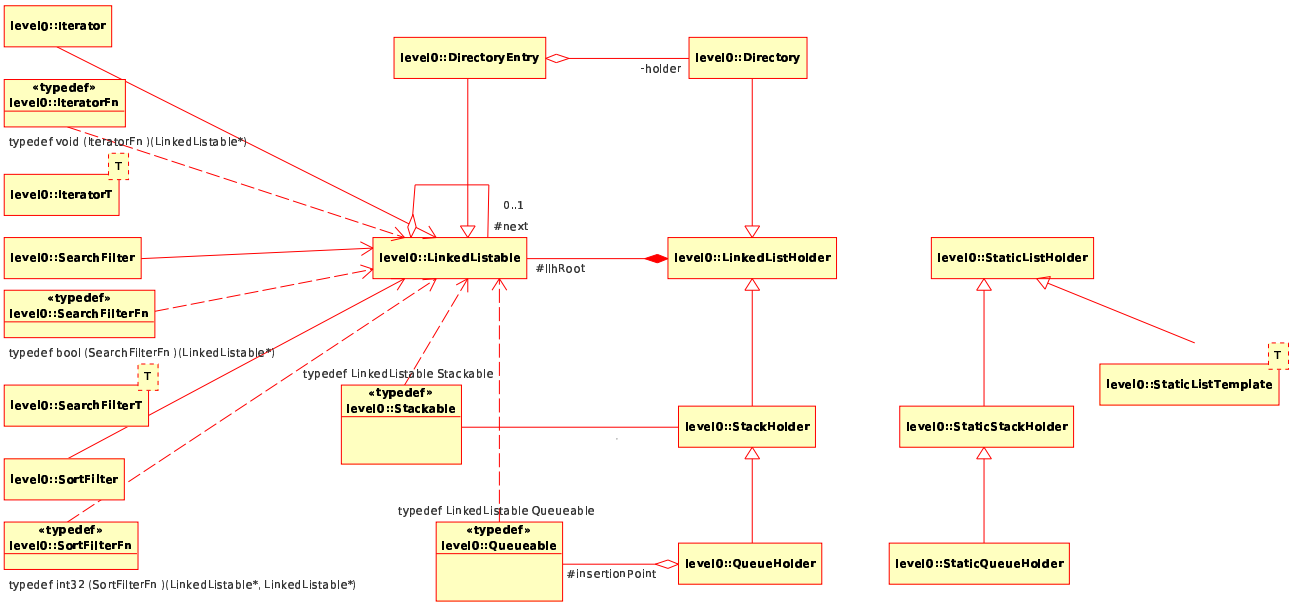

Figure 1.2: BaseLib Level0 Lists classes

- StackHolder
- QueueHolder
- Iterator, IteratorT, IteratorFn
- SearchFilter, SearchFilterT, SearchFilterFn
- SortFilter, SortFilterFn

Still in this section we discuss about the `Directory` example that let you puts the hands on the BaseLib *LinkedList* interface.

- DirectoryEntry
- Directory

LinkedListable

`[LinkedListable.h]`
A linked list member. Can be used as root of a linked list, a specific linked list member can be derived from this class (so can be subclassed). The `LinkedListable` interface must be subclassed to became specific.

A `LinkedListable` object holds only a pointer to the next `LinkedListable` object.

```
protected:
   LinkedListable *next;
```

The `Next` method return the pointer to the next `LinkedListable` object; while using `SetNext` it is possible to set the attirbute. Such list is a single concatenated linked list, it can be navigated only in one way. The *Size* method count how many `LinkedListable` iteam are on the list from the object itself, i.e. for example in Figure 1.4 imagine to be on the second `tLinkedListable` object from the left, `Size` will return 4. The `BSort` methods will operate like `Size` on the rightside objects on the list.

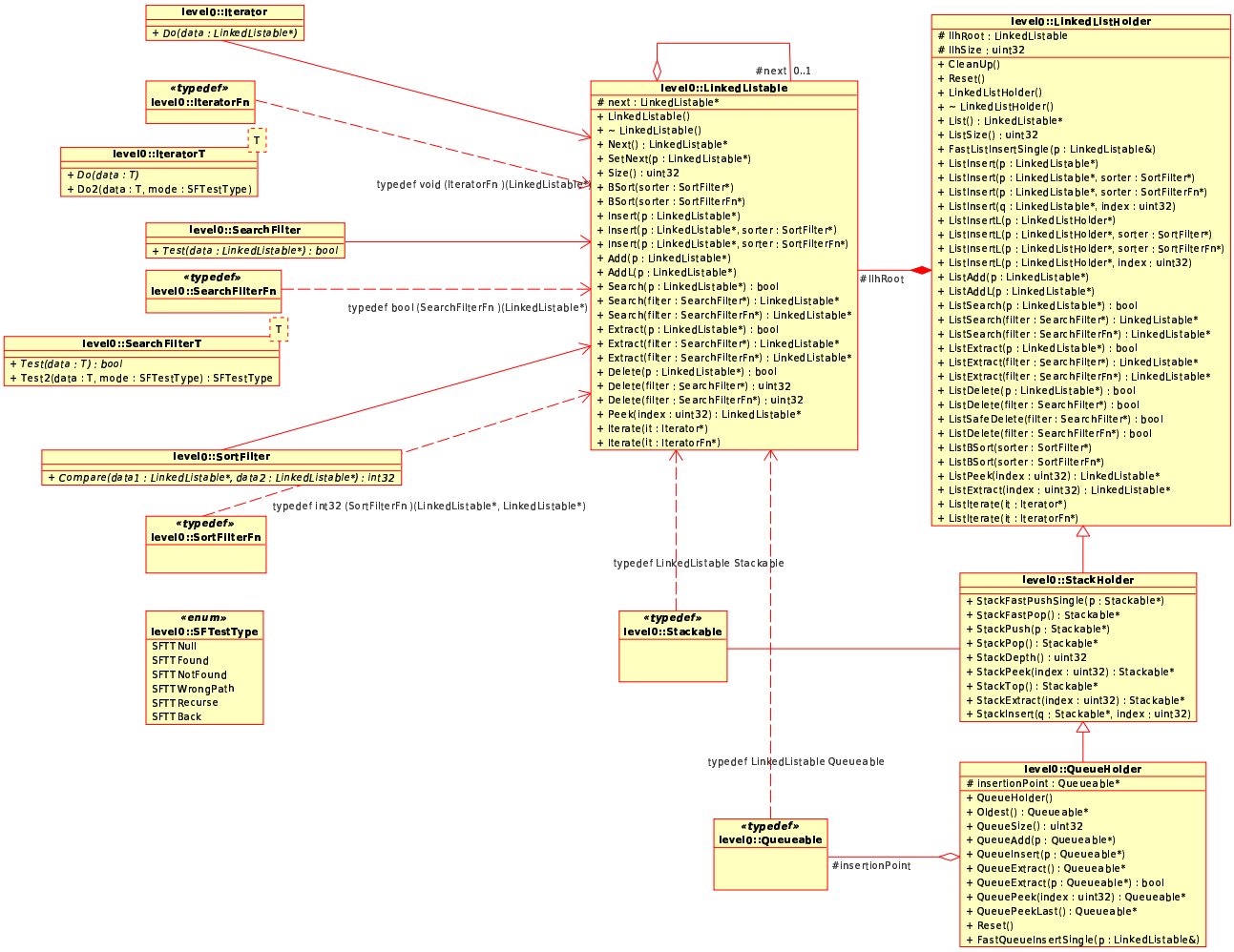

Figure 1.3: BaseLib Level0 Linked Lists classes

```
public:
   LinkedListable *Next()const;
   void SetNext(LinkedListable *p);

   uint32 Size();
   void BSort(SortFilter *sorter);
   void BSort(SortFilterFn *sorter);
```

Next functions deal with the insertion and the deletion and the searching activity on a list, probably such functions will be better coded in an iterator, but this is a design choice. `Insert` method insert the `LinkedListable` object passed as an argument after the current object in the list. N.B. This method is very powerfull, infact you can also add a list to a list! But be very carefull because if you doesn't want to add a list the `p*` must be the `next` attribute resetted to `NULL`. `Add` and `AddL` add an element or a list to the end of the current queue where the current object, on which you are calling the method, is lying. `Search` method search downside the list for a matching object. The other two search methods search down the list using a `SearchFilter` or a `SearchFilterFn`.

```
   void Insert(LinkedListable *p);
   void Insert(LinkedListable *p,SortFilter *sorter);
   void Insert(LinkedListable *p,SortFilterFn *sorter);

   void Add(LinkedListable *p);
   void AddL(LinkedListable *p);
```

```
bool Search(LinkedListable *p);

LinkedListable *Search(SearchFilter *filter);
LinkedListable *Search(SearchFilterFn *filter);
```

`Extract` functions remove the requested element from the list start searching from the next element or using a special filter, returning the requested object if searched and not specified. `Delete` routines simply search and remove the element requested from the list without returning it, the object will be destroied from the system and no reference will still exists. If the object will be not destroyed the system can suffer of memory leak.

Last few methods are to navigate and browse the linked list.

```
bool Extract(LinkedListable *p);
LinkedListable *Extract(SearchFilter *filter);
LinkedListable *Extract(SearchFilterFn *filter);

bool Delete(LinkedListable *p);
uint32 Delete(SearchFilter *filter);
uint32 Delete(SearchFilterFn *filter);

LinkedListable *Peek(uint32 index);
void Iterate(Iterator *it);
void Iterate(IteratorFn *it);
```

LinkedListHolder

`[LinkedListHolder.h]`
The `LinkedListHolder` class is a class that can handle linked lists of `LinkedListable` objects. As in the schema of Figure 1.4 a `LinkedListable` object is structure that hold linked lists and ease the usual operations on such lists (inserting, removing, seraching and sorting).

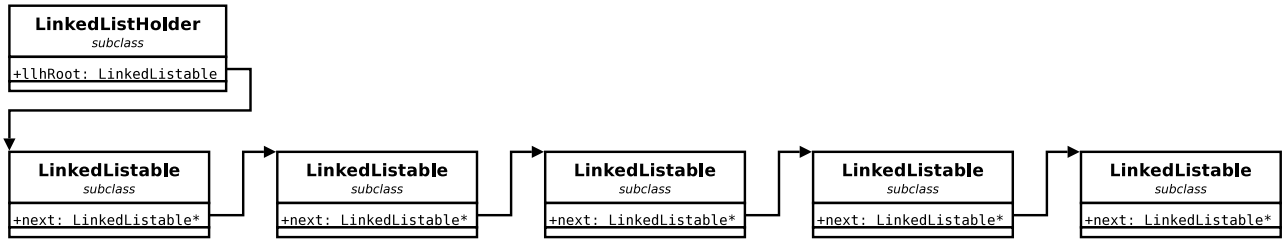

Figure 1.4: BaseLib Level0 linked list schema

A `LinkedListHolder` is like a list container. Operations on Lists are permitted, the basic informations it holds is the first `LinkedListable` node of the list and the list size in integers.

```
protected:
   LinkedListable  llhRoot;
   uint32          llhSize;
```

`CleanUp` will deallocate all the contents and then sets the pointer to `llhRoot->next` to `NULL` and `llhSize` to zero, the `Reset` method does the same but doesn't deallocate content.

The `List` method return the first element on the list (`llhRoot->next`) while `ListSize` return the size of the linked list (`llhSize`).

```
public:
   void CleanUp();
   void Reset();

   LinkedListable *List() const;
   uint32 ListSize()const;
```

The method `FastListInsertSingle` insert a single `LinkedListable` element `p` as the unique element of the queue. Overloaded method `ListInsert` let the user insert a `LinkedListable` element as the first element or in an ordered way using sorters, rather of `ListInsertL` manipulate `LinkedListHolder` elements.

Method `ListAdd` add one single `LinkedListable` element at the end of the queue; `ListAddL` add a list of `LinkedListable` elements at the end of the queue.

```
inline void FastListInsertSingle(LinkedListable &p);
void ListInsert(LinkedListable *p);
void ListInsert(LinkedListable *p,SortFilter *sorter);
void ListInsert(LinkedListable *p,SortFilterFn *sorter);
void ListInsert(LinkedListable *q,uint32 index);
void ListInsertL(LinkedListHolder *p);
void ListInsertL(LinkedListHolder *p,SortFilter *sorter);
void ListInsertL(LinkedListHolder *p,SortFilterFn *sorter);
void ListInsertL(LinkedListHolder *p,uint32 index);

void ListAdd(LinkedListable *p);
void ListAddL(LinkedListable *p);
```

`ListSearch` methods search the list comparing the element comparing with the one passed by argument or using a `SearchFilter` or a `SearchFilterFn`. Overloaded methods `ListExtract` do the same job as `ListSearch` but removing the element founded. `ListDelete` is the same as `ListExtract` but never return the object founded. Function `ListSafeDelete` is a special `ListDelete` version that is safe from reentrance from destructor of object (complex graph destruction).

Methods `ListBSort` simply sort the linked list using the sorter passed as an argument. `ListPeek` looks into the list an treat it as an array it will return the `index`-n element of the linked list. `ListIterate` methods associate an iterator to the linked list.

```
bool ListSearch(LinkedListable *p);
LinkedListable *ListSearch(SearchFilter *filter);
LinkedListable *ListSearch(SearchFilterFn *filter);

bool ListExtract(LinkedListable *p);
LinkedListable *ListExtract(SearchFilter *filter);
LinkedListable *ListExtract(SearchFilterFn *filter);
LinkedListable *ListExtract(uint32 index=0);

bool ListDelete(LinkedListable *p);
bool ListDelete(SearchFilter *filter);
bool ListSafeDelete(SearchFilter *filter);
bool ListDelete(SearchFilterFn *filter);

void ListBSort(SortFilter *sorter);
void ListBSort(SortFilterFn *sorter);

LinkedListable *ListPeek(uint32 index);

void ListIterate(Iterator *it);
void ListIterate(IteratorFn *it);
```

StackHolder

`[StackHolder.h]`
A stack data structure is simply developed on top of the single linked list abstraction. Inserting (push) elements on the head of the list and removing (pop) elements (one at a time) always from the head of

the list. Such operations require only a single linked list (i.e. no double linked list is required). Figure 1.3 show that a `StackHolder` is a subclass of a `LinkedListHolder` so is a linked list container but adds the tipical method a stack will have.

The typename `Stackable` is just a rename of the `LinkedListable` type, it is defined like below in file `level0/Iterators.h`.

```
typedef LinkedListable Stackable;
```

`StackFastPushSingle` method and pop's `StackFastPop` let you push and pop a single `Stackable` (`LinkedListable`) element from the stack. `StackPush` and `StackPop` are more slow to execute but let you add not a single element but also a linked list (or another stack).

```
public:
   inline void StackFastPushSingle(Stackable *p);
   inline Stackable *StackFastPop();

   void StackPush(Stackable *p);
   Stackable *StackPop();
```

Last methods are the same as the `LinkedListHolder` implementation let only the user call those methods with a more friendly "stackable" name.

```
   uint32 StackDepth();
   Stackable *StackPeek(uint32 index);
   Stackable *StackTop();
   Stackable *StackExtract(uint32 index);
   void StackInsert(Stackable *q,uint32 index);
```

This class is really simply and each method usually call the same method of the superclass.

QueueHolder

`[QueueHolder.h]`
`QueueHolder` is a class that can handle a queue of `Queueable` elements; the typename `Queueable` is just an alias of the `LinkedListable` type, it is defined like below in file `level0/Iterators.h`.

```
typedef LinkedListable Queueable;
```

A queue data structure is simply depicted in Figure 1.5.

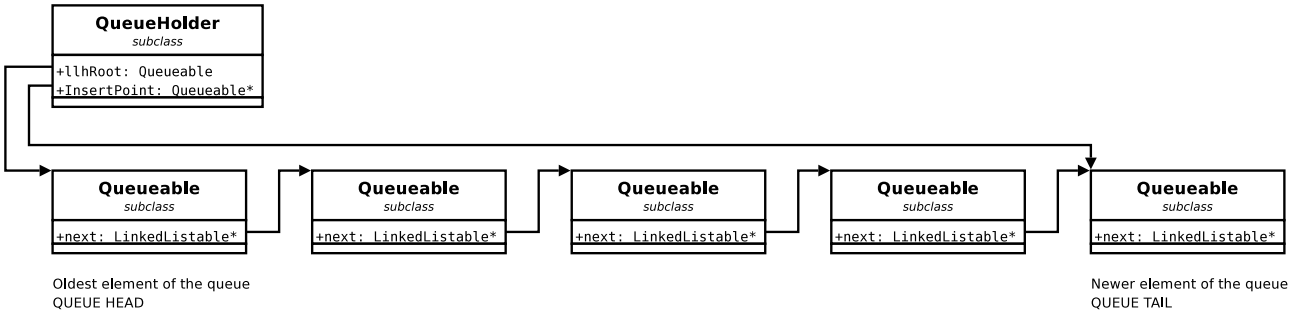

Figure 1.5: BaseLib Level0 queue schema

The `QueueHolder` class add the protected attribute `insertionPoint` to the `StackHolder` class; in this way mantaining only a single linked list with only the *next* pointer it has a full control of the queue knowing the head and tail elements, navigation of the queue remain possible only in one direction, no bidirectional navigation could be possible. In this way removing elements at the tail is very time consuming; the design allows removing elements only from the head of the queue.

```
protected:
   Queueable *insertionPoint;
```

The *oldest* element is the element at the head of the queue and is also the oldest added to it. The insertion of a new element will be made at the tail of the queue.

The method `Oldest` return the oldest inserted element in the queue, returning doesn't mean extracting infact the element still reamin in the queue. Note that it is possible to return only one element at a time. The extraction activity is done by the `QueueExtract` method that return and remove the oldest element on the queue. Last `QueueExtract` search from the oldest `Queueable` item the element passed by argument and remove it.

The `QueueAdd` method add on the queue head the passed by `Queueable` elements, it is possible to add also a list of `Queueable`s. `FastQueueInsertSingle` does the same but add to the queue only one element at a time. `QueueInsert` insert the passed by argument `Queueable` element/s resetting all the content of the queue to it/them; use it with care because can remove all elements in the queue.

```
public:
   Queueable *Oldest();
   Queueable *QueueExtract();
   bool QueueExtract(Queueable *p);

   void QueueAdd(Queueable *p);
   inline void FastQueueInsertSingle(LinkedListable &p);
   void QueueInsert(Queueable *p);
```

Methods `QueuePeek` and `QueuePeekLast` do not remove the required element from the queue; the first one do not do any check on queue boundary before performing the peek action. `Reset` method simply call the superclass one removing one by one elements from the list. Elements could be not more refereneced and so can be lost wasting memory.

```
   Queueable *QueuePeek(uint32 index);
   Queueable *QueuePeekLast();

   void Reset();
   uint32 QueueSize();
```

Iterator, IteratorT, IteratorFn

`[Iterator.h]`
The `Iterator` class define only one pure virtual function so it is a C++ interface. The function is defined as:

```
public:
   virtual void Do (LinkedListable* data)=0;
```

The `Do` function perform the iteration passing a `LinkedListable` element as an argument. `IteratorT` is a more specialised version of the `Iterator` letting the developer not to specify a `LinkedListable` element or a subclass of it in the `Do` method. `IteratorT` is also an interface.

`IteratorFn` is simply a typedef defined as follow:

```
typedef void (IteratorFn )(LinkedListable* data);
```

SearchFilter, SearchFilterT, SearchFilterFn

`[Iterator.h]`
The `SearchFilter` class define only one pure virtual function so it is a C++ interface. The function is defined as:

```
public:
   virtual bool Test (LinkedListable* data)=0;
```

The `Test` function perform the search on a set of data passing a `LinkedListable` element as an argument. A new class that implement the searching activity must be defined. `SearchFilterT` is a more specialised version of the `SearchFilter` letting the developer not to specify a `LinkedListable` element or a subclass of it in the `Test` method. `SearchFilterT` is also an interface.

`SearchFilterFn` is simply a typedef defined as follow:

```
typedef bool (SearchFilterFn )(LinkedListable* data);
```

SortFilter, SortFilterFn

`[Iterator.h]`
The `Iterator` class define only one pure virtual function so it is a C++ interface. The function is defined as:

```
public:
   virtual void Do (LinkedListable* data)=0;
```

The `Do` function perform the iteration passing a `LinkedListable` element as an argument. `IteratorT` is a more specialised version of the `Iterator` letting the developer not to specify a `LinkedListable` element or a subclass of it in the `Do` method. `IteratorT` is also an interface.

`IteratorFn` is simply a typedef defined as follow:

```
typedef void (IteratorFn )(LinkedListable* data);
```

TODO
TODO
TODO
thanks Ric
aggiugere qualche esempio di utilizzo (ps prendi il libro di dati e algoritmi)

DirectoryEntry

`[Directory.h]` Such `DirectoryEntry` class represent a file or a subdirectory. Usually a directory structure is a tree data structure but such class is implemented using a single linked list; this abstraction doesn't represent a tree of files or directories but simply the directory content, letting navigate in an OS directory.

There are three private attributes: the file or subdirectory name, some statistics about the file and a pointer to a `Directory` conteiner that is a `LinkedListHolder`.

```
private:
   const char* fname;
   struct stat fileStatistics;
   Directory* holder;
```

Such attributes let the `Directory` and `DirectoryEntry` classes change the `LinkedListHolder` and `LinkedListable` structure to showed in Figure 1.6.

A creation time the developer must supply the `Directory` container and the name of the file or subdirectory. The `Name` method return the name of the file or subdirectory and the methods `IsDirectory` and `IsFile` let you check if the instance represent a directory or a file.

`ReadOnly` let you query if the file has the *read only* attribute, `Size` method return the size of the file. The method `Time` return the last write time and `LastAccessTime` get the last access time.

```
public:
   DirectoryEntry(Directory *holder, const char *fname = NULL);
   ~DirectoryEntry();
```

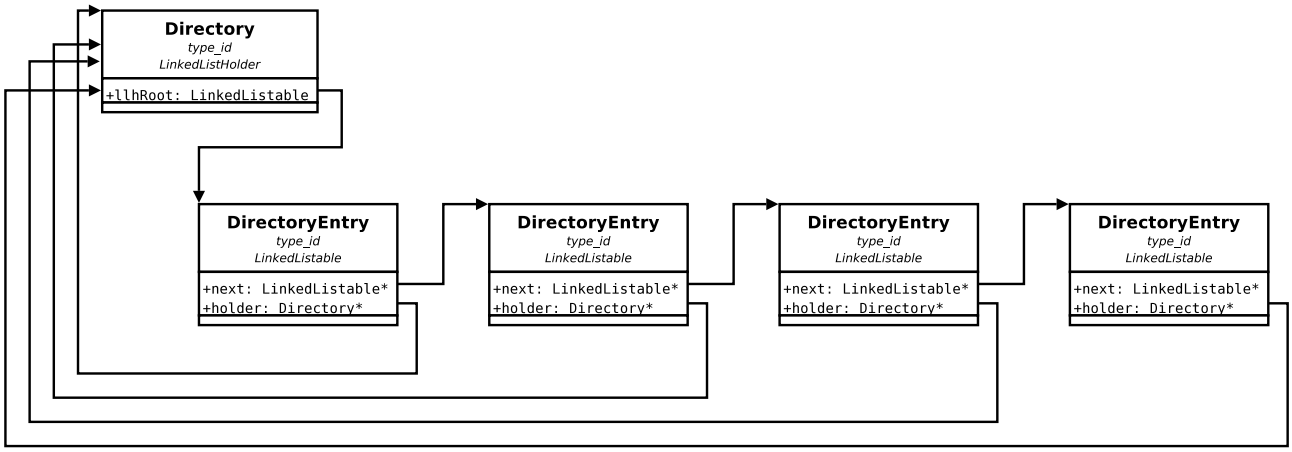

Figure 1.6: BaseLib Level0 Directory scheme

```
const char* Name();
bool IsDirectory();
bool IsFile();

bool ReadOnly();
int64 Size();

time_t Time();
time_t LastAccessTime();
```

Directory

[Directory.h] The `Directory` class contains information about a directory and all its content. It basically hold a single linked list of `DirectoryEntry` that can be file and/or subdirectory names.

A `Directory` is identified by a path in the FileSystem hierical on each common OS nowadays. So the first attribute of the class is the path of the directory to be listed `baseAddress`; the second attribute is the size of the entire directory `size`.

```
private:
   char* baseAddress;
   uint64 size;
```

Such class has only few methods; the constructor does the major stuff. The constructor require the path of the directory a file mask and a sorter, in this way it is possible to have a list of only the file of a certain format. `TotalFileSize` simply return the `size` attribute.

The two static methods are currently coded only for Windows platforms and let the user create a directory and test for directory existance.

```
public:
   Directory(const char* address,const char* fileMask="*",SortFilterFn* sorter=NULL);
   uint64 TotalFileSize();

   static bool Create(const char* address);
   static bool DirectoryExists(const char* address);
```

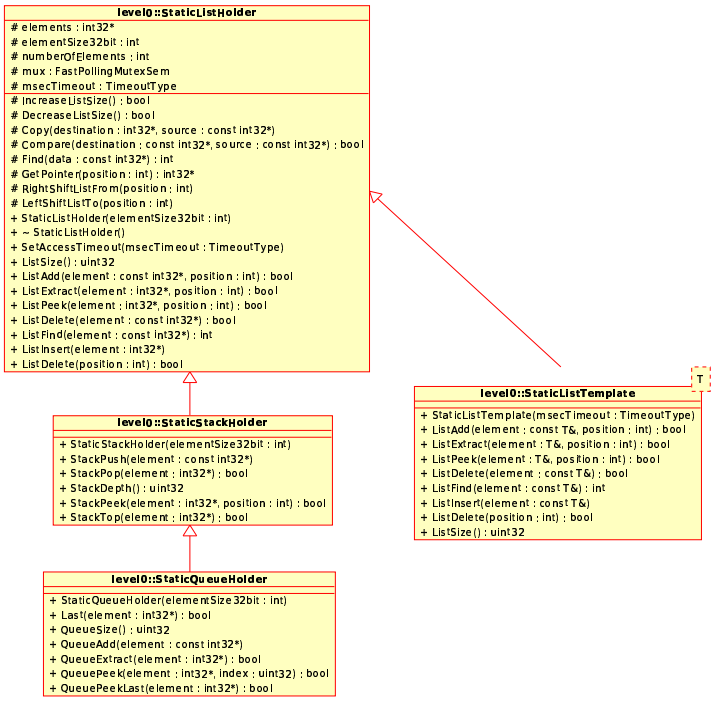

Figure 1.7: BaseLib Level0 Static Lists classes

1.2.2 Static Lists

Static Lists are basically *arrays* or *vectors*. Such structure contains 32bit pointers or any other structure. This list of pointers can grow and shrink and the reallocation is done by the class itself using library routines like `malloc` and `realloc`. It obvioulsy allows inserting and removing of elements. It also offers the possibility to control the data access via a semaphore.

Class diagram is depicted in Figure 1.7, classe in this group are:

- StaticListHolder
- StaticStackHolder
- StaticQueueHolder
- StaticListTemplate

Static list classes are written with the aim of mantaining the same methods signatures of the Linked List interface.

StaticListHolder

`[StaticListHolder.h]`
The main data structure that hold all the data is an array; this array is dynamically allocated at runtime and is pointed from the `elements` attribute. Such pointer point to 32bit integers, this integers are usually 32bit pointers, but can be also a structures of the same size; the constructor let you create an array with elements of a size multiple of 32bit. So the *StaticList* can hold 32bit ptr or 64bit ptr or any other structure. The protected attribute `elementSize32bit` give you the size of one item on the *StaticList*; the `numberOfElements` give the number of items in the *StaticList*, the `FastPollingMutexSem` give a way to allow multiple threads protected access to the data and the `msecTimeout` is the timeout to wait on the semaphore for catching the resource.

```
protected:
   int32* elements;
   int elementSize32bit;
   int numberOfElements;

   FastPollingMutexSem mux;
   TimeoutType msecTimeout;
```

The following protected methods are self explained and are used to grow and shrink the ata structure, copy and move datas and finding and indexing an element in the `StaticListHolder`. Comments are left.

```
   /** increases the list size by 1.
       Manages the reallocation of memory when necessary */
   bool IncreaseListSize();
   /** decreases the list size by 1.
       Reallocation of memory is not performed */
   bool DecreaseListSize();

   /** copies data from a buffer to the position */
   inline void Copy(int32 *destination,const int32 *source);
   /** compares data between source and destination. True means equal */
   inline bool Compare(const int32 *destination,const int32 *source);
   /** finds data in list. -1 means not found */
   inline int Find(const int32 *data);

   /** retrieves address of data element */
   inline int32* GetPointer(int position);
   /** moves all the pointers from position to the right.
       Assumes that the last position is empty */
   inline void RightShiftListFrom(int position);
   /** removes the element in position and shifts all the elements
       at the right of it to the left */
   inline void LeftShiftListTo(int position);
```

Public methods have the same names as `LinkedListHolder`'s methods, arguments differ. The size of the elements, in 4byte units, is defined by the constructor and is no more modificable.

To control multiple access it is possible to set a timeout on the waiting operation, the timeout can be set with `SetAccessTimeout`. To retrieve the number of elements in the `StaticList` use `ListSize`.

`ListAdd` add one element at any specified position; as a shortcut if the element must be added at the top of the list you can pass 0 as at position argument and -1 for the end of the list; `ListInsert` simply add the element at the top of the array.

The method `ListFind` finds at what index the specified data is located, it returns -1 if doesn't found any matching object.

`ListExtract` removes an element from any position, if `position` is 0 it removes from the top, if -1 removes from the end; `element` is a pointer to a buffer of data that the will be filled by the method. `ListPeek` reads an item from the data structure without affecting the list; `element` argument is a pointer to a buffer of data that will be copied from the list. The method `ListDelete` removes an element from the list using a copy of the `element` as a search key. The last `ListDelete` removes at the specified indexed position.

```
public:
   StaticListHolder(int elementSize32bit = 1);
   virtual ~StaticListHolder();

   void SetAccessTimeout(TimeoutType  msecTimeout = TTInfiniteWait);
   uint32 ListSize()const;

   bool ListAdd(const int32 *element,int position = SLH_EndOfList);
```

```
   inline void ListInsert(int32 *element);

   int ListFind(const int32 *element);

   bool ListExtract(int32 *element=NULL,int position = SLH_EndOfList);
   bool ListPeek(int32 *element=NULL,int position = SLH_EndOfList);
   bool ListDelete(const int32 *element);
   inline bool ListDelete(int position);
```

StaticStackHolder

[StaticStackHolder.h]
This class is try to mimic the same API of StackHolder but relaying on arrays. The constructor creates a stack with the given element size (note that the size is in 32 bit multiples), the size of the stack my vary during run time. `StackPush` insert on top a single element. When the space is finished the bottom is discarded; `StackPop` get an element from top of the array and remove it from the stack; for not removing the element from the stack use `StackTop`. Method `StackPeek` looks into the stack for an element with the given index `position`. `StackDepth` return the number of elements in the stack.

```
public:
   StaticStackHolder(int elementSize32bit=1);

   inline void StackPush(const int32* element);
   inline bool StackPop(int32* element);
   inline bool StackTop(int32* element);
   inline bool StackPeek(int32* element,int position);

   inline uint32 StackDepth();
```

StaticQueueHolder

[StaticQueueHolder.h]
This class is still try to mimic the same API of QueueHolder but relaying on arrays. Like the `StaticStackHolder` and the superclass `StatiListHolder` this class require the size of the elements in 32bit units in the constructor. `QueueSize` return the number of elements in the queue. `QueueAdd` insert an element on the queue, `Last` return the last inserted element in the queue, `QueueExtract` removes the oldest element from the queue; `QueuePeek` looks into the queue, index 0 is the most recent added; `QueuePeekLast` looks into the queue to the last element inserted.

```
public:
   StaticQueueHolder(int elementSize32bit = 1);

   inline void QueueAdd(const int32* element);
   inline bool Last(int32* element);
   inline bool QueueExtract(int32* element);
   inline bool QueuePeek(int32* element,uint32 index);
   inline bool QueuePeekLast(int32* element);

   inline uint32 QueueSize();
```

StaticListTemplate

[StaticListTemplate.h]
This class is a template to customise a `StaticListHolder` for a specific struct or class. Such class is a subclass of the `StaticListHolder`. This class is never used in BaseLib. This class lets specializing the superclass for a particular type of object. Using such specialized class you have an array of object of the same size of the object templatized for. The interface, really similar to the `StaticListHolder` one follows.

```
public:
   StaticListTemplate(TimeoutType  msecTimeout = TTInfiniteWait);

   inline bool ListAdd(const T &element,int position=SLH_EndOfList)
   inline void ListInsert(const T &element);

   inline bool ListExtract(T &element,int position=SLH_EndOfList)
   inline bool ListPeek(T &element,int position = SLH_EndOfList)
   inline bool ListDelete(const T &element);
   inline bool ListDelete(int position);

   inline int ListFind(const T &element);

   uint32 ListSize()const;
```

1.2.3 Design Notes

There is no need to typedef `LinkedListable` in `Stackable` and `Queueable` but is not an error, if there is no need to extend it with further information why aliasing it? A good choice could be to extend `LinkedListable` in `Queueable` creating a double linked list that could be a real advantage to boost the performance of walking a list.

(why not call them symmetrically? Like `StackFastPush` and `StackFastPop`)
(those function can have a more significant name like `QueueOldest` and `QueueExtract`)
The `QueueAdd` method add on the queue head the passed by `Queueable` elements, it is possible to add also a list of `Queueable`s. `FastQueueInsertSingle` does the same but add to the queue only one element at a time. This is another misusage of names in this section: better names can be `QueueAdd` and `QueueAddFast`.

Some methods, like `queueInsert` are to be used with care. `queueInsert` add elements to the queue deleting all other elements in the queue without any check or call to deletion wasting all references.

Sometimes there are no boundaries checks in the *linked* implementation but also in the *static* implementation.

Probably there is a lack of a Template, look at *level0/Iterator.h* there are those declariotions: `Iterator`, `IteratorFn`, `IteratorT`; `SearchFilter`, `SearchFilterFn`, `SearchFilterT`; `SortFilter`, `SortFiletrFn` and `SortFilterT` is not defined.

The `Directory` pair of classes must be ported also on other OSes. The `Driectory` structure can be developed on a tree data structure instead on a single linked list. So there is the need to add the tree data structure to this data structure section.

The `StaticQueue` structure will be better developed on a circular buffer instead on a `StaticListHolder`. Will be more processor efficient.

1.3 IPC

This group of classes add the semaphore abstraction and the spinlock mechanism to the BaseLib library. Figure 1.8 illustrate the classes structure, semaphores and spinlocks, of a great interest is the *SemNameDataBase*.

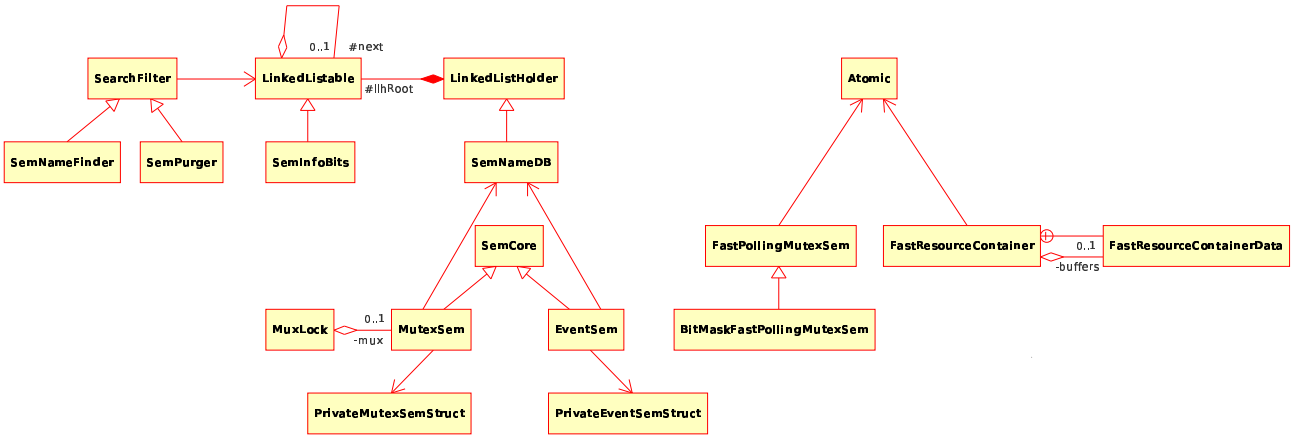

Figure 1.8: BaseLib Level0 IPC classes

1.3.1 Semaphores

A semaphore is one of the simplest Inter Processes Communication mechanisms. BaseLib introduce different types of semaphores. One of the most important functionality it adds is the *SemNameDataBase* that is a list of all BaseLib's semaphores created run-time. Such stuff lets the developer exploits named semaphores on each Operating System.

Figure 1.9 depict semaphore classes in this section (other classes previously anlized are also present: `LinkedListable`, `LinkedListHolder` and `SearchFilter`):

- SemInfoBits
- SemNameDB
- SemNameFinder, SemPurger
- SemCore
- EventSem
- MutexSem
- MuxLock

We are going first to explore the *SemNameDataBase* architecture. The OS2 naming convention is used in this group of classes, in OS2 there are `EventSem` and `MutexSem`. Follow a table that summarizes some differences between ipc's API in OS/2, POSIX.1b (UNIX, System V) and POSIX.1c (pthread). The mapping between OS/2 and POSIX.1b is not straightforward.

Table 1.2 must clarify the mapping between *EventSem* in OS/2 and *Conditional Variables* in POSIX.1c and between *MutexSem* and *mutex* in POSIX.1c.

SemInfoBits

`[SemCore.cpp]`
The class `SemInfoBits` is a `LinkedListable` list item of the `SemNameDataBase` (that is a `LinkedListHolder`). There is a `SemInfoBits` list item for each semaphore istantiated in the library, for each semaphore in the DataBase we save a `name`, an `id`, some `data` and a usage count (`useCount`).

```
private:
   char* name;
   SEM_ID id;
```

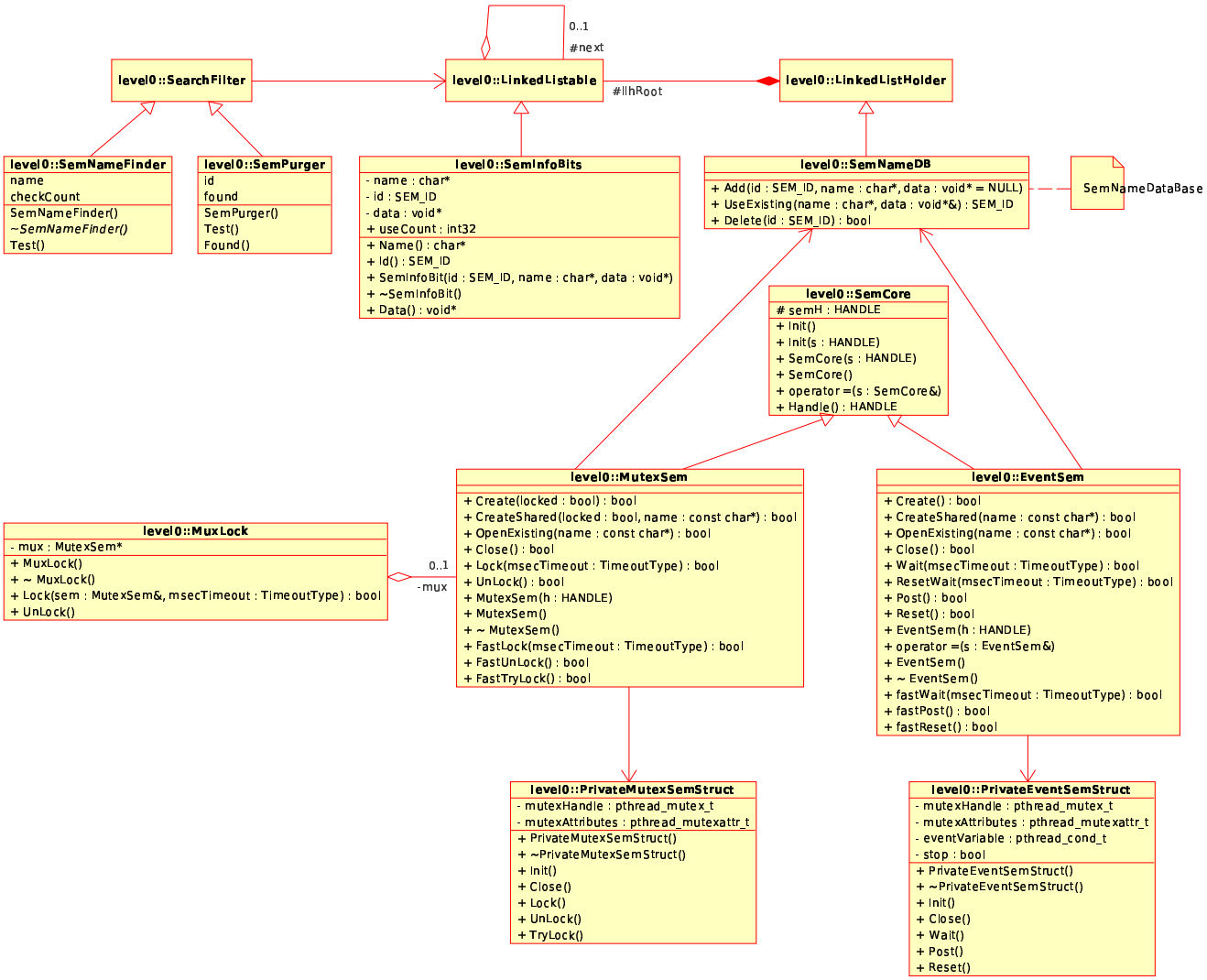

Figure 1.9: BaseLib Level0 semaphore classes

OS/2	POSIX.1c	POSIX.1b
DosCreateMutexSem	pthread_mutex_init	sem_init
DosRequestMutexSem	pthread_mutex_lock / pthread_mutex_trylock	sem_wait / sem_trywait
DosReleaseMutexSem	pthread_mutex_unlock	sem_post
DosCloseMutexSem	pthread_mutex_destroy	sem_destroy
DosCreateEventSem	pthread_cond_init	sem_init
DosWaitEventSem	pthread_cond_signal_wait / pthread_cond_timedwait	sem_wait / sem_trywait
DosPostEventSem	pthread_cond_signal / pthread_cond_broadcast	sem_post
DosCloseEventSem	pthread_cond_destroy	sem_destroy

Table 1.2: Basic IPC in OS2, POSIX.1b (UNIX, System V) and POSIX.1c (pthread)

```
   void* data;
public:
```

```
int32 useCount;
```

All methods are public and except the constructor there are only get methods that let you retrieve attributes value.

```
SemInfoBit(SEM_ID id,char* name,void* data);
~SemInfoBit();

char* Name();
SEM_ID Id();
void* Data();
```

SemNameDB

[SemCore.cpp]
The `SemNameDataBase` is the only instance of the `SemNameDB`; it holds the run-time database of all semaphores alive in the system. Such class extends `LinkedListHolder` adding the three methods below that let the user adding and deleting a semaphore in the list and getting its identifier as a `SEM_ID` data type.

```
public:
   void Add(SEM_ID id,char* name,void* data=NULL);
   bool Delete(SEM_ID id);

   SEM_ID UseExisting(char *name,void *&data);
```

Data type `SEM_ID` is a pointer or an handle, it depends on the Operating System in use.

SemNameFinder, SemPurger

[SemCore.cpp]
Those two classes represent the last part of the *SemNameDataBase* implementation; togheter they let the user searching the database linked list, those classes are subclass of the `SearchFilter` class.

A `SemNameFinder` class must be created each time we want to search a semaphore by name and obviously we know the name. After creating the class wit is possible to use the `Test` method passing a pointer to a `LinkedListable` element and know if there is a semaphore with that name in the sublist passed by argument. The class interface follow. The attirbute `checkCount` is not used right now. Such class must be used to wuery about the existence of a particular semaphore instance.

```
class SemNameFinder : public SearchFilter{
   char *name;
   bool checkCount;
public:
   SemNameFinder(char* name,bool checkCount=False);
   virtual ~SemNameFinder();

   bool Test(LinkedListable* data);
};
```

A `SemPurger` class behave in the same way as a `SemNameFinder` but search the *SemNameDataBase* by `SEM_ID` not by name. Just to not be symmetrically they have a different API (difficult to remember). `Found` method return the saved result of the searching activity, the class has no way to know if the search activity was just performed or not and so doesn't know about the freshness of the `found` value.

```
class SemPurger : public SearchFilter{
   SEM_ID id;
   bool found;
public:
   SemPurger(SEM_ID id);
```

```
   bool Test(LinkedListable* data);
   bool Found();
};
```

SemCore

[SemCore.h, SemCore.cpp]
The `SemCore` class isn't a semaphore or a semaphore interface instead it is an abstract class that provide OS handle registration; this class doesn't provide registration and deletion with the *SemNameDataBase* that is done by the subclasses `MutexSem` and `EventSem` (see Figure 1.9), the only thing it does is managing OS handle.

The `Init` method set the semaphore handle to 0 or to the passed by handle; constructors call the relative `Init` function.

```
protected:
   HANDLE semH;
public:
   void Init();
   void Init(HANDLE s);

   SemCore();
   SemCore(HANDLE s);

   void operator=(SemCore& s);
   inline HANDLE Handle();
```

EventSem

[EventSem.h]
Such class define and implement an event shemaphore. Such class deal directly with the OS. Is really thightly bounded. The constructor simply set if passed by the OS semaphore handle, in this case is the user that create a semaphore. The class itself has method to create a sempahore with the OS or to find and reuse other existing registered semaphores. Is some OS the class use an helper class called `PrivateEventSemStruct`.

```
public:
   EventSem();
   EventSem(HANDLE h);
   ~EventSem();
```

To open and create a new semaphore there are several methods; `Create` create a general semaphore without a name; `CreateShared` create a named semaphore and register it in the *SemNameDataBase*; `OpenExisting` query the *SemNameDataBase* for the required semaphore and take a reference to it.

The `Close` method closes the semaphore possibly deleting it from the system if the usage count is zero; the method ask also to the *SemNameDataBase* to delete it. Some more checks on the usage count must be done.

```
   bool Create();
   bool CreateShared(const char* name);
   bool OpenExisting(const char* name);

   bool Close(void);

   void operator=(EventSem &s);
```

In BaseLib `EventSem` offers an interface like the OS2 operating system, providing the wait, reset and post operations in two flavours. In POSIX such stuff correspond to Conditional Variables. `Wait`

method try to enter a critical region waiting an for an event with a timeout passed as an argument, `ResetWait` resets the semafore and then waits to enter the critical section; `Post` exit from the critical section and `Reset` reset the semaphore to its unposted state.

Methods `fastWait`, `fastPost` and `fastReset` behave in the same way as the previous routines but without signalling for errors and making other stuff, on Linux and Solaris those methods call the non fast methods.

```
bool Wait(TimeoutType msecTimeout = TTInfiniteWait);
bool ResetWait(TimeoutType msecTimeout = TTInfiniteWait);
bool Post(void);
bool Reset(void);

inline bool fastWait(TimeoutType msecTimeout = TTInfiniteWait);
inline bool fastPost(void);
inline bool fastReset(void);
```

MutexSem

`[MutexSem.h]`
Also the `MutexSem` class born from OS2 like the `EventSem` class and then was ported on other operating systems. The OS2 naming convention is used also in BaseLib. This class behave like a simple mutex.

Like in the `EventSem` class you can create a `MutexSem` passing by an handle to an operating system object (that is not checked against the type).

```
public:
   MutexSem(HANDLE h);
   MutexSem();
   ~MutexSem();
```

The primitives uses common naming as other operating systems so are quite simple to understand. The `Create` method create a mutex (or binary semaphore) with a given initial state without a name; `CreateShared` create a new named mutex registering its name to the `SemNameDataBase`; `OpenExisting` queries the *SemNameDataBase* about the named semaphore and return its reference.

The `Close` method closes the semaphore possibly deleting it from the system if the usage count is zero; the method ask also to the *SemNameDataBase* to delete it. Some more checks on the usage count must be done.

```
bool Create(bool locked=False);
bool CreateShared(bool locked,const char *name);
bool OpenExisting(const char *name);

bool Close();
```

To enter a critical section call the `Lock` method with a timeout argument, on exit call the `UnLock` method. Last three methods call directly the OS specific API without doing anything else.

```
bool Lock(TimeoutType msecTimeout = TTInfiniteWait);
bool UnLock(void);

inline bool FastLock(TimeoutType msecTimeout = TTInfiniteWait);
inline bool FastUnLock(void);
inline bool FastTryLock();
```

MuxLock

`[MuxLock]`
This class simplify the use of a `MutexSem`. It ensures that at every exit point of a function the mux

is unlocked. Use in conjunction with a MutexSem guarantees mux unlocking upon exiting the scope, this is achieved using global or local static allocation (i.e. global or per frame) calling `UnLock` in the distructor. Note that for each `MutexSem` there can be N instantiated `MuxLock`.

```
class MuxLock{
   MutexSem *mux;
public:
   MuxLock() { mux = NULL; }
   ~MuxLock() { UnLock(); }
   bool Lock(MutexSem& sem,TimeoutType msecTimeout=TTInfiniteWait);
   void UnLock();
};
```

1.3.2 Spinlocks

The second mechanism presented in this section is the spinlock. Spinlock are not OS based but hardware based commonly used in kernels to low level synchronize different entity.

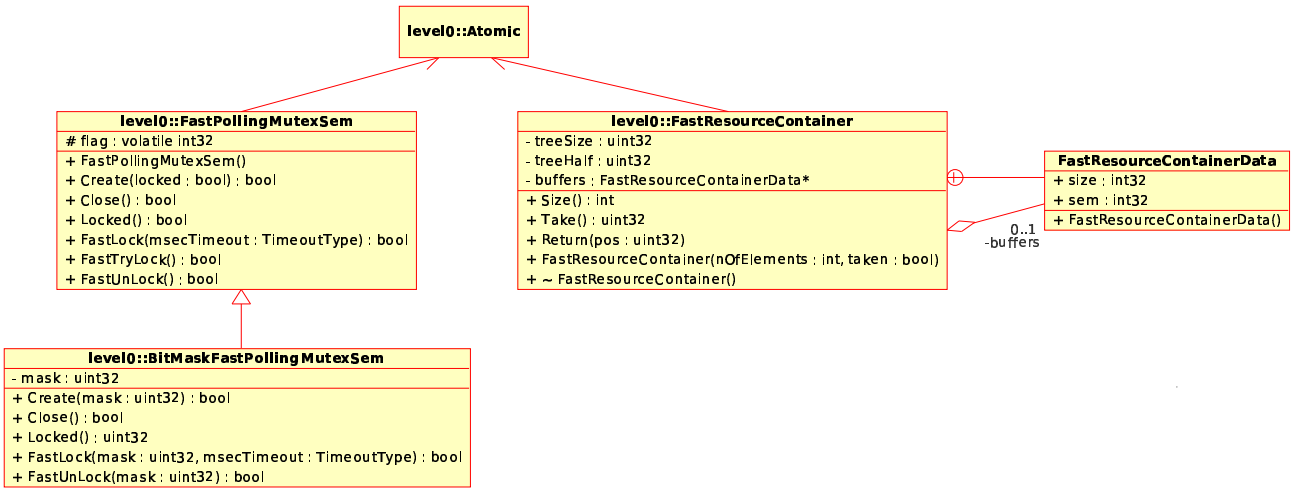

Figure 1.10: BaseLib Level0 IPC Fast classes

Figure 1.10 depict spinlock classes in this section (other classes previously anlized are also present: `Atomic`):

- FastPollingMutexSem
- BitMaskFastPollingMutexSem
- FastResourceContainer

FastPollingMutexSem

`[FastPollingMutexSem]`
Class `FastPollingMutexSem` implements a spinlock using atomic test and set operations. Such class can be lock and unlock using also a timeout that is calculated using the `HRT` class.

API naming is quite similar to the `MutexSem` implementation. The interface is quite simple. `Create` method initializes the semaphore and reads it; `Close` undo semaphore initialization.

The ambiguous named method `Locked` returns the status of the semaphore.

In this case "fast" functions have a meaning infact are the core of this class. `FastLock` poll the state of the mutex variable at a rate of a one milli seconds until `msecTimeout` is expired; this is a user

space implementation of a spinlock; `FastTryLock` basically try to achieve the lock one time without waiting; `FastUnLock` unlock the mutex.

```
public:
   bool Create(bool locked = False);
   bool Close();

   inline bool Locked();

   inline bool FastLock(TimeoutType msecTimeout=TTInfiniteWait);
   inline bool FastTryLock();
   inline bool FastUnLock(void);
```

This class is suitable for multiprocessor systems and is the best choice when no OS interference is needed.

BitMaskFastPollingMutexSem

`[BitMaskFastPollingMutexSem.h]`
This structure group 32 mutexes, the locked/unlocked state of those mutexes can not be changed interleavedly so a mutex exist to control the access to the shared resource that is the `uint32` attribute holding the state of all mutexes. This class subclasses the `FastPollingMutexSem` class requiring it to provide access control to the shared 32 mutex's states.

The `mask` argument in the following methods is required to identify which of the 32 mutexes we want deal with.

```
private:
    uint32 mask;
public:
   bool Create(uint32 mask=0);
   bool Close();

   inline uint32 Locked();

   inline bool FastLock(uint32 mask,TimeoutType msecTimeout=TTInfiniteWait);
   inline bool FastUnLock(uint32 mask);
```

FastResourceContainer

`[FastResourceContainer.h]`
The class `FastResourceContainer` is basically an array data structure that if associated to a pool of resource give you the state of each resource. This means that let's for example that you have a pool of N `int`s in an array that goes from index 0 to index N-1, so you need one of them but without taking care of the index; this could be the condition when multiple entity concurrently need the same type of resource. In this situation you want to ask for a resource and take it without waiting and whitout OS's intervention. A `FastResourceContainer` is the key in this situation, it doesn't allocate space buffer for you but can take care of which element of your pool are in use and which are not in use, answering for resource in a time O(n log n). If the pool has no resource your request simply fails. An example application will follow.

This mechanism can be imaged as a counting semaphore, independent from the OS because relies only on atomic hardware operations, so can be used between processes, threads, interrupt context on different processors in the mean time returning always immediate (without delay) in a bounded time (real time).

The class relay on a nested defined class `FastResourceContainerData` that take account of the usage state of each controlled resource. The constructor take care on how many resource do you want to control, `Size` return the same number; `Take` return the index of a free resource and `Return` let you returning a resource index to the system.

```
private:
   uint32 treeSize;
   uint32 treeHalf;

   FastResourceContainerData* buffers;
public:
   FastResourceContainer(int nOfElements,bool taken=False);
   virtual ~FastResourceContainer();

   inline int Size();

   inline uint32 Take();
   inline void Return(uint32 pos);
```

Then follow an illustrative example on how to use a `FastResourceContainer`. In this example we create a buffer memory pool, this buffer memory pool can be used in multiprocessor and in multiple context in the same time thanks to the `FastResourceContainer`.

```
class DynamicPool: protected FastResourceContainer {
   int size;
   char* ptrBuffer;
public:
   DynamicPool(unsigned int size, int nOfElements, bool taken=False) {
      if (!(size)) size = 1;
      FastResourceContainer(nOfElements, taken);
      ptrBuffer = new char[size*nOfElements];
      this->size = size;
   }
   virtual ~DynamicPool() {
      if (ptrBuffer) delete ptrBuffer;
   }
   int alloc(char*& address) {
      int handle = (int)Take();
      if (handle == -1) return -1;
      address = ptrBuffer + (char*)(size*handle);
      return handle;
   }
   bool free(int handle) {
      return Return((unsigned int)handle);
   }
};
```

1.3.3 Design Notes

The *SemNameDataBase* is used in every installation where the OS's libraies doesn't supply the named semaphore abstraction. Some problems with semaphores are stated in the text; one big implementation problem is the static casting. Static casting is usually a bad programming choice, lets take a non skilled programmer: it pass to a function a wrong object without the right attributes but we are using those attributes in the function! What happens? Bugs, bugs and bugs...

SemPurger The `Found` method is not a well designed method: the class is not associated to a `LinkedList` so there is no way to know to which sublist the `found` attribute's value refer. A bad implementation choice in those classes is the interface of the `Test` method: is not really the interface but the code, inside the code there is a static cast of the `data` argument to `SemInfoDataBits` a *dynamic*

cast was probably a better choice to be sure to have some consistency checks in the code.

`SemCore` that probably can be renamed in something like `InterfaceSem`; such class is quite useless right now holding only an handle to an OS's entity. The usage count in the *SemNameDataBase* is not currently checked against deletion of a semaphore.

The ambiguous named method `FastPollingMutexSem::Locked` returns the status of the semaphore, a better choice for this name could be `FastPollingMutexSem::isLocked` (?).

1.4 Network

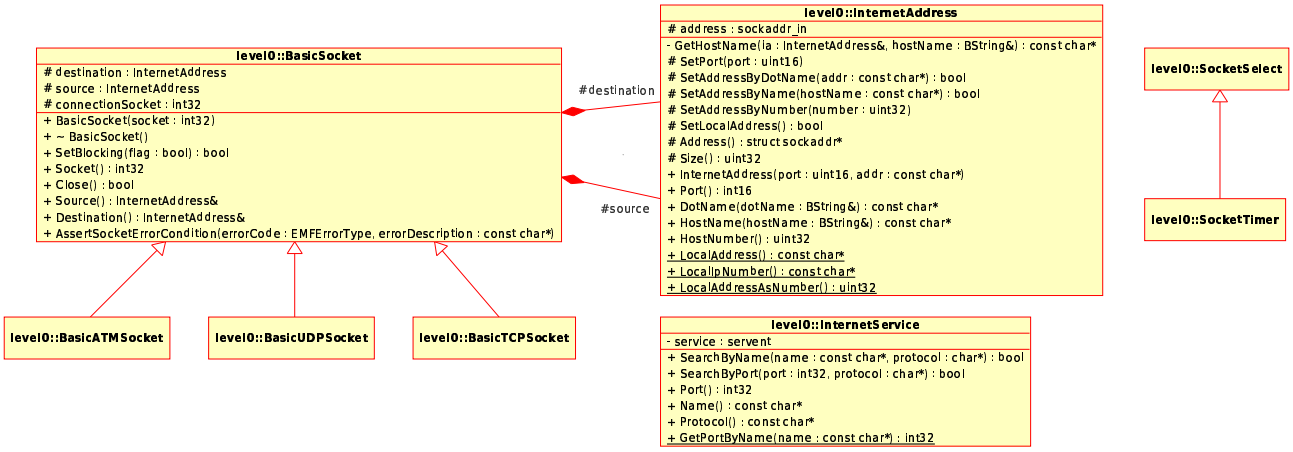

Figure 1.11: BaseLib Level0 network classes

Network classes (Figure 1.11) at these level inherit from the class `BasicSocket`. The class `BasicSocket` treat each source and destination address as a `InternetAddress` structure. We will start examining such class.

Classes in this section:

- InternetAddress
- InternetService
- BasicSocket
- BasicATMSocket
- BasicTCPSocket
- BasicUDPSocket
- SocketSelect
- SocketTimer

InternetAddress

`[InternetAddress.h, InternetAddress.cpp]`
Class `InternetAddress` basically hold the full Internet Protocol version 4.0 (IPv4.0) address (i.e. the IP address and the port number). Right now IPv6.0 is not supported. The class is based on a protected UNIX `sockaddr_in` structure.

```
protected:
   sockaddr_in address;
```

Setting the IP address is achieved by different functions letting friend classes and subclasses set it by dotted number notation (i.e. 192.168.1.2), name (i.e www.google.co.uk), by 32bit number (i.e. usually in hexadecimal format) and automatically, using the localhost IP; also the port number is not publically settable.

```
protected:
   bool SetAddressByDotName(const char* addr);
   bool SetAddressByName(const char* hostName);
   void SetAddressByNumber(uint32 number);
   bool SetLocalAddress();

   void SetPort(uint16 port);
```

Next code lines completes class's interface; the constructor take the port number and the IP in a suitable format to be given in input at the `inet_addr` POSIX API; such function use the numbers-and-dots notation. Public methods let the user only get the IP in different formats, public methods don't allow a user to modify the IP and port number the class holds.

```
   struct sockaddr *Address();
   uint32 Size();
public:
   InternetAddress(uint16 port=0,const char* addr=NULL);

   int16 Port();
   uint32 HostNumber();
   static const char* LocalAddress();
   static const char* LocalIpNumber();
   static uint32 LocalAddressAsNumber();
```

The last two functions use the special `BString` buffer class also defined in directory *level0*. The function `DotName` returns the host name in numbers-and-dots notation the returned `const char*` is a pointer to the `BString` buffer; the function `HostName` returns the host name (as a pointer to the BString buffer) by querying the name server, NULL means failure.

```
   const char* DotName(BString &dotName);
   const char* HostName(BString &hostName);
```

BasicSocket

`[BasicSocket.h, BasicSocket.cpp]`
This class implements generic socket functions. The socket abstraction require to specify an `InternetAddress` destination and a source. Socket is an abstraction that comes from POSIX standars; using the *libc* library, each opened socket, like each opened file in a program, has associated a unique integer.

```
protected:
   InternetAddress destination;
   InternetAddress source;
   int32           connectionSocket;
```

This class is an abstract class defining only the concept behind Sockets: it defines three basic attributes but doesn't implements methods to set or get those attributes, this is not absolutely true but for example you can get the `destination` and `source` address but not set. The constructor require a socket identifier, so this class doesn't create a socket for you, is the user that have to first create a socket and than associate a classes to such socket.

```
public:
   BasicSocket(int32 socket = 0);

   bool SetBlocking(bool flag);
```

```
int32 Socket();
bool Close();
InternetAddress& Source();
InternetAddress& Destination();

void AssertSocketErrorCondition(EMFErrorType errorCode,const char* errorDescription=NULL,...);
void AssertSocketErrorCondition(EMFErrorType errorCode,const char* errorDescription=NULL,...);
```

Last two lines implement error handling for the first time. The enumerated type `EMFErrorType` follows in the section Exceptions in this chapter; `errorCode` is a number identifying the type of error not the dangerous level of the error.

BasicATMSocket

`[BasicATMSocket.h, BasicATMSocket.cpp]`
This class is one of the three subclasses of the `BasicSocket` class. Basically it adds the support to ATM socket at AAL5 protocol. An ATM network is not socket oriented but it relay on channels (Virtual Circuits). Unix developer try to uniformly this kind of network letting ATM network users using the same socket abstraction used for connection oriented protocols.

The most important information regarding a Virtual Circuit is the Virtual Circuit Index (VCI).

```
protected:
   uint32 VCI;
```

This code use the POSIX interface to write and read to an ATM network card. Core functions to read or write are private and are declared as follow. This solution is adopted to let a set of friends function calling the methods have access to protected attributes of `InternetAddress`. A C interface that use the object is given.

```
private:
   bool _Read(const void* buffer,uint32& size,TimeoutType msecTimeout);
   bool _Write(const void* buffer,uint32& size,TimeoutType msecTimeout);
```

Like a `BasicSocket` the constructor doesn't create a socket but it is the user that must supply a newly created socket for the constructor. Using the `Open` method it is possible to open a new ATM socket without passing it via the constructor, `SetVCI` and `GetVCI` let's setting and getting the Virtual Channel Index (a number).

The `Read` and `Write` function reads and writes a block of data, `size` is the maximum size in bytes of the required transfer for a write operation; during a read `size` holds first the maximum buffer size and after the read the readed size in bytes. Timeout is currently not supported.

```
public:
   BasicATMSocket(int32 socket);

   bool Open();
   inline bool Read(void* buffer, uint32& size, TimeoutType msecTimeout=TTDefault);
   inline bool Write(const void* buffer, uint32& size, TimeoutType msecTimeout=TTDefault);
   inline bool SetVCI(int32 VCI);
   inline uint32 GetVCI();
```

To use an ATM oriented channel using such class interface it is necessary to follow those steps, in that order:

1. create a brand new `BasicATMSocket` object;
2. call method `Open` on the object;
3. call method `SetVCI` on the object;
4. now it is possible to read and wirte on the object.

BasicTCPSocket

[BasicTCPSocket.h]
Implements general TCP/IP stream-sockets. All methods implemented in this class are public, no private attribute or method. There are three IO functions: one to write on the stream and two to read from, the second, `Peek` doesn't consume what it reads.

```
public:
   bool BasicWrite(const void* buffer, uint32& size);
   bool BasicRead(void* buffer, uint32& size);
   bool Peek(void* buffer, uint32& size);
```

The constructor take as an argument a socket. If you doesn't want to supply a socket the function `Open` create one for you. `Listen` functions bind the socket on a given port number for incoming connections, the function `WaitConnection` return `BasicTCPSocket*` on newly estabilished connection on the binded socket. `Connect` functions let the user connect to some other remote or local sockets.

```
   BasicTCPSocket(int32 socket = 0);
   bool Open();

   bool Listen(int port,int maxConnections=1);
   bool Listen(char *serviceName,int maxConnections=1);
   BasicTCPSocket* WaitConnection(TimeoutType msecTimeout=TTInfiniteWait,
      BasicTCPSocket* client=NULL);

   bool Connect(const char* address,int port,TimeoutType msecTimeout=TTInfiniteWait);
   bool Connect(const char* address,const char* serviceName,
      TimeoutType msecTimeout=TTInfiniteWait);
   bool IsConnected();
```

This interface permit to specify ports as a number or as a service name (string) this is achieved using the `InternetService::GetPortByName` facility. The `InternetService` interface is reported here:

```
class InternetService {
   servent service;
public:
   bool SearchByName(const char* name,char* protocol=NULL);
   bool SearchByPort(int32 port,char* protocol=NULL);
   int32 Port();
   const char* Name();
   const char* Protocol();
   static int32 GetPortByName(const char* name);
};
```

BasicUDPSocket

[BasicUDPSocket.h]
Implements a general UDP/IP socket. UDP protocol is connectionless and so differently from TCP doesn't need a connection, the `BasicUDPSocket` class implements also the `Connect` function that is used to select the destination of the next sends; infact using the IO function below is not possible to choose a destination address. Every message received on the selected UDP port is received.

```
public:
   bool BasicRead(void* buffer, uint32& size);
   bool BasicWrite(const void* buffer, uint32& size);
```

As usual the constructor let you create an object using a previous opened UDP socket or create a brand new socket with the `Open` method.

```
   BasicUDPSocket(int32 socket = 0);
   bool Open();

   bool Listen(int port,int maxConnections=1);
```

```
   bool Connect(const char *address,int port);
   bool Connect(InternetAddress &dest);
```

SocketSelect

[SocketSelect.h]
This class provide a wrapper to the `select` *libc* function allowing the synchronization of a group of sockets. This stuff can only select sockets from a single source type, is not allowed to mix ATM and UDP on NT platforms.

```
protected:
   fd_set readFDS;
   fd_set writeFDS;
   fd_set exceptFDS;
   fd_set readFDS_done;
   fd_set writeFDS_done;
   fd_set exceptFDS_done;
   int32 readySockets;

public:
   SocketSelect();

   void Reset();
   void AddWaitOnWriteReady(BasicSocket* s);
   void DeleteWaitOnWriteReady(BasicSocket* s);
   void AddWaitOnReadReady(BasicSocket* s);
   void DeleteWaitOnReadReady(BasicSocket* s);
   void AddWaitOnExceptReady(BasicSocket* s);
   void DeleteWaitOnExceptReady(BasicSocket* s);

   bool Wait(TimeoutType msecTimeout=TTInfiniteWait);
   bool WaitRead(TimeoutType msecTimeout=TTInfiniteWait);
   bool WaitWrite(TimeoutType msecTimeout=TTInfiniteWait);
   bool WaitExcept(TimeoutType msecTimeout=TTInfiniteWait);

   int32 ReadySockets();
   bool CheckRead(BasicSocket* s);
   bool CheckWrite(BasicSocket* s);
   bool CheckExcept(BasicSocket* s);
   fd_set& ReadFDS();
   fd_set& WriteFDS();
   fd_set& ExceptFDS();
```

SocketTimer

[SocketTimer.h]
To simplify some use of select; this is a tool to specify a maximum wait on socket actions. The interface is quite simple. The most important thing is the time attribute `timeWait` that model the time to wait in a timeout operation. It is possible to wait for a *read* or *write* operation.

```
private:
   timeval timeWait;

public:
   void SetMsecTimeout(TimeoutType msecTimeout=TTInfiniteWait);
   bool WaitRead();
   bool WaitWrite();
   uint32 MSecTimeout();
```

1.4.1 Design Notes

Sockets are a powerfull abstraction due to UNIX operating system. The construction adopted here is quite general and is fully compatible with many operating systems on the market.

Probably further code to support newer protocol like IPv6.0 is needed in the future. The class diagram using the class developed require that all the system relay on IP, that is not true for ATM networks, it lacks some generality.

Too many code inlining is not ever a good design choice: systems with low memory constraints will became fast a difficult target to support. Code inlining produce many copy of the same code in the library.

1.5 Files, Streams

This section groups togheter two basic concepts: files and streams. For each concepts there is a subsection. There are only few classes involved in this group. If you see Files and Streams implementations you can note that the interface is quite similar between `BasicFile` and `StreamInterface`.

1.5.1 Files

Files abstraction come from UNIX OSes and are nowadays the principal abstraction for data storage. BaseLib's file classes are depicted in Figure 1.12; there are only two classes, without any link, as listed below:

- BasicFile
- BasicConsole

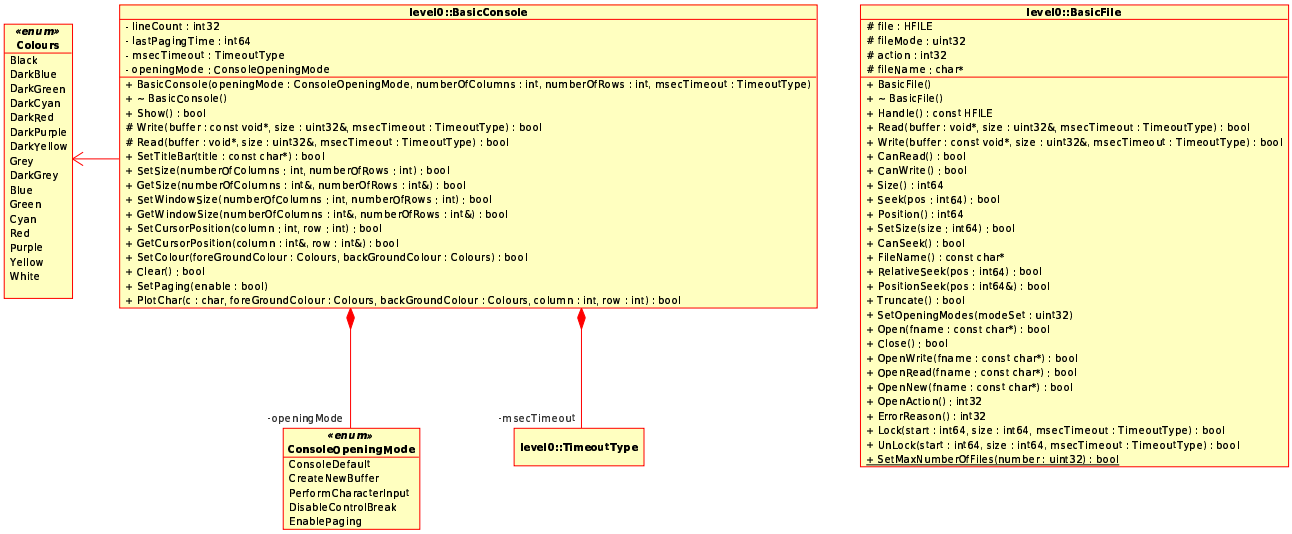

Figure 1.12: BaseLib Level0 file classes

Classes methods are quite similar, this is why they are grouped together, a Console or better a terminal, is in UNIX treated as a file with a more special interface support to terminal (see `man termios`). Infact during the past there were a great use of serial link and a serial link let computer dial togheter, a serial link in a UNIX system is a file and so a serial device is a file (as an example

/dev/ttyS0) serial link has some special character (or set of it) that let the user treat it as a terminal so the UNIX developer introduced those special function as extended functions of a file abstraction. Infact in UNIX a Console can also be locked but in BaseLib no.

BasicFile

[BasicFile.h, BasicFile.cpp]
The BasicFile class is istantiated per opened file the basic information carried from the class is a HFILE handle to the file, a mode mask that can be queried to know if the file was opened in write or read only mode (fileMode), an int32 attribute (action) used to hold an error code in case of failure and the file name (fileName).

```
protected:
   HFILE file;
   uint32 fileMode;
   int32 action;
   char* fileName;
```

There is only one protected method SetFileName that let the developer of subclasses to change the file's name. The class constructor doesn't require any parameters.

Basic *get* methods are Handle that return the OS related file handle (an integer in UNIX); FileName return the name of the file and Size return the size of the file object. The method SetSize lets you set the file size by either truncating or extending the file size. The only static method, SetMaxNumberOfFiles try to set a Operating System wide parameter of the maximum number of files opened in the system, in VxWorks it simply return true.

```
   void SetFileName(const char* name);
public:
   const HFILE Handle();
   inline const char* FileName();

   inline int64 Size();
   inline bool SetSize(int64 size);

   static inline bool SetMaxNumberOfFiles(uint32 number);
```

Before opening a file the first action to do is to set the file opening modes with the SetOpeningModes, opening modes are listed in the firt 50's row of the file *level0/BasicFile.h*. Then it is possible to open a file with the method Open. To avoid the burden of making two calls it is possible to use the methods OpenWrite OpenRead OpenNew to open for write, open read only and create a file. If during an open operation happen something unexpected it's possible to query about what happens with the OpenAction and ErrorReason methods.

```
   inline void SetOpeningModes(uint32 modeSet);
   bool Open(const char *fname,...);
   inline bool Close();

   bool OpenWrite(const char *fname,...);
   bool OpenRead(const char *fname,...);
   bool OpenNew(const char *fname,...);

   inline int32 OpenAction();
   int32 ErrorReason();
```

The Read and Write operation have a timeout to stop blocking the caller processor. The functions have the same interface, the size in each case is readed and written so its an in/out argument. Query methods CanRead and CanWrite parse the attribute fileMode to understand with which options the file was opened. There is no way to understand if the file was just opened or newer opened.

```
inline bool Read(void* buffer, uint32& size, TimeoutType msecTimeout=TTDefault);
inline bool Write(const void* buffer, uint32& size, TimeoutType msecTimeout=TTDefault);
inline bool CanRead();
inline bool CanWrite();
```

For random file access BaseLib supplies `Position` method to know the current file position; with the `CanSeek` method it possible to query regarding the capability of seeking a file (basically an OS's dependent task). Seeking a file is easily obtained using `Seek` to move to a specific absolute position (i.e. the position is expressed from the beginning of the file), `PositionSeek` method behave in the same way but the `pos` is an in/out argument; if you require to move to a relative position simply use `RelativeSeek`.

The method `Truncate` clip the file size to the current seek point (i.e. the current position). The last two function let the user lock and unlock to the application the access of the file region starting from `start` and `size` bytes long. If it was locked wait as much as timeout.

```
inline int64 Position(void);

inline bool CanSeek();
inline bool Seek(int64 pos);
inline bool PositionSeek(int64 &pos);
inline bool RelativeSeek(int64 pos);

inline bool Truncate();

inline bool Lock(int64 start,int64 size,TimeoutType msecTimeout = TTInfiniteWait);
inline bool UnLock(int64 start,int64 size,TimeoutType msecTimeout = TTInfiniteWait);
```

BasicConsole

`[BasicConsole.h, BasicConsole.cpp]`
As stated before a *console* can be treated as a *file* entity but in BaseLib are completely separated but there are some common methods with the same signature. An example of a console in Microsoft Windows© is the Command Prompt in Linux and other UNIX systems you can experienced differents sort of consoles.

Attributes start with some OS's handle different from OS to OS, below are cut and pasted only Window's and Linux's handles. `lineCount` count how many lines since last paging and `lastPagingTime` how long since last paging; `msecTimeout` represents how long to wait when reading on the console, `openingMode` sets of flags describing the console status.

```
#if defined (_WIN32)
   HANDLE inputConsoleHandle;
   HANDLE outputConsoleHandle;
#elif defined (_LINUX)
   struct termio originalConsoleModes;
#endif
   int32 lineCount;
   int64 lastPagingTime;
   TimeoutType msecTimeout;
   ConsoleOpeningMode openingMode;
```

The constructor takes four arguments: the first is of type `ConsoleOpeningMode`, an `enum` (defined in *level0/BasciConsole.h*) that defines how to open the console (can be an orred combination); the second and the third arguments define respectively the number of columns and rows of the opened console; the last argument is a timeout that update the attribute `msecTimeout`. `Show` simply try to show the console; basically it call an OS's API.

Protected methods let a subclass to write and read to the console; only a subclass is able to make IO operations on a `BasicConsole` this require to extend a `BasicConsole` to have a real working *console*.

```
public:
   BasicConsole(ConsoleOpeningMode openingMode=ConsoleDefault,
      int numberOfColumns=-1,
      int numberOfRows=-1,
      TimeoutType msecTimeout=TTInfiniteWait);
   virtual ~BasicConsole();

   inline bool Show();
protected:
   inline bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout);
   inline bool Read(void* buffer,uint32& size,TimeoutType msecTimeout);
```

The following methods are oriented to graphical operating systems were a *console* is usually displayed in a *window*. The first method (`SetTitleBar`) infact lets you set the title bar text of a console window; `SetPaging` enable or disable paging in the console window; `SetColour` sets the font foreground and background colours. Take a look to the `enum Colours` in Figure 1.12 to see the currently available colours.

To set and retrieve the size of the *console buffer* in columns and rows you must use `SetSize` and `GetSize`; to set and retrieve the size of the *console window* in columns and rows there are `SetWindowSize` and `GetWindowSize`. Last couple of functions is full implemented only in Microsoft Windows©. Figure 1.13 explain the differences between the *console window* and *console buffer*.

The method `SetCursorPosition` sets the position of the cursor in columns and rows; `GetCursorPosition` retrieve the cursor position. `Clear` clears the console's content. `PlotChar` write a single char on the console at a given position and with a given colour set.

```
public:
   inline bool SetTitleBar(const char* title);
   inline void SetPaging(bool enable);
   inline bool SetColour(Colours foreGroundColour,Colours backGroundColour);

   inline bool SetSize(int numberOfColumns,int numberOfRows);
   inline bool GetSize(int& numberOfColumns,int& numberOfRows);

   inline bool SetWindowSize(int numberOfColumns,int numberOfRows);
   inline bool GetWindowSize(int& numberOfColumns,int& numberOfRows);

   inline bool SetCursorPosition(int column, int row);
   inline bool GetCursorPosition(int& column,int& row);

   inline bool Clear();
   inline bool PlotChar(char c,
      Colours foreGroundColour,
      Colours backGroundColour,
      int column, int row);
```

1.5.2 Streams

The stream abstraction come from the Object Oriented Design work. C++ defines its standard streams, have a look at `iostream` and `fstream` but BaseLib redefine them. In this group of classes we explore the following classes (depicted in Figure 1.14):

- StreamInterface
- PrintfStreamInterface
- CStream, CStreamNewBufferFN

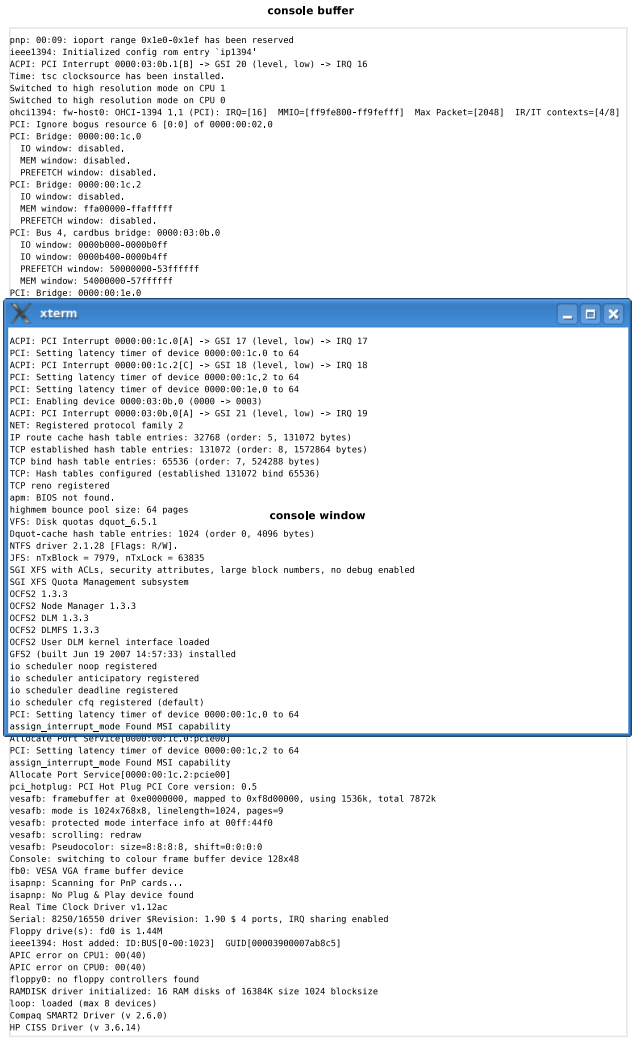

Figure 1.13: BaseLib console scheme

StreamInterface

`[StreamInterface.h, StreamInterface.cpp]`
The StreamInterface is really important: an entire BaseLib's Level is build on it. Such level focuses on streams defining the `Streamable` object on top of the `StreamInterface` that is abstract version. It is used to allow referring to streams at lower levels. The class has many *pure virtual* methods and a few inline methods so its an abstract C++ class.

The method `Read` reads data into `buffer` as much as `size` byte are written, actual size is returned in `size`; `msecTimeout` argument is how much the operation should last, timeout behaviour is class specific, i.e. sockets with blocking activated wait forever when `noWait` is used. The method `Write` writes data from `buffer` to the stream as much as `size` byte are written, actual size is returned in `size`, `msecTimeout` is how much the operation should last; also here timeout behaviour is class specific.

Methods `CanRead` and `CanWrite` let you know about the capability to read or write on the stream. `PutC` implements a single character write operation, `GetC` implements single character read operation.

```
public:
   virtual bool Read(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault)=0;
   virtual bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault)=0;
```

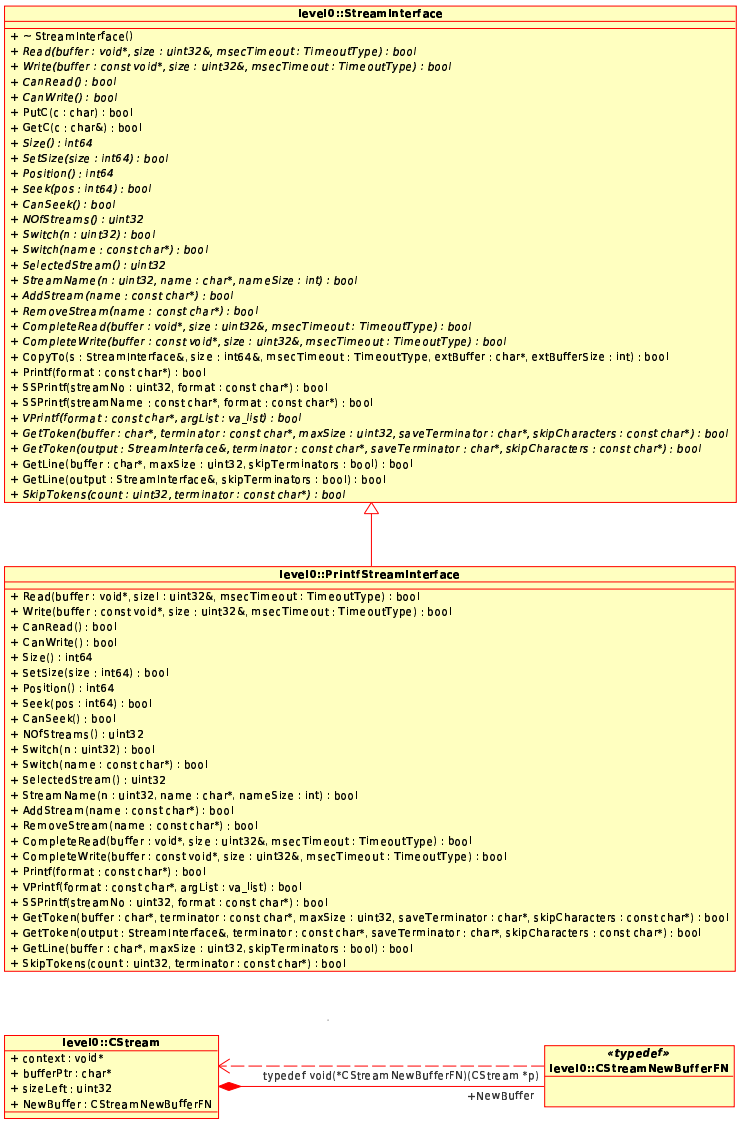

Figure 1.14: BaseLib Level0 base stream classes

```
virtual bool CanRead()=0;
virtual bool CanWrite()=0;

inline bool PutC(char c);
inline bool GetC(char &c);
```

The method `Size` return the size of the stream and `SetSize` clip the stream size to a specified point (similar to `Truncate` method). `Position` returns current position, `Seek` moves within the file to an absolute location in the file and `CanSeek` let you query about the capability of seeking.

```
virtual int64 Size()=0;
virtual bool SetSize(int64 size)=0;

virtual int64 Position(void)=0;
virtual bool Seek(int64 pos)=0;
virtual bool CanSeek()=0;
```

Following methods define a multiple streams interface, `NOfStreams` return how many streams are available, `Switch` select the stream to read from, switching may reset the stream to the start, the same method overloaded let the user `Switch` by name to another stream. `SelectedStream` return the index

of the currently used stream, using `StreamName` it possible to retrieve also the stream name. Using `AddStream` you can add a new stream to write to, removing is possible with `RemoveStream`. Those functions used togheter lets manage streams registration and cancellation.

```
virtual uint32 NOfStreams()=0;
virtual bool Switch(uint32 n)=0;
virtual bool Switch(const char* name)=0;
virtual uint32 SelectedStream()=0;
virtual bool StreamName(uint32 n,char* name,int nameSize)=0;
virtual bool AddStream(const char* name)=0;
virtual bool RemoveStream(const char* name)=0;
```

The method `CompleteRead` performs the job of a `Read` function but gurantees the completion; in case of failure `size` returns the actual data readed, `msecTimeout` is the total allowed wait time checked using `HRT`. The method `CompleteWrite` performs the job of a the `Write` function but gurantees the completion; in case of failure `size` returns the actual data written, `msecTimeout` is the total allowed wait time checked using `HRT`. `CopyTo` copies `size` bytes to the stream `s` it stops on `msecTimeout` or on an EOF, returns `True` on success or on reaching EOF (if `msecTimeout` is `TTInfiniteWait`) otherwise `False`.

`Printf` print a formatted string on the current selected stream, supported format flags are: "o d i X x Lo Ld Li LX Lx f e s c". The first method `SSPrintf` select a stream and writes to it, then returns to previous buffer is a zero terminated string, second method `SSPrintf` creates a stream with `AddStream` and writes into it; `VPrintf` is a `Printf` with vararg format.

First `GetToken` method extracts a token from the stream into a string data until a terminator or 0 is found; `maxSize` is the buffer size, the maximum string size is `maxSize-1`; it skips all `skipCharacters` chars even if calssified also as terminators if at the beginning returns `true` if some data was read before any error or file termination. `false` only on error and no data available; the terminator (just the first encountered) is consumed in the process and saved in `saveTerminator` if provided. This is a buffered method. Second `GetToken` method behave in the same way as before but take as the first argument an `StreamInterface` object. A character can be found in the terminator or in the `skipCharacters` list in both or in none. `SkipTokens` skips a series of tokens delimited by terminators or 0. `GetLine` will skip an empty line or any part of a line termination (the first working on a `StreamInterface` input and the second on a `char*` input extracting the substring after skipping the required line).

char in	action preformed
none	the character is copied
terminator	the character is not copied the string is terminated
skip	the character is not copied
skip and terminator	the character is not copied, the string is terminated if not empty

Table 1.3: GetToken actions

```
virtual bool CompleteRead(void* buffer,uint32& size,
   TimeoutType msecTimeout=TTInfiniteWait)=0;
virtual bool CompleteWrite(const void* buffer,uint32& size,
   TimeoutType msecTimeout=TTInfiniteWait)=0;
inline bool CopyTo( StreamInterface& s,int64& size,TimeoutType msecTimeout=TTInfiniteWait,
   char* extBuffer=NULL,int extBufferSize=0);

inline bool Printf(const char* format,...);
inline bool SSPrintf(uint32 streamNo,const char* format,...);
inline bool SSPrintf(const char* streamName,const char* format,...);
virtual bool VPrintf(const char *format,va_list argList)=0;
```

```
virtual bool GetToken(char* buffer,const char* terminator,uint32 maxSize,
   char* saveTerminator=NULL,const char* skipCharacters=NULL)=0;
virtual bool GetToken(StreamInterface& output,const char* terminator,
   char* saveTerminator=NULL,const char* skipCharacters=NULL)=0;
virtual bool SkipTokens(uint32 count,const char* terminator)=0;

virtual bool GetLine(char* buffer,uint32 maxSize,bool skipTerminators=True);
virtual bool GetLine(StreamInterface& output,bool skipTerminators=True);
```

PrintfStreamInterface

[PrintfStreamInterface.h]
This is a simple implementation of `StreamInterface` that uses **vprintf** standard function on the standard output. Most of the methods are not implemented and simply return `False`, methods implemented are:

```
virtual bool Printf(const char* format,...){
   if (format==NULL) return False;
   va_list argList;
   va_start(argList,format);
   bool ret = vprintf(format,argList);
   va_end(argList);
   return ret;
}
virtual bool VPrintf(const char* format,va_list argList){
   return (vprintf(format,argList) != 0);
}
virtual bool SSPrintf(uint32 streamNo,const char* format,...){
   if(streamNo == 0){
      if (format==NULL) return False;
      va_list argList;
      va_start(argList,format);
      bool ret = vprintf(format,argList);
      va_end(argList);
      return ret;
   }
   return False;
}
```

CStream, CStreamNewBufferFN

[CStream.h, CStream.cpp]
`CStream` is a portable C stream mechanism. Basically a `CStream` is a C strutct that is a C++ class without methods and with public attributes. BaseLib offer the following C functions to operate on `CStream`s; all of that have the same interface as methods in `StreamInterface` but require a `CStream*` as first argument.

```
bool CRead(CStream* cs,void* buffer,uint32& size);
bool CWrite(CStream* cs,const void* buffer,uint32& size);

bool CPrintInt32(CStream* cs,int32 n,uint32 desiredSize=0,
   char desiredPadding=0,char mode='i');
bool CPrintInt64(CStream* cs,int64 n,uint32 desiredSize=0,
   char desiredPadding=0,char mode='i');
bool CPrintDouble(CStream* cs,double ff,int desiredSize=0,
   int desiredSubSize=6,char desiredPadding=0,char mode = 'f');
bool CPrintString(CStream* cs,const char* s,uint32 desiredSize=0,
   char desiredPadding=0, bool rightJustify = True);

bool VCPrintf(CStream* cs,const char* format,va_list argList);
bool CPrintf(CStream* cs,const char* format,...);
```

```
bool CGetToken(CStream* cs,char* buffer,const char* terminator,uint32 maxSize,
   char* saveTerminator=NULL,const char* skip=NULL);
bool CGetCStringToken(const char*& input,char* buffer,const char* terminator,uint32 maxSize);
char *CDestructiveGetCStringToken(char*& input,const char* terminator,
   char* saveTerminator=NULL,const char* skip="");
bool CGetCSToken(CStream* csIn,CStream* csOut,const char* terminator,
   char* saveTerminator=NULL,const char* skip=NULL);

bool CSkipTokens(CStream* cs,uint32 count,const char* terminator);
```

BaseLib2 add a custom implementation of the C `vprintf` function, around this implementation are coded the four function that follow (`bl2_` prefix is instead of BaseLib2).

```
int bl2_vsprintf(char* buffer,const char* format,va_list argList);
int bl2_sprintf(char* buffer,const char* format,...);
int bl2_vsnprintf(char* buffer,size_t size,const char* format,va_list argList);
int bl2_snprintf(char* buffer,size_t size,const char* format,...);
```

1.5.3 Design Notes

A good idea is to redesign this section as in UNIX operating system with console as a subclass of file. This can lead to more comphrension of the modules design defining many common functionality togheter. But infact BaseLib was designed starting from Microsoft Windows and OS/2.

In class `BasicFile` there is no way to understand if the file was just opened or newer opened: the file creation or open must be accomplished in the constructor or must be added a variable that let you check for action just performed.

There are too many methods in `StreamInterface`.

1.6 Errors, Exceptions

There are only two classes in this section, basically there is a class about error management (`ErrorManagement`) and a class about exception management (`ExceptionHandlerInterface`). Figure 1.15 depicted the class's scheme.

There are a great number of `enum`: `EMFErrorType` (in *level0/ErrorManagement.h*), `EMFErrorBehaviour` (in *level0/ErrorManagement.h*)and `ExceptionHandlerBehaviour` (in *level0/ExceptionHandlerDefinition.h*).

ErrorManagement

`[ErrorMamagemet.h, ErrorMamagemet.cpp]`
File *level0/ErrorManagement.h* basically is a soup of C functions that print out formatted strings on some stream like for example the standard output or on a remote logger. Each function can be used in a precise scenario. Then follow a brief discussion about them.

Function `VCAssertErrorCondition` sets the error status and depending on setup does appropriate action; this call is to be called from static members. Function `VCISRAssertErrorCondition` like the previous one sets the error status and depending on setup does appropriate action, this call is to be called from Interrupt Service Routines. Function `CISRStaticAssertErrorCondition` sets the error status and depending on setup does appropriate action, also such call is to be called from Interrupt Service Routines (it is the same as the previous one except for the number of arguments, in such last function you have a variable argument list.

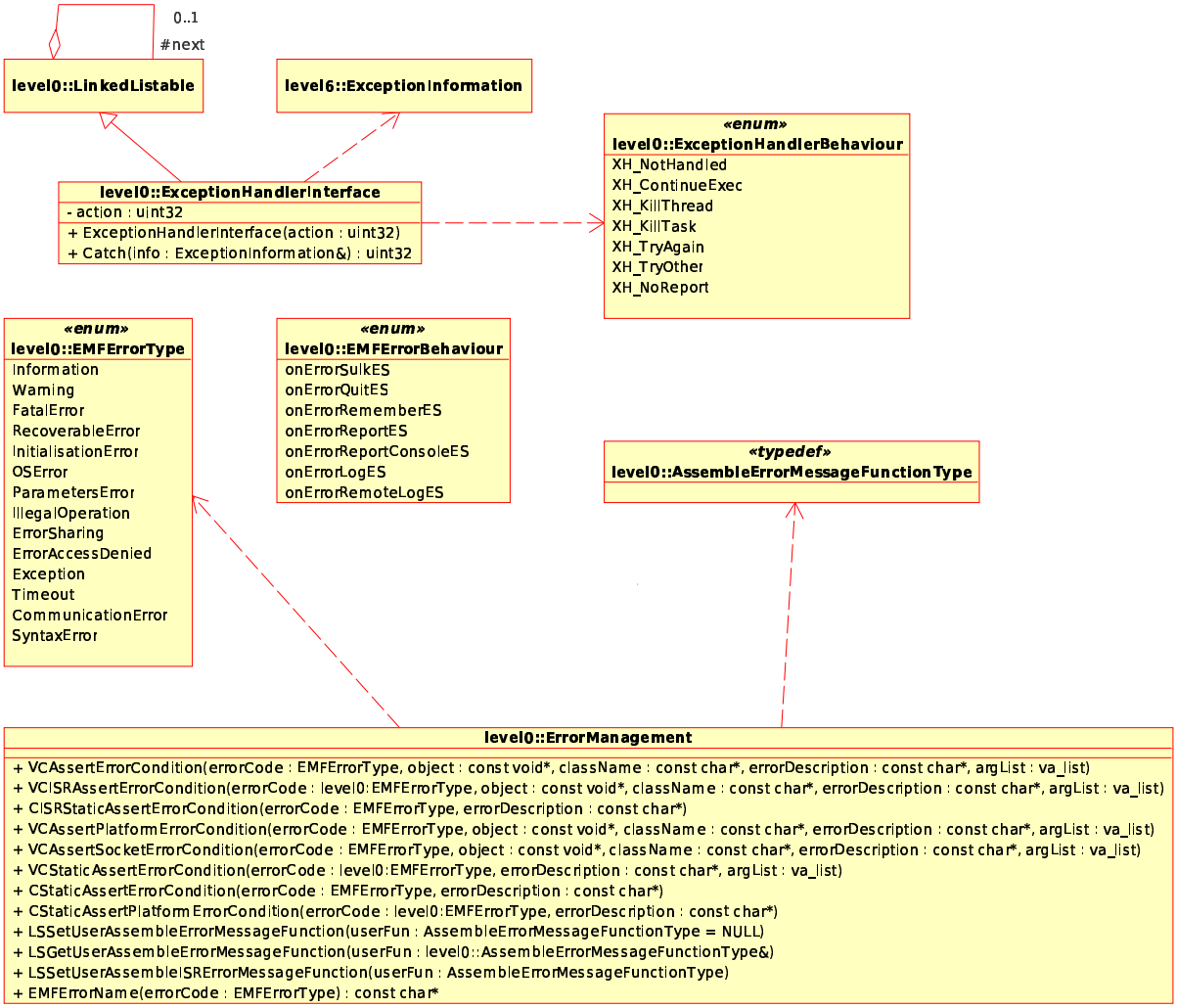

Figure 1.15: BaseLib Level0 errors and exceptions classes

`VCAssertPlatformErrorCondition` sets the error status to *platform error* and depending on setup does appropriate action, `VCAssertSocketErrorCondition` sets the error status to *socket error* and depending on setup does appropriate action.

The function `VCStaticAssertErrorCondition` sets the error status and depending on setup does appropriate action, this call is to be called from static members; function `CStaticAssertErrorCondition` sets the error status and depending on setup does appropriate action, this call is to be called from static members. `CStaticAssertPlatformErrorCondition` sets the error status and depending on setup does appropriate action.

```
void VCAssertErrorCondition(EMFErrorType errorCode,const void* object,
   const char* className,const char* errorDescription,va_list argList);
void VCISRAssertErrorCondition(EMFErrorType errorCode,const void* object,
   const char* className,const char* errorDescription,va_list argList);
void CISRStaticAssertErrorCondition(EMFErrorType errorCode,
   const char *errorDescription,...);

void VCAssertPlatformErrorCondition(EMFErrorType errorCode,const void* object,
   const char* className,const char* errorDescription,va_list argList);
void VCAssertSocketErrorCondition(EMFErrorType errorCode,const void* object,
   const char* className,const char* errorDescription,va_list argList);
```

```
void VCStaticAssertErrorCondition(EMFErrorType errorCode,
   const char* errorDescription,va_list argList);
void CStaticAssertErrorCondition(EMFErrorType errorCode,
   const char* errorDescription,...);
void CStaticAssertPlatformErrorCondition(EMFErrorType errorCode,
   const char* errorDescription,...);
```

Function `LSSetUserAssembleErrorMessageFunction` sets the handler for error messages, function `LSGetUserAssembleErrorMessageFunction` gets a reference to the handler for error messages, `LSSetUserAssembleISRErrorMessageFunction` sets the handler for error messages coming from interrupts. Finally the function `EMFErrorName` translate an error encoded in `EMFErrorType` to an error name in C string format (`NULL`) terminated char array).

```
void LSSetUserAssembleErrorMessageFunction(AssembleErrorMessageFunctionType userFun=NULL);
void LSGetUserAssembleErrorMessageFunction(AssembleErrorMessageFunctionType &userFun);
void LSSetUserAssembleISRErrorMessageFunction(AssembleErrorMessageFunctionType userFun);

const char *EMFErrorName(EMFErrorType errorCode);
```

ExceptionHandlerInterface

`[ExceptionHandlerInterface.h]`
Class `ExceptionHandlerInterface` is an exception handler plugin interface. Is not a real C++ interface. There is only one abstract attribute `action` that defines what to do in case of exception. The constructor let you set the default handling in case of an exception. The method `Catch` return the action to be performed in case of an exception.

```
private:
   uint32 action;
public:
   ExceptionHandlerInterface(uint32 action=XH_NotHandled);
   virtual uint32 Catch(ExceptionInformation& info);
```

1.6.1 Design Notes

Take a look to Figure 1.15 is not a good design choice to have a dependency from level0 to level6: the code must be reorganized. The exception handling pluggable interface is a really nice idea.

1.7 Memory

This section group together classes depicted in Figure 1.16. Those classes have no strong link between them but are all about memory: shared memory handling, at OS level, is done by `SharedMemory` class, basic string management via `BString`; endianity support is handle via `Endianity` and memory basic type conversion is possible using `Intel16`, `IntelU16`, `Intel32`,`IntelU32` classes. Such classes have to be extended also to `Intel64` and `IntelU64` as well.

Some of those classes can be moved to a different category due to the implementation and logical dependency, for example class `SharedMemory` is thightly bounded to the OS so can be moved to *arch* subdirectory, the same can be done for the class `Endianity` but that class has many dependecies in this section. Class `BString` can be moved to *streams* subdirectory.

SharedMemory

`[SharedMemory.h, SharedMemory.cpp]`
This class is the most relevant in this group: it provides function to access shared memory. It holds attributes to the current shared memory address and to the current shared memory size (`memory`, `size`). There are others attributes with a strong dependence on the OS and OS's library, i.e. in Linux the class is implemented using SystemV Shared Memory.

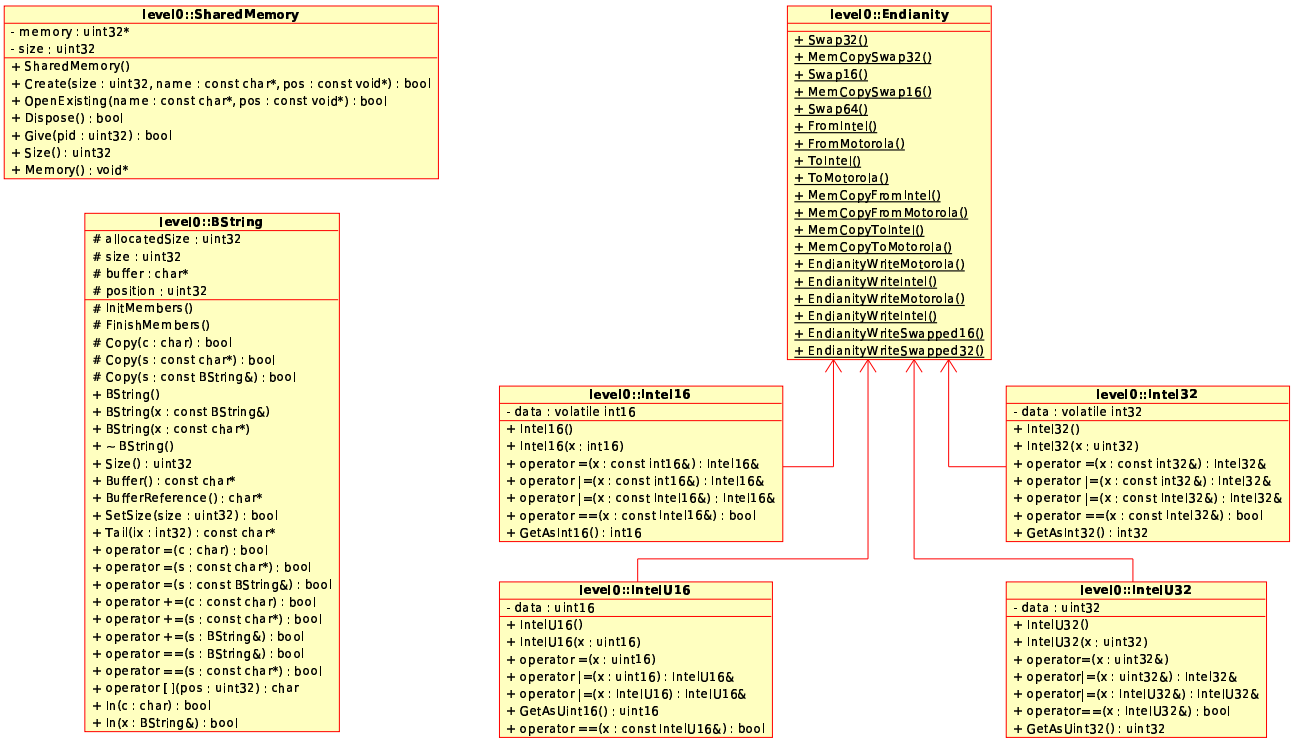

Figure 1.16: BaseLib Level0 memory classes

```
private:
   uint32* memory;
   uint32 size;

#if (defined (_WIN32) || defined(_RSXNT))
   HANDLE handle;
#elif defined(_RTAI)
   unsigned long number;
#elif defined(_LINUX)
   key_t key;
   int id;
#endif
```

The constructor require no arguments and simply initialize the attributes in the class. `Size` simply return the size of the shared memory and `Memory` the start address of the shared memory, no check is done to test if the shared memory was just created or not, i.e. if the shared memory was not created jet the class must return an error.

The method `Create` create a new shared memory buffer by name `name` (in OS/2 the `pos` is disregarded); `OpenExisting` is used to open an existing share memory in the system by name. Last two functions are from OS/2 operating system and are not fully ported to other operating systems. `Dispose` dispose the use of the shared memory and `Give` release the use of the shared memory. If `Create` is used to create a shared memory then `Dispose` is used to return the shared memory to the system. Pay attention that no destructor is coded so is not safe to not return the memory to the system. Difference between `Give` and `Dispose` is not really clear.

```
public:
   SharedMemory();
   uint32 Size();
   void* Memory();
```

```
bool Create(uint32 size,const char* name,const void* pos=0);
bool OpenExisting(const char* name,const void* pos=0);
bool Dispose(void)
bool Give(uint32 pid);
```

BString

[BString.h, BString.cpp]
The class `BString` is the basic implementation of a string, the class want let user treat a string like a file; reading and writing are done on a specific position.

Starting with the attributes, the first attribute, `allocatedSize`, is the size of the allocated memory block for the string; `size` represent the size of the used memory block minus one (it excludes the 0 char); `buffer` holds the memory buffer address and `position` is the currently seek position.

Other protected methods are: `InitMembers` and `FinishMembers` used for constructor and deconstructor. Copy methods (`Copy`) let the user copy a character, a string and a `BString` into the `BString` buffer.

```
protected:
   uint32 allocatedSize;
   uint32 size;
   char* buffer;
   uint32 position;

   void InitMembers();
   void FinishMembers();

   bool Copy(char* c);
   bool Copy(const char* s);
   bool Copy(const BString& s);
```

The first constructor creates an empty string; second constructor is the copy constructor and the last constructor build up a `BString` starting from a C string. Then cames the destructor.

The method `Size` return the size of the string (`size` attribute); `SetSize` clips the string size to a specified point (`size` is expressed from the start of the string). `Buffer` returns a read only access to the internal buffer, `BufferReference` returns a read/write access to the internal buffer and `Tail` returns a read only pointer to the tail of the buffer.

Methods `In` checks if a char is in the string.

```
public:
   inline BString();
   inline BString(const BString& x);
   inline BString(const char* x);
   virtual ~BString();

   inline uint32 Size() const;
   inline bool SetSize(uint32 size);

   inline const char* Buffer() const;
   inline char* BufferReference() const;
   inline const char* Tail(int32 ix ) const;

   inline bool In(char c) const;
   inline bool In(BString& x) const;
```

The following operators redefinition eases the work with strings. First three `operator=` lets sets a `BString` to be a copy of the input parameter; methods `operator+=` address the concatenation between a `BString` and a single char, a C string and a `BString` object; redefinition of `opertator==` lets compare

differents string objects. `operator[]` allows access to character within the buffer, argument `pos` is the position in the buffer to be accessed.

```
inline bool operator=(char c);
inline bool operator=(const char *s);
inline bool operator=(const BString &s);
inline bool operator+=(const char c);
inline bool operator+=(const char *s);
inline bool operator+=(BString &s);
inline bool operator==(BString &s) const;
inline bool operator==(const char *s) const;
inline char operator[](uint32 pos);
```

Endianity

`[Endianity.h]`
The endianity class implements endianity conversion between types. Basically many methods were cut in Figure 1.16 to fit the UML schema in the page. We now explore the methods it offers.

Core methods are first coded and basically do all the work. The first `Swap32` method swaps the 4 bytes in a 32 bit number, i.e. simply changes the endianity; the second `Swap32` swaps 4 bytes in a 32bit vector; `MemCopySwap32` swaps the 4 bytes while copying a vector of 32 bit numbers.

The first `Swap16` method swaps the 2 bytes in a 16 bit number, i.e. changes the endianity; the second `Swap16` swaps 2 bytes in a 16bit vector; `MemCopySwap16` swaps the 2 bytes while copying a vector of 16 bit numbers.

Last `Swap64` method swaps the 8 bytes in a 64 bit number.

```
public:
   static inline void Swap32(volatile void *x);
   static inline void Swap32(volatile void *x,uint32 sizer);
   static inline void MemCopySwap32(volatile void *dest,volatile const void *src,uint32 sizer);

   static inline void Swap16(volatile void *x);
   static inline void Swap16(volatile void *x,uint32 sizer);
   static inline void MemCopySwap16(volatile void *dest,volatile const void *src,uint32 sizer);

   static inline void Swap64(volatile void *x);
```

The following function has a different implementation if compiled on different architecture (infact this class will be moved in *arch*). The first overloaded method `FromMotorola` need in input a number in *Motorola* format (i.e. big endian) and convert it in the currently system endianity format. `FromIntel` need in input a number in *Intel* format (i.e. little endian) and convert it in the currently system endianity format. `ToMotorola` covert the value in big endian and `ToIntel` convert the value in little endian.

All such methods use the before presented methods.

```
   static inline void FromMotorola(volatile double &x);
   static inline void FromMotorola(volatile float &x);
   static inline void FromMotorola(volatile uint64 &x);
   static inline void FromMotorola(volatile uint32 &x);
   static inline void FromMotorola(volatile uint16 &x);
   static inline void FromMotorola(volatile int64 &x);
   static inline void FromMotorola(volatile int32 &x);
   static inline void FromMotorola(volatile int16 &x);

   static inline void FromIntel(volatile double &x);
   static inline void FromIntel(volatile float &x);
   static inline void FromIntel(volatile uint64 &x);
   static inline void FromIntel(volatile uint32 &x);
   static inline void FromIntel(volatile uint16 &x);
   static inline void FromIntel(volatile int64 &x);
```

```
static inline void FromIntel(volatile int32 &x);
static inline void FromIntel(volatile int16 &x);

static inline void ToMotorola(volatile double &x);
static inline void ToMotorola(volatile float &x);
static inline void ToMotorola(volatile uint64 &x);
static inline void ToMotorola(volatile uint32 &x);
static inline void ToMotorola(volatile uint16 &x);
static inline void ToMotorola(volatile int64 &x);
static inline void ToMotorola(volatile int32 &x);
static inline void ToMotorola(volatile int16 &x);

static inline void ToIntel(volatile double &x);
static inline void ToIntel(volatile float &x);
static inline void ToIntel(volatile uint64 &x);
static inline void ToIntel(volatile uint32 &x);
static inline void ToIntel(volatile uint16 &x);
static inline void ToIntel(volatile int64 &x);
static inline void ToIntel(volatile int32 &x);
static inline void ToIntel(volatile int16 &x);
```

The group of methods that comes are really equivalent to the ones above but instead of swapping a single value they swap an entire vector of values. The signature of methods is fully reported here.

```
static inline void MemCopyFromMotorola(float *dest,const float *src,uint32 size);
static inline void MemCopyFromMotorola(uint32 *dest,const uint32 *src,uint32 size);
static inline void MemCopyFromMotorola(uint16 *dest,const uint16 *src,uint32 size);
static inline void MemCopyFromMotorola(int32 *dest,const int32 *src,uint32 size);
static inline void MemCopyFromMotorola(int16 *dest,const int16 *src,uint32 size);

static inline void MemCopyFromIntel(float *dest,const float *src,uint32 size);
static inline void MemCopyFromIntel(uint32 *dest,const uint32 *src,uint32 size);
static inline void MemCopyFromIntel(uint16 *dest,const uint16 *src,uint32 size);
static inline void MemCopyFromIntel(int32 *dest,const int32 *src,uint32 size);
static inline void MemCopyFromIntel(int16 *dest,const int16 *src,uint32 size);

static inline void MemCopyToMotorola(float *dest,const float *src,uint32 size);
static inline void MemCopyToMotorola(uint32 *dest,const uint32 *src,uint32 size);
static inline void MemCopyToMotorola(uint16 *dest,const uint16 *src,uint32 size);
static inline void MemCopyToMotorola(int32 *dest,const int32 *src,uint32 size);
static inline void MemCopyToMotorola(int16 *dest,const int16 *src,uint32 size);

static inline void MemCopyToIntel(float *dest,const float *src,uint32 size);
static inline void MemCopyToIntel(uint32 *dest,const uint32 *src,uint32 size);
static inline void MemCopyToIntel(uint16 *dest,const uint16 *src,uint32 size);
static inline void MemCopyToIntel(int32 *dest,const int32 *src,uint32 size);
static inline void MemCopyToIntel(int16 *dest,const int16 *src,uint32 size);
```

Last methods, that are to be moved to streams, all use the `fwrite` function (binary stream output) but does the same things as the one above.

```
static inline void EndianityWriteMotorola(int8 &x,FILE *f);
static inline void EndianityWriteMotorola(int16 &x,FILE *f);
static inline void EndianityWriteMotorola(int32 &x,FILE *f);
static inline void EndianityWriteMotorola(uint8 &x,FILE *f);
static inline void EndianityWriteMotorola(uint16 &x,FILE *f);
static inline void EndianityWriteMotorola(uint32 &x,FILE *f);

static inline void EndianityWriteIntel(int8 &x,FILE *f);
static inline void EndianityWriteIntel(int16 &x,FILE *f);
static inline void EndianityWriteIntel(int32 &x,FILE *f);
static inline void EndianityWriteIntel(uint8 &x,FILE *f);
static inline void EndianityWriteIntel(uint16 &x,FILE *f);
static inline void EndianityWriteIntel(uint32 &x,FILE *f);
```

```
   static inline void EndianityWriteMotorola(int8 *x,uint32 n,FILE *f);
   static inline void EndianityWriteMotorola(int16 *x,uint32 n,FILE *f);
   static inline void EndianityWriteMotorola(int32 *x,uint32 n,FILE *f);
   static inline void EndianityWriteMotorola(uint8 *x,uint32 n,FILE *f);
   static inline void EndianityWriteMotorola(uint16 *x,uint32 n,FILE *f);
   static inline void EndianityWriteMotorola(uint32 *x,uint32 n,FILE *f);

   static inline void EndianityWriteIntel(int8 *x,uint32 n,FILE *f);
   static inline void EndianityWriteIntel(int16 *x,uint32 n,FILE *f);
   static inline void EndianityWriteIntel(int32 *x,uint32 n,FILE *f);
   static inline void EndianityWriteIntel(uint8 *x,uint32 n,FILE *f);
   static inline void EndianityWriteIntel(uint16 *x,uint32 n,FILE *f);
   static inline void EndianityWriteIntel(uint32 *x,uint32 n,FILE *f);

   static void EndianityWriteSwapped16(void *x,uint32 n,FILE *f);
   static void EndianityWriteSwapped32(void *x,uint32 n,FILE *f);
```

Intel16, IntelU16, Intel32, IntelU32

`[Intel16.h, IntelU16.h, Intel32.h, IntelU32.h]`
Those classes implements a 16 bit integer in little endian encoding, a 16 bit unsigned int in little endian encoding, a 32 bit integer in little endian encoding and a 32 bit unsigned integer in little endian encoding.

Basically they all have the same interface with the same methods, constructors and operators ridefinition. We just cut and past the class definition of `Intel16`.

```
class Intel16 {
private:
   volatile int16 data;
public:
   inline Intel16();
   inline Intel16(int16 x);

   inline Intel16& operator= (const int16& x);
   inline Intel16& operator|= (const int16& x);
   inline Intel16& operator|= (const Intel16& x);
   inline bool operator == (const Intel16& x);

   inline int16 GetAsInt16();
};
```

1.7.1 Design Notes

This section doesn't need to much notes because classes are probably not well grouped together. Some class can be moved to another groups. It is really important that the `SharedMemory` class has a distructor that will destroy the give back the shared memory helded to keep resource consumption under control.

1.8 Processes, Threads

ThreadInitialisationInterface

`[ThreadInitialisationInterface.h]`
This class stores information associated with a specific thread. Such class is an interface used to implement a common thread initialisation procedure; despite the name `ThreadInitialisationInterface` it also provides a minimal set of functionalities; for this reason it has not been made pure virtual and can be istantiated. Let's take a look at its attributes and methods.

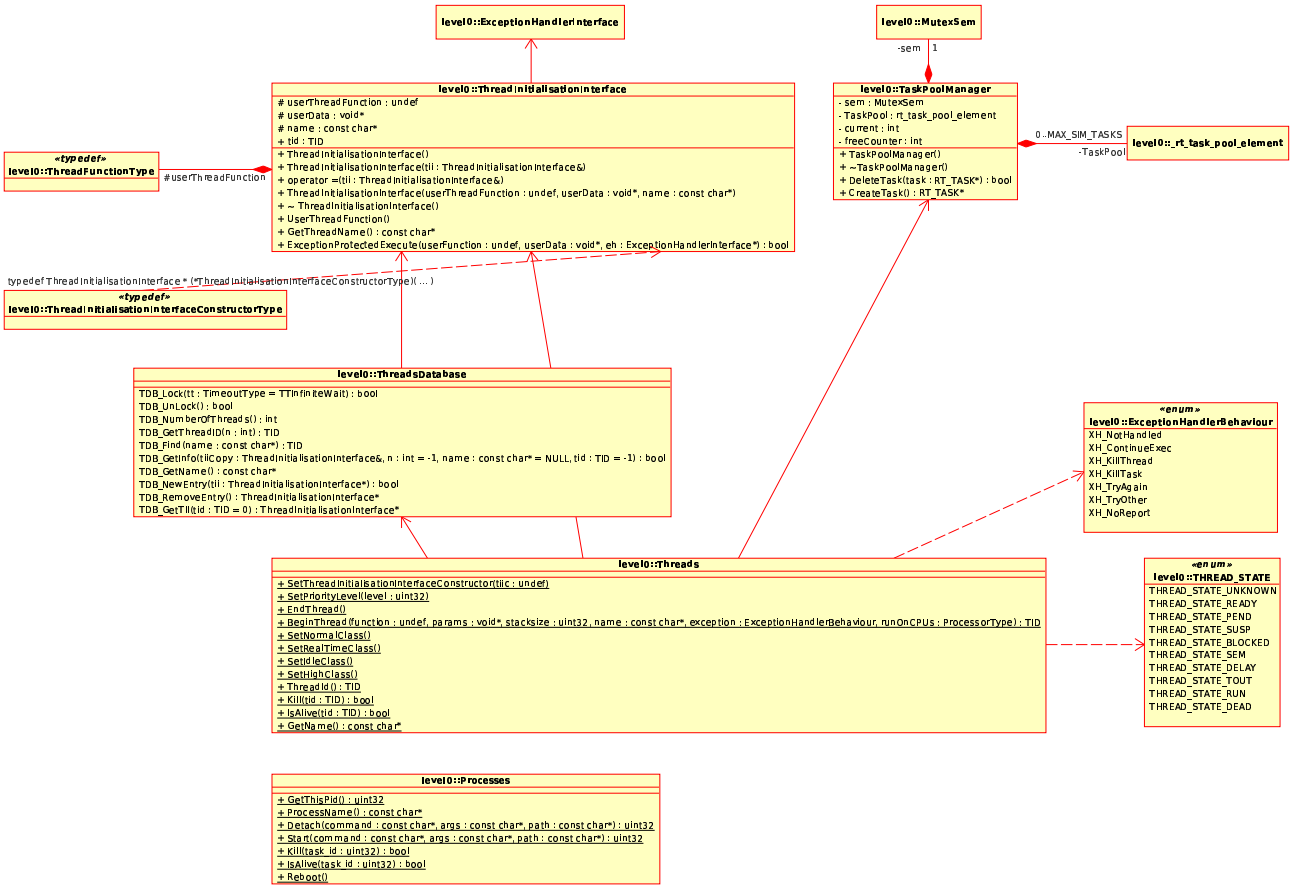

Figure 1.17: BaseLib Level0 proc classes

The type of a function that can be used for a thread is typedefined in *level0/ThreadInitialistionInterface.h*

```
typedef void (*ThreadFunctionType)(void *parameters);
```

The attribute `userThreadFunction` is of type `ThreadFunctionType` and is the holder of a thread entry point. `userData` is a pointer to a structure containing thread data and *name* is the name of the thread. Last attribute, a `public` one holds the thread number.

```
protected:
   ThreadFunctionType userThreadFunction;
   void* userData;
   const char* name;
public:
   TID tid;
```

There is an `operation=` ridefinition and a copy constructor; other two constructors create a new `NULL` object and a new object with a new `ThreadFunctionType`, a `userData` and a `name`.

The method `UserThreadFunction` execute the registered thread; `GetThreadName` gets the name of the thread and `ExceptionProtectedExecute` allows to call a subroutine within an exception handler protection, this implementation is a dummy one, so it must be recoded at higher levels (i.e. to be subclassed).

```
   void operator=(ThreadInitialisationInterface &tii);
   ThreadInitialisationInterface(ThreadInitialisationInterface& tii);

   ThreadInitialisationInterface();
   ThreadInitialisationInterface(ThreadFunctionType userThreadFunction,
      void* userData, const char* name);
   virtual ~ThreadInitialisationInterface();
```

```
virtual void UserThreadFunction();
virtual const char *GetThreadName();
bool ExceptionProtectedExecute(ThreadFunctionType userFunction,
   void *userData, ExceptionHandlerInterface *eh);
```

Threads DataBase

[ThreadsDatabase.h,ThreadsDatabase.cpp]
The *Threads DataBase* is not a class but a set of function. It is depicted in Figure 1.17 only for dependency purpouse. Such entity is a register of all the threads currently running (or expected to be running) in the system. All functions that lets you manage the *Threads DataBase* follows, functions can be founded in *level0/ThreadsDatabase.h*.

Threads DataBase (TDB) is managed via the class `Threads` the following three functions are only accessible by *level0/Threads.cpp* (i.e. are not exported or declared anywhere else). The method `TDB_NewEntry` creates new TDB entry associated to the `ThreadInitialisationInterface` object passed by pointer. Note that a `Threads` object is not a subclass of `ThreadInitialisationInterface`. The method `TDB_RemoveEntry` destroy the TDB entry associated wtih the current calling thread, note that the method has no arguments. The method `TDB_GetTII` access private thread information, on timeout returns `NULL`.

```
bool TDB_NewEntry(ThreadInitialisationInterface *tii);
ThreadInitialisationInterface* TDB_RemoveEntry();
ThreadInitialisationInterface* TDB_GetTII(TID tid=0);
```

Now comes some generic usable functions. Every time a user want to access the TDB it must request a lock to use it with `TDB_Lock` and unlock with `TDB_UnLock`.

The method `TDB_NumberOfThreads` returns the number of threads registered in the database. `TDB_GetThreadID` retrieves the TID of thread `n`, the method `TDB_Find` return the TID of thread named `name`. `TDB_GetInfo` retrieves information about a thread identified either by `name` or `TID` or index; `TDB_GetName` retrieves pointer to name in `ThreadInitialisationInterface` attribute for current thread.

```
bool TDB_Lock(TimeoutType tt = TTInfiniteWait);
bool TDB_UnLock();

int TDB_NumberOfThreads();
TID TDB_GetThreadID(int n);
TID TDB_Find(const char* name);
bool TDB_GetInfo(ThreadInitialisationInterface& tiiCopy,
   int n=-1,const char* name=NULL,TID tid=(TID)-1);
const char* TDB_GetName();
```

Differently with others *DataBase*'s structures this *Threads DataBase* doesn't relay on a linked list but on an array of pointers to `ThreadInitialisationInterface` objects.

Threads

[Threads.h, Threads.cpp]

This is the core threads class, it provides for thread registration in the *Threads Database* and provide a complete interface to thread managment (processors mask are managed via the `ProcessorType` class).

The method `SetThreadInitialisationInterfaceConstructor` sets the function used to build the thread initialisation interface. An `ThreadInitialisationInterface` object is created using either the default value or the parameter passed to the function by the `BeginThread` method. The

method `SetPriorityLevel` changes thread priority; applies only to current thread 0..31 (on windows it is actually /4). `EndThread` is called implicitly at the end of the main thread function, calling this method leaves some allocated memory unfreed. Calling `BeginThread` will start a thred this method will dynamically allocate an object of type `ThreadInitialisationInterface` using the function hook `ThreadInitialisationInterfaceConstructor`; this allows the programmer to choose which constructor has to be used in the case a `ThreadInitialisationInterface` derived class had been used.

In BaseLib that is a thread based library there are four classes of priority (from the maximum priority to the minimum priority):

- RealTime
- High
- Normal
- Low

`Threads` class define only static methods so each static method apply to the current executed thread. You can change the thread priority using `SetRealTimeClass`, `SetHighClass`, `SetNormalClass` and `SetLowClass`.

`ThreadId` method gets the current thread id; `Kill` kills a thread asynchronously; `IsAlive` checks whether thread `tid` is still alive. The method `GetName` basically retrieve thread's name.

```
public:
   static void SetThreadInitialisationInterfaceConstructor(ThreadInitialisationInterfaceConstructorT
   static void SetPriorityLevel(uint32 level);

   static void EndThread();
   static TID BeginThread(ThreadFunctionType function,
      void* parameters = NULL,
      uint32 stacksize=THREADS_DEFAULT_STACKSIZE,
      const char* name=NULL,
      ExceptionHandlerBehaviour exceptionHandlerBehaviour=XH_NotHandled,
      ProcessorType runOnCPUs=PTDefaultCPUs);

   static void SetNormalClass();
   static void SetRealTimeClass();
   static void SetIdleClass();
   static void SetHighClass();

   static TID ThreadId();
   static bool Kill(TID tid);
   static bool IsAlive(TID tid);
   static const char *GetName();
```

Processes

`[Processes.h]`
The class `Processes` give an interface to operating system wide functions like creating a process, i.e. loading an executable file, process index, killing a process and reboot the system. The description of functions interface follows.

The method `GetThisPid` returns the current's process id and `ProcessName` retrieves the name of this applicationm, it is obtained by parsing `argv[0]` it does not work if `REAL_MAIN` is defined.

Methods `Detach` and `Start` are exactly the same method for every system except OS/2. `Detach` method launchs a specified command as a newly detached process and return the task id of the new process or 0 if the process has not been created; `command` argument is a pointer to a string containing

the command to execute (i.e. the application that will run); `args` argument is a pointer to a string containing the arguments of the command; `path` is a pointer to a string containing the path to the command.

The method `Kill` lets the user kill the process specified by the `task_id`; `IsAlive` lets query the system to know if the process specified by argument is alive; last method: `Reboot` causes the machine to reboot.

```
public:
   static uint32 GetThisPid();
   static const char *ProcessName();

   static uint32 Detach(const char *command,const char *args,const char *path = NULL);
   static uint32 Start(const char *command,const char *args,const char *path=NULL);
   static bool Kill(uint32 task_id);
   static bool IsAlive(uint32 task_id);

   static void Reboot();
```

1.8.1 Design Notes

Class `Processes` is likely to be moved in directory *arch*. I'm not sure about the need of a `ThreadInitializationInt` a `ThreadInterface` can be better.

1.9 Mathematic

In this section there are some class involving mathematical conversion and complex number handling. Classes in this seciton are here listed and depicted in Figure 1.18.

- Complex
- NordFloat
- FastMath

Complex

`Complex.h`

Comples is an implementation of the complex numerical type as 2 doubles.

The structure of the class is straightforward: it relies on two `double` attributes: `real` that holds the real part of the complex number and `imaginary` that holds the imaginary part of the complex number. There are some contructors; the first one takes no arguments, the second and third take only one argument:the real part (`double` and `int`), fourth and fifth take two doubles as the real and imaginary parts, last constructor is the copy constructor.

There are two set methods `Real` that return the real part of the number and `Complex` that return the complex part.

```
private:
   double real;
   double imaginary;
public:
   inline Complex();
   inline Complex(double x);
   inline Complex(int x);
   inline Complex(double r,double i);
   inline Complex(double r[2]);
```

Figure 1.18: BaseLib Level0 math classes

```
inline Complex (const Complex &x);

inline double Real();
inline double Imaginary();
```

The class redefine all the mathematical operators using complex numbers. `operator` return the complex conjugate of the number.

```
inline Complex& operator=(const Complex &x);
inline Complex& operator=(double x);
inline Complex& operator=(int x);

inline friend const Complex operator~(const Complex &x);
inline friend const Complex operator-(const Complex &x);
inline friend const Complex operator+(const Complex &a,const Complex &b);

inline friend const Complex operator+(const Complex &a,double b);
inline friend const Complex operator+(double a,const Complex &b);

inline void operator+=(const Complex &a);
inline void operator+=(double a);

inline friend const Complex operator-(const Complex &a,const Complex &b);
inline friend const Complex operator-(double a,const Complex &b);
inline friend const Complex operator-(const Complex &a,double b);

inline void operator-=(const Complex &a);
```

```
   inline void operator-=(double a);

   inline friend const Complex operator*(const Complex &a,const Complex &b)
   inline friend const Complex operator*(const Complex &a,double b);
   inline friend const Complex operator*(double a,const Complex &b);

   inline void operator*=(const Complex &a);
   inline void operator*=(double a);

   inline friend const Complex operator/(const Complex &a,const Complex &b);
   inline friend const Complex operator/(const Complex &a,double b);
   inline friend const Complex operator/(double a,const Complex &b);

   inline void operator/=(const Complex &a);
   inline void operator/=(double a);
   inline bool operator< (const Complex &x);
   inline bool operator> (const Complex &x);
   inline bool operator== (const Complex &x);
   inline bool operator== (double x);
```

There are also a set of simple mathematical functions. `Norma2` returns the norma squared, `Norma` return the norma and `Arg` return the argument (?).

```
   inline friend const Complex sin(const Complex &x);
   inline friend const Complex cos(const Complex &x);
   inline friend const Complex exp(const Complex &x);
   inline friend const Complex log(const Complex &x);
   inline friend const Complex Clog(double x);
   inline friend const Complex sqrt(const Complex &x);
   inline friend const Complex Csqrt(double x);
   inline friend const Complex sqr(const Complex &x);

   inline double Norma2()const;
   inline double Norma()const;
   inline double Arg()const;
```

NordFloat

`NordFloat.h`
Class `NordFloat` converts a float between network byte order (*Nord*) and Intel byte order. There is only one attribute: `data` that is a `char[6]` array; there are two private methods that address the conversion between formats. There are constructors one zero constructor and one constructor that takes a `float`. Some operators are also defined.

```
private:
   char data[6];

   void ConvertFromIEEE(float x);
   float ConvertToIEEE();
public:
    NordFloat();
    NordFloat(float x);

    void operator=(NordFloat x);
    void operator=(float x);
    bool operator==(NordFloat x);
    bool operator==(float x);
    float operator()();
```

FastMath

`FastMath.h`
Source *FastMath.h* doesn't define a class but for illustrative needes it is represented like a class in Figure 1.18. The file contains one function definition and many mathematical constants. Function `FastFloat2Int` converts a float to an integer using processor instructions (will be moved in *arch/*.

```
static inline int32 FastFloat2Int(float input);
```

Constants follow in Table 1.4

e	`M_E`	2.7182818284590452354
$log2(e)$	`M_LOG2E`	1.4426950408889634074
$log10(e)$	`M_LOG10E`	0.43429448190325182765
$log(2)$	`M_LN2`	0.69314718055994530942
$log(10)$	`M_LN10`	2.30258509299404568402
pi	`M_PI`	3.14159265358979323846
$pi/2$	`M_PI_2`	1.57079632679489661923
$pi/4$	`M_PI_4`	0.78539816339744830962
$1/pi$	`M_1_PI`	0.31830988618379067154
$2/pi$	`M_2_PI`	0.63661977236758134308
$2/sqrt(pi)$	`M_2_SQRTPI`	1.12837916709551257390
$sqrt(2)$	`M_SQRT2`	1.41421356237309504880
$sqrt(1/2)$	`M_SQRT1_2`	0.70710678118654752440

Table 1.4: constants in `FastMath`

1.9.1 Design Notes

The aim of class `NordFloat` is not really clear; probably the conversion is not between float in differents byte format. The design is ok also if there are no links between classes. Probably the class `FastMath` must moved to arch because is architecture dependent. See as an example

`http://www.dmh2000.com/cpp/dswap.shtml`

1.10 Design Notes

In BaseLib the support to different operating systems as well as the support of different architectures is done by using the pragma def statement, this is today considered as a poor designing choice, difficult to mantain, not easy to scale and also not easy to understand. A better solution would be to adopt the same scheme adopted in the Linux kernel directory *arch/*.

Chapter 2

BaseLib Level 1

BaseLib Level1 group togheter differents BaseLib data structures that are vital for each application developed on the library. The first component this chapter deal with is the *Object Registry Database* (ORDB) a data structure that let the user query about the run time type identification of an object (something similar with the C++'s RTTI infrastructure addressed below). Another really usefull infrastructure is the *Global Object Database* (GODB) a database structure that holds a reference to each instantiated class and also let it be garbage collectable. An *Error System Instruction* is partially introduced as a linked list stucture. In Level1 you are going to see for the first time the foundamental *Configuration Database*, a tree data structure that holds the most important information for the loading and configuration of the system. In such level it also possible to find the file `StreamAttributes.h` that defines some attributes that concern streams, such file will be not addressed in the following.

Classes in BaseLib Level1 can be grouped with the following criteria:

- Object Registry DataBase (ORDB)
- Global Object DataBase (GODB)
- Error System Instruction (ESI)
- Configuration DataBase (CDB)
- Stream Attributes

Such groups cover all BaseLib Level 1 classes, we start analysing the first group.

2.1 Object Registry DataBase

Before starting to introduce the Object Registry DataBase structure we spend some words trying to create a general basic knowledge about Object Oriented Languages and Run Time Type Identification. RTTI, i.e. Run-Time Type Information, or Run-Time Type Identification, refers to a C++ system that keeps information about an object's data type in memory at runtime. Run-time type information can apply to simple data types, such as integers and characters, or to generic objects. This is a C++ implementation of a more generic concept called reflection.

2.1.1 RTTI

Run Time Type Identification provides some information about objects at run time such as the name of its type. The C++ language has RTTI support, which fulfills the minimal requirements, but it is not enough for many applications of RTTI, such as object persistency. Other languages (like Java and C#) have better RTTI system making possible to declare properties for accessing the objects and the language implements also persistency (serialization in Java), but these languages have other disadvantages.

C++ programs may also need persistency and the advanced features of RTTI systems. The C++ language is so powerful that it is possible to implement properties and an advanced RTTI system.

The standard C++ provides the `typeid()` operator for getting type information. Its argument is an expression (a reference or a pointer of an object) or a type name. It returns a constant reference to a `type_info` object containing some information of the object's type.

The `type_info` class has only a few member functions:

```
class type_info {
public:
  virtual ~type_info();
  bool operator== (const type_info& rhs) const;
  bool operator!= (const type_info& rhs) const;
  bool before (const type_info& rhs) const;
  const char* name() const;
private:
  type_info (const type_info& rhs);
  type_info& operator= (const type_info& rhs);
};
```

The method `name` gets the string representation of the type (e.g. `int`, or `MyClass`); `before` is used for ordering; `operator==()` and `operator!=()` for comparing `type_info` objects; this information does not help to find the relations of objects and does not solve most of the problems that arise in applications. It is not designed for that. All applications have different requirements and the same standard type description cannot fulfill all the requirements.

However the `type_info` class can be used as a key in a map storing more detailed type information [Strousstrup 15.4.4]. This way every application can define and use its own RTTI system, which is a very flexible solution. The problem is, that someone has to define the structure of the RTTI record and fill the map with records for every type. This job is not trivial and some code has to be written for every application for doing that.

An alternative to this problem could be to write a piece of code able to walk the RTTI structure builded by the C++ compiler. RTTI data structures are necessary at run time for the *dynamic cast* process; but what happens is that every compiler, developed from different companies, has its own data structures that differ from product to product. Using such strategy it is possible to exploit compile time produced data structures without creating your own data structures, i.e. having two parallels RTTI tables.

2.1.2 ORDB

BaseLib addresses the problem of knowing every class in the system holding an ORDB that is a simple linked list of all class types loaded in the system. There is only one instance of an ORDB (of type `ObjectRegistryDataBase`) at runtime and is called `ObjectRegistryDataBaseInstance` (see file *level1/ObjectRegistryDataBase.cpp*).

The ORDB doesn't hold the hierarchy relationship between classes but it's only a list of all class types registered in the system, this list can be searched for a class by name or by an id number and it is possible to retrieve the name address of the class and other usefull informations. The whole structure of all classes that makes the ORDB work is depicted in Figure 2.1 that follows.

Classess that belong to this group are:

- ClassStructureEntry
- ClassStructure

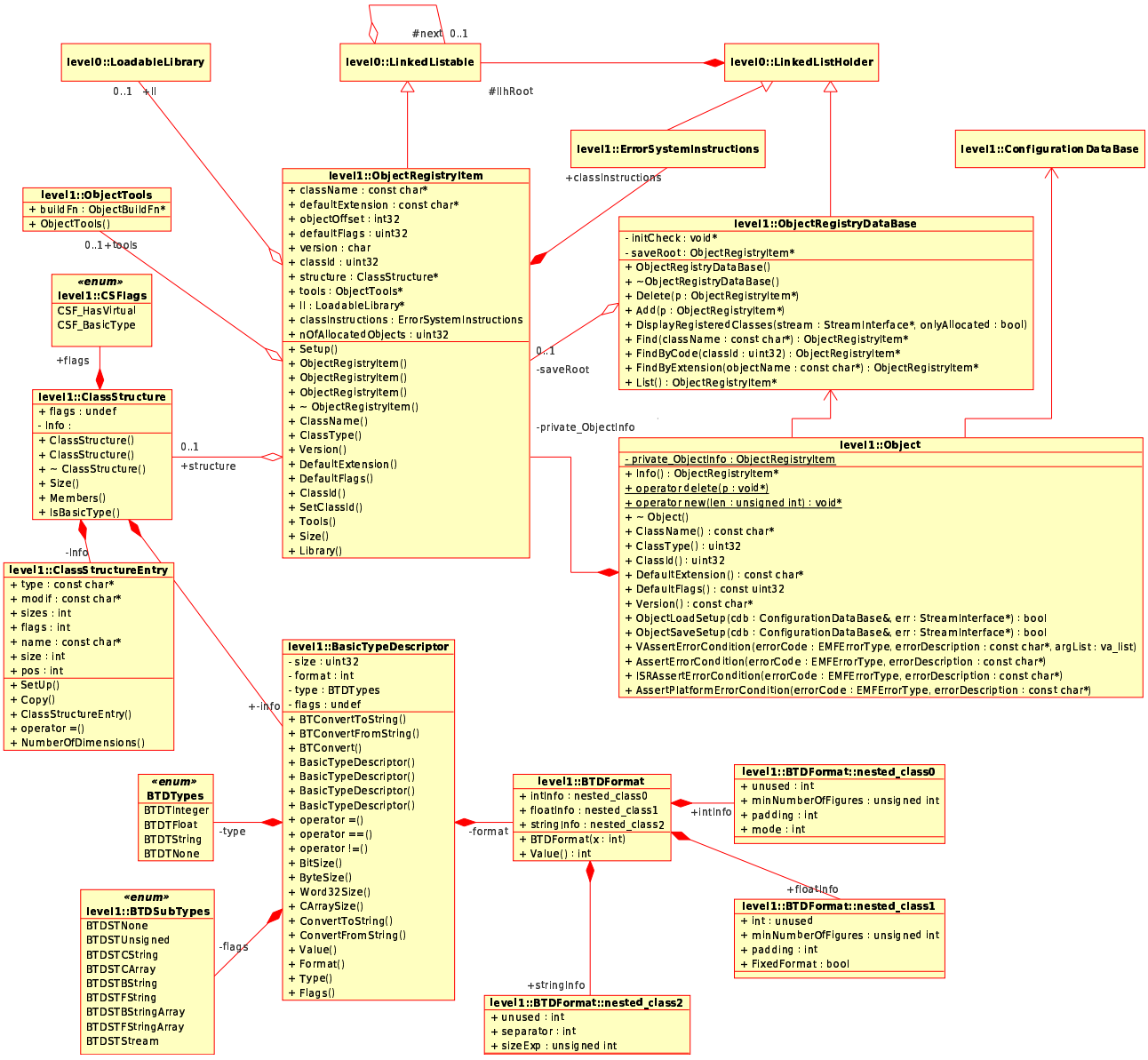

Figure 2.1: BaseLib Level1 Object Registry Database (ORDB) classes

- BasicTypeDescriptor
- BTDFormat
- ObjectRegistryItem, ObjectTools
- ObjectRegistryDataBase
- Object

All classes listed above estabilsh the *Object Registry Database* structure. The basic idea behind this work is to imitate what is developed in the Java language: every object inherit from the `Object` class providing a common ancestor of every object that force to have some common methods (like `ToString` in Java). In BaseLib we want to achieve the same common ancestor letting all the objects inheriting common methods and common functionalities like the most important one: be part of the ORDB. Such ORDB can be easily walked by any other object providing debbugging functionalities

and runtime linking and loading.

We first go throught the implementation of the `ObjectRegistryDataBase` and then we show how the class `Object` can take advantage of it and which basic methods add to any other objects that inherith from that. We start with some utility classes.

BTDFormat

`[BasicTypes.h]`
Class **BTDFormat** is declared as a `union` with a constructor and a single method `Value`. Attribute can be a `struct intInfo`, `struct floatInfo` or `struct stringInfo`. Each struct, that follows, try to describe the format of a single basic type that can be *integer*, *float* or *string*. The format of the `BTDFormat` depends on the endianity so that to fit in the 14 bit space in the **BasicTypeDescriptor** attribute's `format` analized in the next section.

```
union BTDFormat {
   struct {
      unsigned int minNumberOfFigures:5;
      int padding:3;
      int mode:3;
      int unused: 21;
   } intInfo;
   struct {
      unsigned int minNumberOfFigures:5;
      int padding:3;
      bool fixedFormat:1;
      int unused: 23;
   }floatInfo;
   struct {
      int separator:3;
      unsigned int sizeExp:4;
      int unused: 25;
   } stringInfo;

   BTDFormat(int x=0);
   int Value();
};
```

`intInfo` is used in single strings and streams, `floatInfo` is used in floats to determine how many meaningful figures are there and `stringInfo` is used in strings of C ARRAYs and BSTRING type.

BasicTypeDescriptor

`[BasicTypes.h, BasicTypes.cpp]`

Support for basic types of built in language types. Each basic type is described in BaseLib with a `BasicTypeDescriptor`, such class has the following attributes that account about the size of the object, attribute `size`, maximum size is 1024 units (10 bit), the format and type.

```
private:
   uint32 size:10;
   int format:14;
   BTDTypes type:4;
   BTDSubTypes flags:4;
```

Actually a `BasicTypeDescriptor` describes a basic type with a `type` and a `subtype`, so we have a two level description. The first level (`enum BTDTypes`) discriminate about:

- `BTDTInteger`, an integer;
- `BTDFloat`, standard float;

- `BTDString`, any type of string;
- `BTDTNone` that's denote not a `BTDType`;

For some of that specifiers there is a more precise description thanks to the `enum BTDSubType`, note that in the source code there is no `enum BTDSubType` but can be a good idea to make it. The `BTDSubType flag` define the meaning of the `size` attribute.

- `BTDSTNone` no flags;
- `BTDSTUnsigned` modifier for the integer;
- `BTDSTCString` the `void*` is casted to `char**`; it's assumed to be a vector of pointers of `size` matching that of the source, when used as destination the existing pointer is first freed and then replaced with a new malloced one;
- `BTDSTCArray char[]`, `size` field is the array size, the string is always 0 terminated;
- `BTDSTBString BString` class, `size` field is meaningless `void*` is `BString*` it is a pointer to a single `BString` the parts will be separarted using the character specified in the format field;
- `BTDSTFString FString` class, `size` field is meaningless `void*` is `FString*` it is a pointer to a single `FString` the parts will be separated using the character specified in the format field;
- `BTDSTBStringArray BString` class, `size` field is meaningless `void*` is `BString*` it is an array of `BString`s matching the input size;
- `BTDSTFStringArray FString` class, `size` field is meaningless `void*` is `FString*` it is an array of `FString`s matching the input size;
- `BTDSTStream StreamInterface` class, `size` field is meaningless.

A complete picture of that hierarchy is showed in Figure 2.2. It is really important to note the relationship between those types.

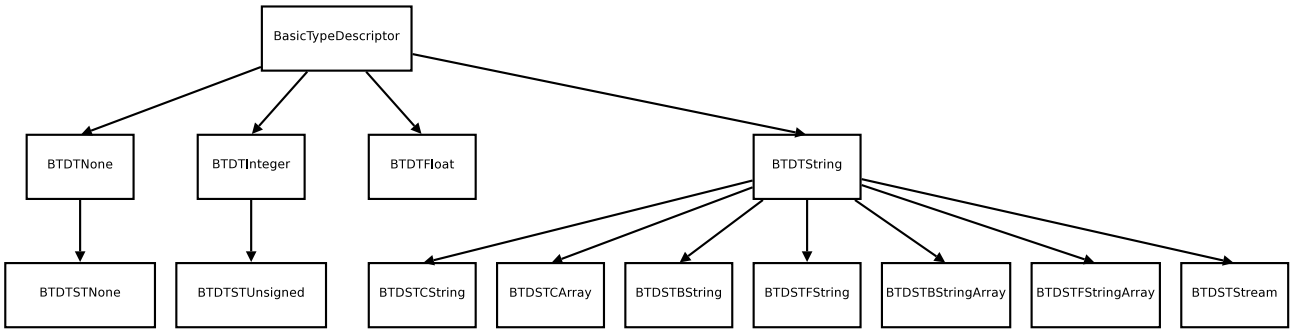

Figure 2.2: BaseLib Level1 BasicTypes hierarchy

Now we take a look to the methods exported from such class, there are four constructors, the first one create a `BasicTypeDescriptor` with `size` 32 bit, `type BTDTInteger`, `flags BTDSTNone` and `format NULL`, so it makes a `BasicTypeDescriptor` of an *int32*. The second constructor helps making a new object using another one via the method `Value`. The third constructor is the most used in the library it let you construct your own basic type by setting each attribute in the class. The comes the copy constructor.

Then comes some getter methods that returns the size of the basic type counting the bits, bytes abd words (`BitSize`, `ByteSize` and `Word32Size` that is the minimum number of words to hold it);

`CArraySize` is the size of a `CArray`. The method `Value` convert a `BasicTypeDescriptor` in an integer, `Format` return the `format` attribute, `Type` return the `type` attribute and `Flags` the `flags` attribute.

Methods `ConvertToString` and `ConvwertFromString` convert a string like "uint" in a `BasicTypeDescriptor` of type `BTDTInteger` and subtype `BTDSTUnsigned` and back.

```
public:
   BasicTypeDescriptor();
   BasicTypeDescriptor(int32 equivalent);
   BasicTypeDescriptor(uint32 size,BTDTypes type, BTDSubTypes flags,BTDFormat format = BTDFNone);
   BasicTypeDescriptor(const BasicTypeDescriptor& desc);
   BasicTypeDescriptor operator=(const BasicTypeDescriptor &desc);

   bool operator==(const BasicTypeDescriptor &desc) const;
   bool operator!=(const BasicTypeDescriptor &desc) const;

   int32 BitSize() const;
   int32 ByteSize() const;
   int32 Word32Size() const;
   uint32 CArraySize() const;

   int32 Value() const;
   BTDFormat Format() const;
   BTDTypes Type() const;
   BTDSubTypes Flags() const;

   const char* ConvertToString(BString& string) const;
   bool ConvertFromString(const char* name);
```

In the file *level1/BasicTypes.h* are declared the following basic type descriptors, with the following parameters.

type	BaseLib type	bit size	`BTDTypes`	`BTDSubTypes`
8 bit signed integer	`BTDInt8`	8	BTDTInteger	BTDSTNone
16 bit signed integer	`BTDInt16`	16	BTDTInteger	BTDSTNone
32 bit signed integer	`BTDInt32`	32	BTDTInteger	BTDSTNone
64 bit signed integer	`BTDInt64`	64	BTDTInteger	BTDSTNone
8 bit unsigned integer	`BTDUint8`	8	BTDTInteger	BTDSTUnsigned
16 bit unsigned integer	`BTDUint16`	16	BTDTInteger	BTDSTUnsigned
32 bit unsigned integer	`BTDUint32`	32	BTDTInteger	BTDSTUnsigned
64 bit unsigned integer	`BTDUint64`	64	BTDTInteger	BTDSTUnsigned
32 bit float	`BTDFloat`	32	BTDTFloat	BTDSTNone
64 bit float	`BTDDouble`	64	BTDTFloat	BTDSTNone
char* string	`BTDCString`	0	BTDTString	BTDSTCString
`BString` pointer	`BTDBString`	0	BTDTString	BTDSTBString
`BString` for each element	`BTDBStringArray`	0	BTDTString	BTDSTBStringArray
`StreamInterface*`	`BTDStream`	0	BTDTString	BTDSTStream

Table 2.1: `BasicTypeDescriptor`s defined in BaseLib

All such types are declared `static const` and they are placed in a header file. This file is directly included in BaseLib from three **.cpp* sources and within a first indirection by other six files, so in two levels nine times, we doesn't try to recover all times that types will be recompiled in the whole BaseLib but within one level the same classes are recompiled 9 times: its a waste of memory allocate the same object privately for each compiled object.

ClassStructureEntry

[ClassStructureEntry.h]
This class describes an entry within a class structure. Each class `ClassStructureEntry` describes an attribute of the class is defining about. So each entry has a `type`, modifiers in attribute `modif` that usually are "*", i.e. the dereference and the reference (or address of) operator. The attribute `sizes` holds four integers, `flags` some flags, `name` is the name of the attribute we are asking for registration and `pos` is the offset between the class (extracted using `indexof`).

```
public:
   const char* type;
   const char* modif;
   int sizes[CSE_MAXSIZE];
   int flags;
   const char* name;
   int size;
   int pos;
```

The class has a `SetUp` method that is a helper constructor methods; it helps the constructor setting class's attributes, a `Copy` method that call the previous method and at the end the method `NumberOfDimensions` that count in case of an array the number of dimension of the arrays.

```
   void SetUp(const char* type,const char* modif,
      int size0,int size1,int size2,int size3,
      int flags,const char* name,
      int size,int pos);
   void Copy(ClassStructureEntry& x);

   ClassStructureEntry(const char* type="",const char* modif="",
      int size0=0,int size1=0,int size2=0,int size3=0,
      int flags=0,const char* name="",
      int size=0,int pos=0);
   ClassStructureEntry& operator=(ClassStructureEntry& x);

   int NumberOfDimensions();
```

ClassStructure

[ClassStructure.h]
The class `ClassStructure` holds information about a basic type or about a class. Class distinction is made on the attribute `flags` that is of enumeration type `CSFlags`.

The most important attribute of this class is the `union`; in the union it is possible to store or informations about one basic type (indexed by an `int32`) otherwise a set of `ClassStructureEntrys`. In the first case the class describe a basic type and in the last case a complete class defined by the user. In the `struct csInfo` the field `size` is the total size in bytes and `members` is a `NULL` terminated list of members.

```
public:
   CSFlags flags;
   union {
      struct {
         int32 btd;
      }btInfo;
      struct {
         int size;
         ClassStructureEntry** members;
      }csInfo;
   };
```

The first constructor builds up a `ClassStructure` object that holds a basic type, and the second builds up a class's description object of a class.

The method `size` return the size in bytes of the class, `Members` return the `csInfo.members` attribute and `IsBasicType` queries about the type of the stored object.

```
public:
   ClassStructure(const char* name,BasicTypeDescriptor btd);
   ClassStructure(const char* name,int size,CSFlags flags,ClassStructureEntry** members);
   ~ClassStructure();

   int32 Size() const;
   ClassStructureEntry** Members() const;
   bool IsBasicType(BasicTypeDescriptor &btd);
```

ObjectRegistryItem, ObjectTool

`[ObjectRegistryItem.h, ObjectRegistryItem.cpp]`

The first thing the file *ObjectRegistryItem.h* defines is the type `ObjectBuildFn` that is a simple function type that return an `Object`.

```
typedef Object *(ObjectBuildFn)();
```

This `typedef` is used in the first class defined in the header file that is `ObjectTools`. Such class has only one attribute, public, of type `ObjectBuildFn`, the constructor let you simply set this attribute. Someone can argue about the necessity of having a class that hold a single attribute. It is not necessary infact.

```
class ObjectTools{
public:
   ObjectBuildFn* buildFn;
   ObjectTools(ObjectBuildFn* buildFn){
      this->buildFn = buildFn;
   }
};
```

Let's now spend some time on the `ObjectRegistryItem` class. This class stores information about a class type. An `ObjectRegistryItem` save the name of the class in `className` attribute, the extension used as default in `defaultExtension`; attribute `objectOffset` is the relative position of `Object` within the class; `defaultFlags` holds some default attributes for the class; `version` holds the version string of this class. The attribute `classId` is an user defineable class identification, the default one is calculated as a checksum of the name based on $sum(x)sum(x^2)sum(x^3)sum(x^4)$, `structure` holds information about the structure of this class; `tools` is the function to create this class; `ll` attribute maintain information about the loadable library where the class's object code resides; `classInstructions` collect all information about the behaviour on error situations. The attribute `nOfAllocatedObjects` count runtime, for each class type, how many objects of that type are instantiated in the system.

```
public:
   const char* className;
   const char* defaultExtension;
   int32 objectOffset;
   uint32 defaultFlags;
   char version[8];
   uint32 classId;
   ClassStructure* structure;
   ObjectTools* tools;
   LoadableLibrary* ll;
   ErrorSystemInstructions classInstructions;
   uint32 nOfAllocatedObjects;
```

Then follow a set of getter methods, there is a setter method only for the `classId` attribute. The method `ClassType` returns the type of the class, this identification code is unique within an application, in the actual implementation it returns value of the char pointer `className`.

```
const char* ClassName();
uint32 ClassType();
const char* DefaultExtension();

const uint32 DefaultFlags();
const char* Version();

uint32 ClassId();
void SetClassId(uint32 id);
ObjectTools* Tools();
int Size();

LoadableLibrary* Library();
```

The method `Setup` is an helper method for the constructor, argument `className` is the name of the class, `version` is the version of the class as a string (see more on next sections). Constructors initialise the structure details of a *Object Registy DataBase* record using the structure information, adding this `ObjectRegistryItem` to the database after calling the main constructor if a record of class `className` is not found in the ORDB.

```
void Setup(const char* className,
           const char* version,
           const char* defaultExtension = NULL,
           uint32 defaultFlags = 0,
           ObjectTools* tools = NULL);

ObjectRegistryItem();
ObjectRegistryItem(const char* className,
                   const char* version,
                   int objectOffset,
                   const char* defaultExtension = NULL,
                   uint32 defaultFlags = 0,
                   ObjectTools* tools = NULL);
ObjectRegistryItem(const char* className, ClassStructure* structure);

~ObjectRegistryItem();
```

ObjectRegistryDataBase

`[ObjectRegistryDataBase.h, ObjectRegistryDataBase.cpp]`
The *Object Registry Database* is the system-wide class database that take notes of every class type registered in your application; for each class type, we saw on previous section about `ObjectRegistryItem` we have a usage count.

The `ObjectRegistryDataBase` can be used to save and load classes creating recogniseable objects. To perform such activity the `ObjectRegistryDataBase` is of type `LinkedListHolder` and holds `ObjectRegistryItem`s. Extending the *Linked List* concept it inherits the searching and filtering capability.

There are just two attributes, a `void*` called `initCheck` that points to itself, if not, the object is not initialized; there is also a `ObjectRegistryItem*` that keeps a copy of the pointer to the list. Note that someone can argue that this attribute is not really necessary because `ObjectRegistryDataBase` is a subclass of `LinkedListHolder` that holds an attribute of type `LinkedListable` and `ObjectRegistryItem` is a sublcass of such class. So it is simply a matter of casting the `ObjectRegistryDataBase::llhRoot` element to a `ObjectRegistryItem`.

```
   void* initCheck;
   ObjectRegistryItem* saveRoot;
```

The constructor during initialization wipes the list since this object could be initialized later than it was used, a trick had to be used, if the list has been already initialized then recover the lost list. All the methods are aimed at allowing insertion to the list even if it has not been yet initialized, the top of the list is saved continuously. Methods `Delete` and `Add` lets you delete and add an `ObjectRegistryItem` to the database.

The method `Find` search a class using the `className` passed by argument in the loaded loadable libraries in the system. First it checks against the syntax of the request and than scan to search the library. Other find methods search a class by an identification or by the full object file name or just its extension. The method `List` return back the complete list of `ObjectRegistryItem` objects.

```
public:
   ObjectRegistryDataBase();
    ~ObjectRegistryDataBase();

   void Delete(ObjectRegistryItem* p);
   void Add(ObjectRegistryItem* p);

   void DisplayRegisteredClasses(StreamInterface* stream,bool onlyAllocated);

   ObjectRegistryItem* Find(const char* className);
   ObjectRegistryItem* FindByCode(uint32 classId);
   ObjectRegistryItem* FindByExtension(const char* objectName);

   ObjectRegistryItem* List();
```

There is only one instance at runtime of the `ObjectRegistryDataBase` class in the system and you can look at that in the file *level1/ObjectRegistryDataBase.cpp* and the name is `ObjectRegistryDataBaseInstance`.

Object

`[Object.h, Object.cpp, ObjectMacros.h]`
This is the standard base class providing a type information system; it needs that each subclass provides a `className` buffer and overrides both methods `Name()` and `Type()`. Every class that inherits from `Object` must have the following methods:

```
public:
   virtual ~Object()

   const char* ClassName() const;
   uint32 ClassType();
   uint32 ClassId();
   const char* DefaultExtension();
   const uint32 DefaultFlags();
   const char* Version();
```

Those methods lets the ORDB work infact they supply all the type information the system need. For each class objects the class's name is returned with `ClassName`, the type is returned with `ClassType`, an identification is returned using `ClassId` extension, flags and version using `DefaultExtension`, `DefaultFlags` and `Version`. All that informations are not stored inside the `Object` class, as you can see in Figure 2.1 but in an `ObjectRegistryItem` attribute associated with the object. Such attribute `private_ObjectInfo` is not declared as a static attribute in the class but it's a static global object declared in the `*.cpp` code using the macros that follow.

Next two paragraphs address two BaseLib's objects from this level that are treatened in next sections, those objects are really important for mechanisms involved in this section.

Configuration DataBase A *Configuration DataBase* is a sort of data structure that is the source and also the sink of configuration parameters of the library. The method `ObjectLoadSetup` is the standard `Object` creation function. Uses a CDB to pass the initialisation parameters. The CDB information is read from the subtree that is currently addressed. The method `ObjectSaveSetup` is the standard `Object` save function. Uses a CDB to save the parameters.

It works like this: a CDB holds all informations about classes that must be built at runtime, you load a CDB and then reading the CDB BaseLib instantiate the objects the CDB is listing for. Each `Object` subclass must also overload last two methods, in this way each subclass can read and save its own parameters in such methods.

Error Management There also some methods for error management. The method `VAssertErrorCondition` sets the error status and depending on setup does appropriate action, this call is to be called from static members; the method `AssertErrorCondition` sets the error status and depending on setup does appropriate action, such call is to be called from static members. The method `ISRAssertErrorCondition` sets the error status and depending on setup does appropriate action, this call is to be called from interrupts. `AssertPlatformErrorCondition` sets the error status and depending on setup does appropriate action.

```
virtual bool ObjectLoadSetup(ConfigurationDataBase& cdb,StreamInterface* err);
virtual bool ObjectSaveSetup(ConfigurationDataBase& cdb,StreamInterface* err);

void VAssertErrorCondition(EMFErrorType errorCode,
                           const char* errorDescription,
                           va_list argList);
void AssertErrorCondition(EMFErrorType errorCode,
                           const char* errorDescription=NULL,...) const;
void ISRAssertErrorCondition(EMFErrorType errorCode,
                           const char* errorDescription=NULL,...);
void AssertPlatformErrorCondition(EMFErrorType errorCode,
                           const char* errorDescription=NULL,...);
```

Object Macros If you compare the previous listing of the class's methods with the class interface in Figure 2.1 it is possible to note that in our last listing there is no class's attribute and there are less methods then in the UML.

This happens because the ORDB relay on some code macros to work. All macros used throught BaseLib, concerning the ORDB are from the file *level1/ObjectMacros.h*. Now we spend some words about those macros that must be used in the development of new pieces of code. We start providing an example of how that macros are used in the `Object` class just analized. In `Object`'s header file the following macros are used outside the class declaration, the first one, and the second one inside it.

```
OBJECT_DLL(Object)
OBJECT_DLL_STUFF(Object)
```

In the source file (`*.cpp`) the following macros is used:

```
OBJECTREGISTER(Object,"$Id:␣level1.tex,v␣1.11␣2009/10/09␣18:00:50␣abarb␣Exp␣$")
```

Figure 2.3 depict show macro placement. We now expand it to understand better what they create improving the understanding of the library behaviour.

The first two macros presented expand to the following code. That code redefine the class `new` and `delete` operators, provide a `Info` methods that return a `ObjectRegistryItem*` about that inform about the class type.

```
extern "C" {
   ObjectRegistryItem* Get_private_ObjectInfo();
}
```

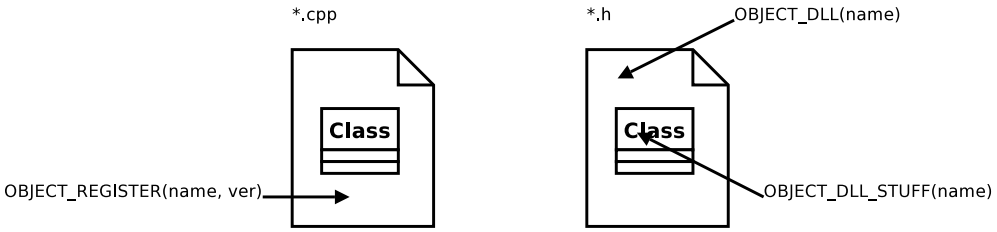

Figure 2.3: BaseLib Level1 Object Macros position

```
class Object {
public:
   virtual ObjectRegistryItem* Info() const {
      return Get_private_ObjectInfo();
   }
   static void operator delete(void* p) {
      OBJDeleteFun(p,Get_private_ObjectInfo());
   }
   static void* operator new(unsigned int len) {
      return OBJNewFun(len,Get_private_ObjectInfo());
   }
   friend Object* ObjectBuildFn__ ();
```

Last macro listed expand to the following code that it is possible to find in the `*.cpp` file and is statically compiled one time. The first three row statically declare an `ObjectRegistryItem` that will be registered in the ORDB providing type information with the version information.

```
   static ObjectRegistryItem _private_ObjectInfo(Object,
      "$Id:␣level1.tex,v␣1.11␣2009/10/09␣18:00:50␣abarb␣Exp␣$",
      ObjectClassOffset(Object));
   ObjectRegistryItem *Get_private_ObjectInfo(){
      return &_private_ObjectInfo;
   }
```

There are also other macros in the *level1/ObjectMacros.h* and are listed below. Comments are from the source code.

- `OBJECT_DLL(name)` To allow registering a class contained in a DLL. Use before the class declaration outside the class
- `OBJECT_DLL_STUFF(name)` To allow registering a class contained in a DLL. Use inside the class remember to set the public/private/protected afterwards.
- `OBJECT_STUFF(name)` To allow registering a class. Use inside the class. remember to set the public/private/protected afterwards.
- `OBJECTREGISTER(name,ver)` Register a class with its version. Use in the `*.cpp` file
- `OBJECTLOADREGISTER(name,ver)` Register a class with its version. Use in the `*.cpp` file. Automatically creates the Build function.
- `OBJECTLOADREGISTERFLAGS(name,ext,flags,ver)` Register a class with its version. Use in the `*.cpp` file. Automatically creates the Build function. Sets the flags an the file extension.
- `STRUCTREGISTER(name,structure)` Register a structure with its fields (structure). Use in the `*.cpp` file.
- `BASICTYPEREGISTER(name,type)` Register a basic type (int float ...). Use in the `*.cpp` file.

- `BASICTYPEREGISTER2(mod,name,type)` Register a basic type with a modifier in the name (short int long a = t ...). Use in the `*.cpp` file.
- `BASICTYPEREGISTER3(mod,mod2,name,type )` Register a basic type with 2 modifiers in the name (short int long b=t ...). Use in the .cpp file.

2.1.3 Remarks

TODO
TODO
TODO
Spiegare come sfruttare la reflection per creare una classe creare una classe via nome.

2.1.4 Design Notes

This section highlighted the basic RTTI-like functionality that BaseLib provide to the user. The ORDB presented here is a simply linked list with searching functionality. The design was made without thinking about preformance but focusing on the usability. The ORDB is really like a database, every loaded component of BaseLib as soon is loaded in the system register all class type it has and then, accessing the ORDB one can find the class it want and through the CDB loading it with the help of the ORDB.

At this level there is no hierical information between classes stored. So for example there is no method that tell you if a `ObjectRegistryDataBase` is a `LinkedListHolder` or not. This hierical relationship must be showed from the ORDB this can help also for the garbage collection (is it implemented in the GODB?).

ObjectTool is not necessary as a class: it has only one attribute can be deleted or improved.

The idea of having basic types and structured types is a good choice. Union's are probebly not the best choice because are rarely used in Object Oriented languages. The best design choice is to create an abstract type and subclassing it by basic types and by structured object: this is OOP (Object Oriented Programming).

2.2 Global Object DataBase

Such section address the *Global Object Database* that mantain a list of all instantiated objects that registrates with them. A garbage collector mechanisms is implemented, each object that inherits from some other that is GarbageCollectable can be handled by the collector and make your work memory safe; every object that will be registered will be automatically deleted by the library, like in Java. This helps the programmer providing a way to not waste memory.

Objects of the class `GCNamedObject` inherits from `Object` but are also `GarbageCollectable`, so they are from a registered type in the ORDB but also can be collected for garbage reclaiming and within the `GCNamedObject` class they also have a name that is used to store different or the same object types in the GlobalObjectDataBase (GODB); in this way you can instantiate object by name and also reclaiming object.

What is the difference between an `Object` and an `GCNamedObject`? An `Object` registers in the ORDB a type of data, i.e. a class structure, a `GCNamedObject` instead register an instance of an object by name (or handle, or id). A `GCNamedObject` is also `GarbageCollectable` but an `Object` is not.

Classes in this section are depicted in the UML diagram of Figure 2.4 and are listed below:

- GarbageCollectable

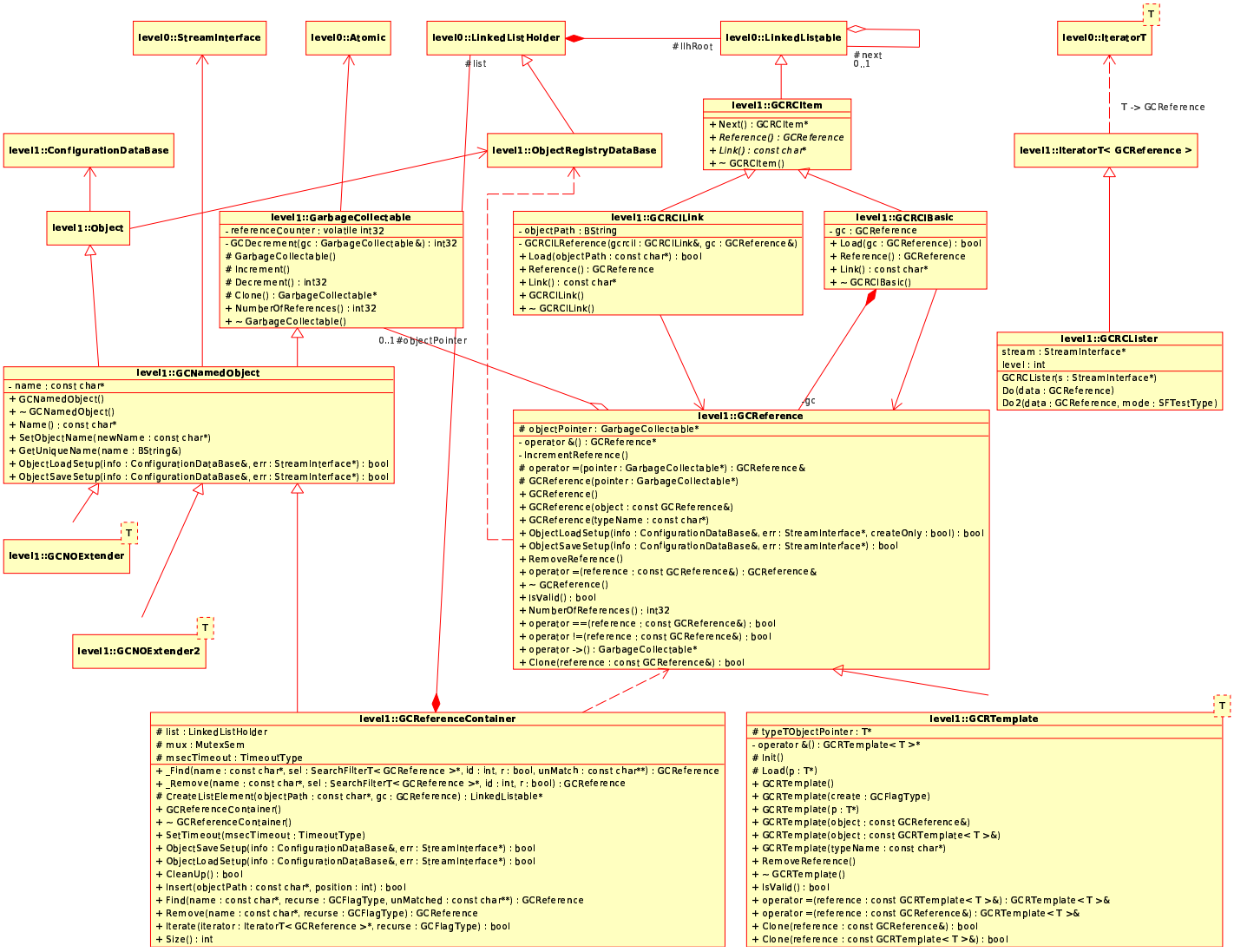

Figure 2.4: BaseLib Level1 Global Object Database (GODB) classes

- GCNamedObject
- GCNOExtender, GCNOExtender2
- GCRCItem, GCRCILink, GCRCIBasic
- GCRCLister
- GCReference
- GCRTemplate
- GCReferenceContainer
- GlobalObjectDataBase

GarbageCollectable

[GarbageCollectable.h, GarbageCollectable.cpp]
A class interface for implementing classes whose number of references is always accounted for; inherit from this class to implement garbage collection on a class. It accounts for reference to a class. It is really a simple class it provides only reference counting.

There is only one attribute `referenceCounter` that let supply all the functionality we need; the constructor initialises `referenceCounter` to zero, the method `Increment` is used to atomically increment the reference counter and the method `Decrement` is used to atomically decrement the reference counter.

To allow cloning of objects using references the final class must implement the method `Clone`. The method `NumberOfReferences` gets the number of references. The destructor is only called when the object is actually destroyed. The code does nothing here!

```
private:
   volatile int32 referenceCounter;
protected:
   GarbageCollectable();

   void Increment();
   int32 Decrement();

   virtual GarbageCollectable* Clone() const;
public:
   int32 NumberOfReferences() const;
   virtual ~GarbageCollectable();
```

GCNamedObject

[GCNamedObject.h, GCNamedObject.cpp]
A `GCNamedObject` is an extension of the `Object` class to include garbage collection and naming. It is used later to store object in the `GlobalObjectDataBase`.

A `GCNamedObject` is an `Object`, so it's class type will be registered, and it is also garbage collectable, so also it's usage is supervised and if it is not used jet it will be eliminated.

There is only one attribute `name` that adds to the `Object` a name string that will be necessary to identify it and so it must always contains a valid pointer to allocated memory. The constructor initialise to no name and the destructor deallocates memory. The method `Name` accesses the name, `SetObjectName` sets the name and `GetUniqueName` returns a name that contains also the address, it begins with *(address)name*.

The method `ObjectLoadSetup` initialises the name of the object using a CDB object, if `Name` is found than the name is taken from there, if missing then the `nodeName` is used, if `nodeName` start with `+` then the `+` is removed (after reading the stuff about CDB you will better understand those comments). `ObjectSaveSetup` saves the settings in the CDB passed by argument.

```
private:
   const char* name;
public:
   GCNamedObject();
   virtual ~GCNamedObject();

   const char* Name()const;
   void SetObjectName(const char* newName);
   void GetUniqueName(BString& name);

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,
      StreamInterface* err);
```

```
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,
      StreamInterface* err);
```

GCNOExtender, GCNOExtender2

[GCNOExtender.h]
The following templates creates classes that are joint `GCNamedObject` and user defined class, used to add `GarbageCollection` and naming to existing classes. The first template is really simple: it creates a new class that inherits from `GCNamedObject` and the templatized class without adding any method.

```
template <class T>
class GCNOExtender: public GCNamedObject, public T
{
public:

};
```

The second template make the new class extending `GCNamedObject` and the templatized class as before but adds a redefinition of the methods `delete` and `new`.

```
template <class T>
class GCNOExtender2: public GCNamedObject, public T
{
public:
   static void operator delete(void* p){
      T::operator delete(p);
   }
   static void* operator new (unsigned int len){
      return T::operator new(len);
   }
};
```

Those templates are rarely used in the whole BaseLib (in BaseLib version 2 they are used only 2 times, one time in *level1* and one time in *level5*), probably are to be considered new stuff.

GCRCItem, GCRCIBasic, GCRCILink

[GCRCItem.h]
Classes that follow are all subclasses of the class `LinkedListable`. Such elements are used to store references in the references containers.

GCRCItem The class `GCRCItem` is not an interface, because has the `Next` method implemented, like an interface it also has no attributes. It is an abstract class for container of references. It requires to implements only a few simple methods that will complete the `LinkedListable` interface; `Next` gets next element skipping no `GCRCItem` derivatives; it has two virtual methods: `Reference` and `Link` that give you access to the object reference and to the link.

```
public:
   GCRCItem* Next();
   virtual GCReference Reference() = 0;
   virtual const char* Link() = 0;

   virtual ~GCRCItem(){};
```

GCRCIBasic The basic implementation of an abstract class `GCRCItem` is done by the class `GCRCIBasic`. It simply implements the two virtual methods above, i.e. `Reference` and `Link`; the method `Link` return `NULL`. There is a new method `Load` that let you set the class's attribute `gc`. The attribute is of type `GCReference` that is address in the next subsections.

```
private:
   GCReference gc;
public:
   bool Load(GCReference gc);
   virtual GCReference Reference();
   virtual const char* Link();

   virtual ~GCRCIBasic();
```

GCRCILink A `GCRCILink` object is a soft link to an object. It defines the same method as a `GCRCIBasic` but the `Load` method as a `const char*` argument instead of a `GCReference`. There is only one attribute of type `level0::BString`. The method `Link` return the attribute `objectPath`.

```
private:
   BString objectPath;
public:
   bool Load(const char* objectPath);
   virtual GCReference Reference();
   virtual const char* Link();

   GCRCILink();
   virtual ~GCRCILink();
```

GCReference

`[GCReference.h, GCReference.cpp]`
A class managing a pointer to an `Object` of type `GarbageCollectable`. The class implements a garbage collection mechanism that will destroy a class only when all references are destroyed. Access to the methods of the referred object is obtained via the `()` operator. Shared use of the same reference on multiple threads (static) needs semaphore protection. Use of different references to same object is safe. The only attribute of the class is a pointer to a `GarbageCollectable` object, that is private.

```
protected:
   GarbageCollectable* objectPointer;
```

We now explore the differents methods of the class. The first method, that is an operator overloading is like that to prevent users to make a copy of a reference by taking its address. The method `IncrementReference` allow to increment the usage count of the instantiated object.

Then follow a group of constructors and some copy constructors. The first two methods create a new `GCReference` object accepting a `GarbageCollectable` object pointer as an argument. Next constructor creates an empty reference, the other one creates a new reference from an existing one. The last constructor creates a new object of tyepe `typeName` and builds a reference to it. This is done by searching the *Object Registry Database* for the class by name.

```
private:
   GCReference* operator&();
   void IncrementReference();

protected:
   GCReference(GarbageCollectable* pointer);
   GCReference& operator=(GarbageCollectable* pointer);
public:
   GCReference();
   GCReference(const GCReference& object);
   GCReference(const char* typeName);
   GCReference& operator=(const GCReference& reference);
```

In the last sections of this chapter we will look at the basics of what a *Configuration Database* is. For now just think at a `Configuration Database` as a configuration file like `*.ini` file in MS Windows

or any file in the */etc/* UNIX directory.

We now introduce some piece of such configuration files that enable the cretion of objects using that files via the class `GCReference`. The method `ObjectLoadSetup` that is the same we have seen in the `Object` class, it is not an overloading because `GCReference` doesn't inherits from any other class. Below is shown how to write a snippets of configuration file to create a class by name.

```
Name = {
   Class = <class Name>
   <Class specific>
}
```

A name can also be a linK to an object in the *Global Object Database*. To load and save objects via a configuration file you need the methods `ObjectLoadSetup` and `ObjectSaveSetup`.

```
virtual bool ObjectLoadSetup(ConfigurationDataBase& info,
          StreamInterface* err,
          bool createOnly=False);
virtual bool ObjectSaveSetup(ConfigurationDataBase& info,
          StreamInterface * err);
```

The method `IsValid` estabilishes if the reference is pointing to a valid object or not; the method `NumberOfReference` gets the number of references the `GCReference` object holds. Follow some operator redefinitions. Then come the `Clone` method that creates a reference to a duplicate object; this is necessary, otherwise when `GCReference::Clone` is called by `GCRTemplate`, at this point the `IsValid` function of `GCRTemplate` would be called, returning false as the setup of `GCRTemplate` templatized object is not yet done. The method `RemoveReference` correctly removes the reference. The distructor simply call this last function. If the reference count goes to zero the object will be destructed.

```
virtual bool IsValid() const;
inline int32 NumberOfReferences() const;

inline bool operator==(const GCReference& reference) const;
inline bool operator!=(const GCReference& reference) const;
inline GarbageCollectable* operator->() const;

inline bool Clone(const GCReference &reference);
virtual void RemoveReference();

virtual ~GCReference();
```

GCRTemplate

`[GCRTemplate.h]`
Definition of the widely used `GCRTemplate` class. This is a subclass of the `GCReference` class and it provides method overriding. Infact most of the `GCReference` methods are overriden. This template generates a specialised `GCReference` that is able to refer to objects of class `T` derivates.

`A GCRTemplate is an holder of a reference to an object. It helps garbage collection by increment and decrement the object usage count.`

The template class has only one (protected) attribute of type we are templatized for, called `typeTObjectPointer`.

```
protected:
   T* typeTObjectPointer;
```

We now skip operator redefinitions and we analize only other methods. The method `Init` simply sets all the class and superclass attributes to null values; the method `Load` first call `Init` and then

using the argument passed by sets the class attibutes (`objectPointer` and `typeTObjectPointer`).

The first constructor coming is the default constructor and it simply call `Init`; the next constructor creates an empty reference or a reference to base type `T`, if argument `create` has value `GCFT_CreateInstance` then it will also create the object (of type `T`).

The third constructor creates a reference to an object descendent from base type `T`. Recommended usage is `GCRTemplate<T>(new MYClass())`.

Fourth constructor creates a new reference copting from a generic one, the operation might fail, in which case the reference produced is invalid. The user has no feedback to understand if the operation was done with success or not, so it must test the object calling the `IsValid` method. This operation is done via the `operator=` that use the `dynamic_cast` operator.

Next constructor creates a new reference from an existing one and the last one creates a new reference to an object spcified by name (via `typeName` argument).

```
template<typename T>
class GCRTemplate : public GCReference{
private:
   GCRTemplate<T>* operator&();
protected:
   void Init();
   void Load(T *p);

public:
   GCRTemplate();
   GCRTemplate(GCFlagType create);
   GCRTemplate(T *p);
   GCRTemplate(const GCReference& object);
   GCRTemplate(const GCRTemplate<T>& object);
   GCRTemplate(const char* typeName): GCReference(typeName);

   GCRTemplate<T>& operator=(const GCRTemplate<T>& reference);
   GCRTemplate<T>& operator=(const GCReference& reference);
```

Some overridden methods comes now; it is really interesting that each different class that can be loaded from a configuration file need to override the `ObjectLoadSetup` method to personalize its initialization beacuse each object needs its own params. `IsValid` estabilish if the reference is pointing to a valid object. In the same way `Clone` methods and the `RemoveReference` act as before but are specific for the subclass.

```
   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,
                  StreamInterface* err,
                  bool createOnly=False);

   virtual bool IsValid() const;

   bool operator== (const GCRTemplate<T>& reference);
   T* operator->()const;

   inline bool Clone(const GCReference& reference);
   inline bool Clone(const GCRTemplate<T>& reference);
   virtual void RemoveReference();

   virtual ~GCRTemplate();
```

GCReferenceContainer

`[GCReferenceContainer.h, GCReferenceContainer.cpp]`

A `GCReferenceContainer` is a container (a list) of objects that are `GarbageCollectable`. If the object are also descendent from `GCNamedObject` then they can be accessed also by name.

The most important attribute in this class is the `list` it holds all objects contained by the conteiner. Other two attributes are a semaphore and the associated timeout, those attribute lets the access to the object be in an ordinate fashion to achieve data consistency.

```
protected:
   LinkedListHolder list;

   MutexSem mux;
   TimeoutType msecTimeout;
```

The method `CreateListElement` creates a new `GCRCILink` or `GCRCIBasic` object depending if the argument `objectPath` passed by is filled in or not. `SetTimeout` sets the internal semaphore timeout in msec. Methods `Lock` and `Lock2` lock the resource, must be unlocked using `UnLock` method.

```
protected:
   LinkedListable* CreateListElement(const char* objectPath,GCReference gc);
public:
   GCReferenceContainer();
   virtual ~GCReferenceContainer();
   void SetTimeout(TimeoutType msecTimeout);

   bool Lock();
   bool Lock2(TimeoutType tt)
   bool UnLock()

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
```

We have just mentioned the last two methods, `ObjectSaveSetup` and `ObjectLoadSetup`, such two are involved in configuration and self instantiation of classes. We now explore the syntax of the accepted configuration parameters of the `GCReferenceContainer`. The basic syntax then follow, valid `<command>` are `Add`, `Remove`, and `AddReference`.

```
   <command> = {
      <command parameters>
   }
```

Imagine we want to add a new class of type `<name of class>` instantiated with the name `name1`, then we can write:

```
   Add = {
      name1 = {
         Class = <name of class>
         <class parameters>
      }
   }
```

A more easy way to do this is:

```
   +name1 = {
      Class = <name of class>
      <class parameters>
   }
```

To add a reference, i.e. to copy the reference from the `Global Object Database` of a class `name2` to a new instance called `name3` you have to write:

```
   AddReference = {
      name2 name3
   }
```

Removing an object can be done listing the named instances or by using special words like `ALL` below.

```
Remove = {
   name1 name2 name3
}
Remove = {
   ALL
}
```

After these few example with procede to finish to explore `GCReferenceContainer`'s methods.

The first `Insert` method adds an element at position `position`, the object added is a reference to the global object specified by `objectPath` argument; if `position` is `0` then the object is added before any other element, if is `-1` is added at the end of the list. The second `Insert` method adds an element at position `position` but now the object added is referenced to by `gc` that is a `GCReference` object pointer and contain the two arguments needed by the previous method.

The method `_Find` is an helper method and is then called by the other following `Find` methods that we are going to explain. The first `Find` finds an object by `name` and returns a reference to it into `reference` argument; if `name` contains separators like `.` than the first segment is matched and the remainder is passed to the matching object if it is a container. If `recurse` is `true` and the pattern cannot be matched from the root then it will try searching also from the subnodes. The string tree is visited side to side. If `unMatched` is provided then a partial match is accepted and the unmatched part of the name is returned in `unMatched` as a pointer to name unmatched substring.

The second `Find` finds an object and returns a reference to it into `reference`, uses the `selector` `Test2` function (not `Test`) thus distinguishes between not found and not the right path and notifies of returning from recursion level using `SFTTBack`. One can match the partial finding on the way back out of recursion by returning `SFTTFound` in this last case.

The last `Find` gets a reference to the `index` -th element if `referenceAsBString` is `true` then a reference will be returned as a `BString`.

Like the method `_Find` before there is a method `_Remove` that is an helper method called by the following `Remove` methods. The first `Remove` method removes an object by `name` and returns a reference to it into `reference` argument. If `name` contains separators like `.` or `:` than the first segment is matched and the remainder is passed to the matching object if it is a container; if `recurse` is true then it will try searching also the subnodes. The tree is visited side to side.

The second `Remove` removes any element from the list that fit the criteria specified by `selector`.

The last `Remove` removes the `index`-th element of the list.

The method `Cleanup` removes all references from the `GCReferenceContainer`. The method `Iterate` acts on each element of the list using the provided `iterator` if `recurse` is enabled it applies the `iterator` also on the sub containers after and before applying it on the container itself; note that this function does not keep the container locked during the call to the user function, when recursing, only the current container is locked. Last method, `Size` return the number of elements in the list.

```
   inline bool Insert(const char* objectPath,int position=-1);
   inline bool Insert(GCReference gc,int position=-1);

private:
   inline GCReference _Find(const char* name,SearchFilterT<GCReference>* selector=NULL,
             int index=-1,bool recurse=False,const char** unMatched=NULL);
public:
   inline GCReference Find(const char* name,GCFlagType recurse=GCFT_None,
             const char** unMatched=NULL)
   inline GCReference Find(SearchFilterT<GCReference>* selector,
             GCFlagType recurse=GCFT_None);
   inline GCReference Find(int index,bool referenceAsBString=False);
```

```
private:
   inline GCReference _Remove(const char* name,SearchFilterT<GCReference>* selector=NULL,
            int index=-1,bool recurse=False);
public:
   inline GCReference Remove(const char* name,GCFlagType recurse=GCFT_None);
   inline GCReference Remove(SearchFilterT<GCReference>* selector,
            GCFlagType recurse=GCFT_None);
   inline GCReference Remove(int index);

   inline bool CleanUp();
   bool Iterate(IteratorT<GCReference>* iterator,GCFlagType recurse=GCFT_None);
   int Size();
```

GCRCLister

[GCReferenceContainer.h]
Simply list (print) the content of a `GCReferenceContainer`. A `GCRCLister` is an `IteratorT<GCReference>` so it will handle `GCReference` objects that are linked listed in a `GCReferenceContainer`. Such a `GCRCLister` can be constructed only associating a `StreamInterface` object where to list all the elements founded in the `GCReferenceContainer`. The `StreamInterface` is also an attribute of the class, the other attribute, `level` account for the number of recursion levels.

The constructor require only one argument, if `s` is `NULL` the code will use `printf` to write to the console. Two `Do` methods do the work of listing.

```
   StreamInterface* stream;
   int level;
public:
   GCRCLister(StreamInterface* s = NULL);
   virtual void Do(GCReference data);
   virtual void Do2(GCReference data,SFTestType mode);
```

GlobalObjectDataBase

[GlobalObjectDataBase.h, GlobalObjectDataBase.cpp]

The *Global Object Database* is statically declared as a

```
GCRTemplate<GlobalObjectDataBase> globalObjectDataBase;
```

in the file *level1/GlobalObjectDataBase.cpp*. So the *Global Object Database*, taking a look at Figure 2.4 is a sort of template, templatizing with the class `GlobalObjectDataBase` that is of type `GCReferenceContainer`.

We explore first what a `GlobalObjectDataBase` class adds to a `GCReferenceContainer` type. The source follows.

```
class GlobalObjectDataBase: public GCReferenceContainer{
public:
   bool enableDebugging;
   GlobalObjectDataBase(){
      if (enableDebugging)
         DisplayRegisteredClasses(NULL,True);
   }
   virtual ~GlobalObjectDataBase(){
      if (enableDebugging)
         DisplayRegisteredClasses(NULL,True);
   }
};
```

A `GlobalObjectDataBase` class simply adds a debugging facility to a `GCReferenceContainer` class. Attribute `enableDebugging` enable or disable the debugging facility.

The *Global Object Database*, `globalObjectDataBase`, is a `GCRTemplate` that subclass a `GCReference` and then contain a list of `GlobalObjectDataBase` elements.

The *Global Object Database* is the default container of objects, every contained object could be addressed and indexed. When objects will be destroyed the virtualized destructor will be removed from the leafes.

Performs the job of destroying the database note that the global destructor is called in a single thread environment during the final winding down of an application for this reason there is no need to check for the database being in use that is the hope at least.

2.2.1 Design Notes

There is no hierical of the class saved, no inter relationship between classes is stored.

2.3 Error System Instruction

The *Error System Instruction* (ESI) is a subset of the Error Management System (EMS) that create a linked list of errors and errors' behaviour of the system. All code is in header files and must be included in all files that needed Error Management.

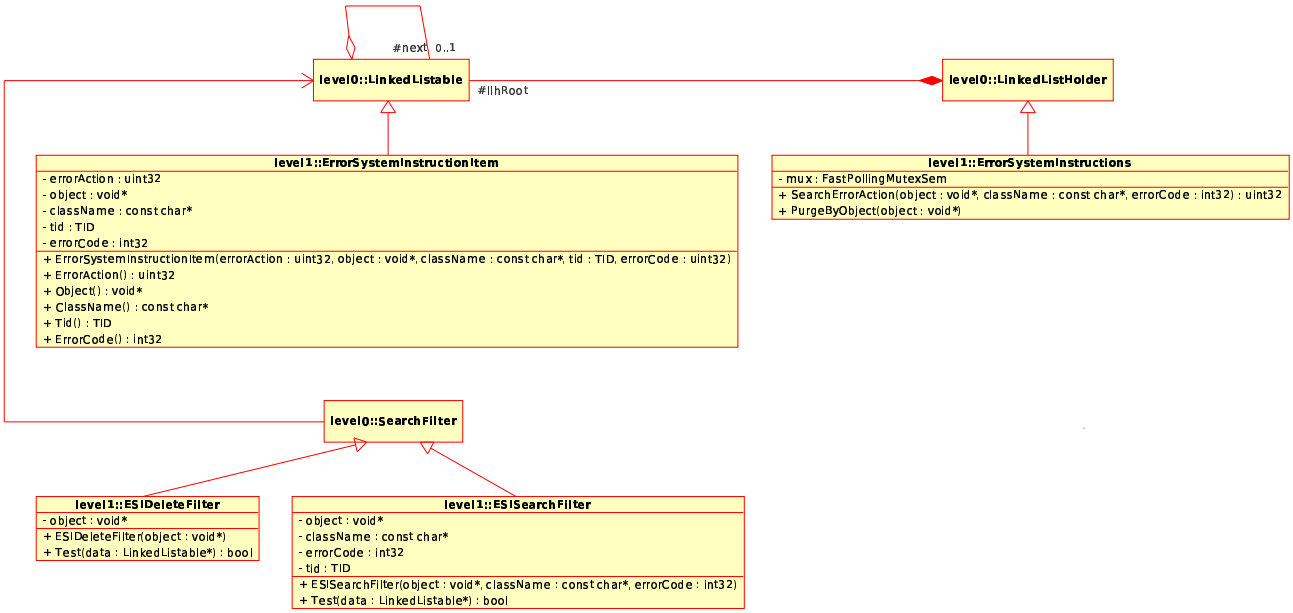

Figure 2.5: BaseLib Level1 Error System Instruction classes

The UML diagram is depicted in Figure 2.5 but if you look also to Figure 2.1 you can note that also `Object` relies on top of the ESI infrastructure; for each class type there is also an ESI item that decides what action to take in case of an error condition.

ESI is another linked listable structure that accounts about errors, i.e. is a catalog of errors and actions, classes involved in this level are:

- ErrorSystemInstructionItem
- ErrorSystemInstructions

- ESIDeleteFilter, ESISearchFilter

ErrorSystemInstructionItem

[ErrorSystemInstructionItem.h]
The basic component of the ESI is the class `ErrorSystemInstructionItem`, there is a class like that for every error and basically tall the system what to do in case a specific error.

We start with the attributes, the first attribute, `errorAction`, tell the system what to do with this error, `object` is the address of the object the action is restricted to, attribute `className` is the name of class the action is restricted to, `tid` is the thread identification the action is restricted to and `errorCode` is the error code the action is restricted to. If you choose to not restrict error management to any of the previous attributes simply sets them to `NULL`.

The constructor lets you set the policy for an error, all the other methods let you get the attributes of the class.

```
   uint32 errorAction;
   void* object;
   const char* className;
   TID tid;
   int32 errorCode;
public:
   ErrorSystemInstructionItem(uint32 errorAction,
         void* object,
         const char* className,
         TID tid,
         int32 errorCode);
   uint32 ErrorAction();
   void* Object();
   const char* ClassName();
   TID Tid();
   int32 ErrorCode();
```

Some defined `ErrorSystemInstructionItem` in file *level1/ErrorSystemInstructionItem.h*:

```
#define ANY_VALUE NULL

#define OBJECT_ERRORINSTRUCTION(errorAction,object,code) \
   new ErrorSystemInstructionItem(errorAction,object,ANY_VALUE,ANY_VALUE,code)
#define ERRCODE_ERRORINSTRUCTION(errorAction,code) \
   new ErrorSystemInstructionItem(errorAction,ANY_VALUE,ANY_VALUE,ANY_VALUE,code)
#define CLASS_ERRORINSTRUCTION(errorAction,classInfo,code) \
   new ErrorSystemInstructionItem(errorAction,ANY_VALUE,classInfo,ANY_VALUE,code)
#define TID_ERRORINSTRUCTION(errorAction,tid,code) \
   new ErrorSystemInstructionItem(errorAction,ANY_VALUE,ANY_VALUE,tid,code)
```

ErrorSystemInstructions

[ErrorSystemInstructions.h]
The class `ErrorSystemInstructions` is a collection of `ErrorSystemInstructionItem`s, i.e. a container of `ErrorSystemInstructionItem`: what to do in case of an error. It is a subclass of `LinkedListHolder`.

The only added attribute to the `LinkedListHolder` class is a `FastPollingMutexSem` that manages the access to the resources. The method `SearchErrorAction` searches using one of the three keys as arguments an method `PurgeByObject` remove the `ErrorSystemInstructionItem` associated to an object passed by pointer.

```
   FastPollingMutexSem mux;
public:
   uint32 SearchErrorAction(void *object,const char *className,int32 errorCode);
   void PurgeByObject(void *object);
```

ESIDeleteFilter, ESISearchFilter

[ErrorSystemInstructions.h]
Methods implemented in the previous `ErrorSystemInstructions` class work using the class defined in this section: `ESIDeleteFiler` and `ESISearchFilter`. The first one, `ESIDeleteFilter`, is a filter to delete `ErrorSystemInstructionItem`; and the last one, `ESISearchFilter`, searches for `ErrorSystemInstructionI` by different criteria.

2.4 Configuration Database

The *Configuration Database* (CDB) is one of the key concepts upon BaseLib is developed. There are not to many classes in this level concerning the CDB, infact its classes are spreaded across many levels. In this level the developer will find a sort of interface to CDB: the `CDBVirtual` abstract class that by the way can be considered as an interface but has one non pure virtual method.

In next levels we will face with three CDBs' implementations (CDB, CDBOS, MMCDB). If one want to go a bit further can write a CDB that parse XML file for example and make it work in the framework simply let it work observe the interface we are now discussing about in this section.

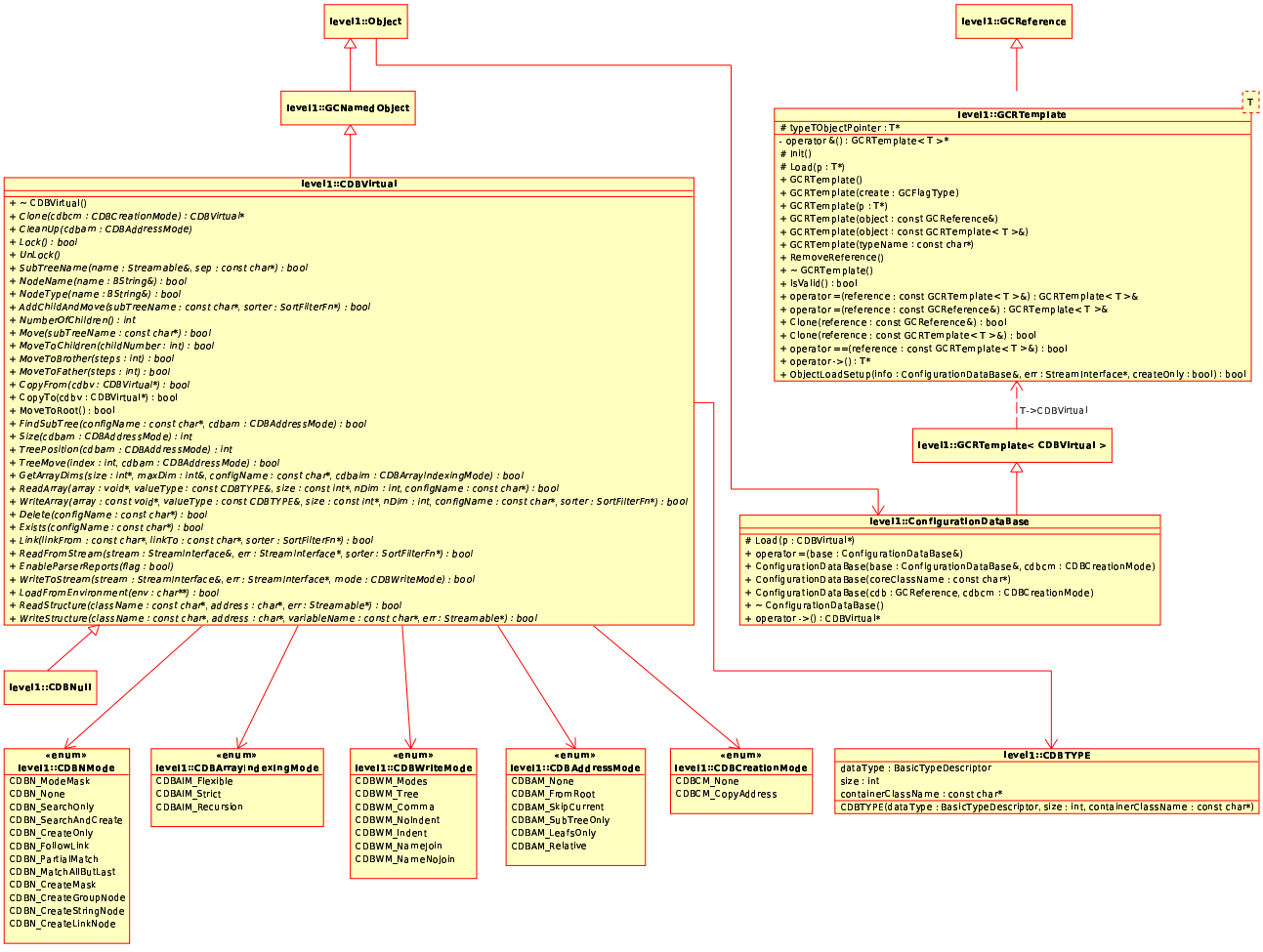

Figure 2.6: BaseLib Level1 Configuration Database classes

Figure 2.6 depicts the UML diagram of the involved classes that are:

- CDBVirtual
- CDBNull

- ConfigurationDataBase

CDBVirtual

[CDBVirtual.h, CDBTypes.h]
The class CDBVirtual provide an interface to a hierarchical database. It works similarly to a filesystem having directory nodes (branches) and file nodes (leaves). Each leaf node is a container of a sequence of characters, both ASCII and not ascii. CDB is infact a reference to an instance of the database. Copying CDBs does not duplicate the database but increases a reference counter. Operation to the same database by different threads is safe. The methods of this class are not usable directly outside BaseLib Use CreateCDB to create an instance of CDB.

Class CDBVirtual has many pure virtual methods, we go throught those. In the file *level1/CDBTypes.h* all the enumeration used are can be founded.

The deconstructor will destroy database if instance count goes to zero. The method Clone creates a new reference to a database, or if that is not possible it creates a copy; cdbcm=CDBCM_CopyAddress ensures that the new object points at the same location. The method CleanUp removes all the content from the database unless some other database instance exists and points to a subtree, one can reques to delete only the current subtree. The method Lock locks the main database for exclusive access: use if a group of transactions should be atomic; UnLock unlocks the main database: remember to unlocking a locked database after useing it.

```
virtual ~CDBVirtual();
virtual CDBVirtual* Clone(CDBCreationMode cdbcm)=0;
virtual void CleanUp(CDBAddressMode cdbam=CDBAM_FromRoot) = 0;
virtual bool Lock() = 0;
virtual void UnLock() = 0;
```

The method SubTreeName finds the overall path leading to the current node, return the path in the argument name, NodeName returns the name of the current node in name; NodeType return the type of the current node in name; the method AddChildAndMove can be used to create a new subtree.

The method NumberOfChildren tells how many branches from this node, negative number implies that the location is a leaf; Move moves to the specified location; the movement is relative to the current location, UpNode is one level up, RootNode is all the way up. The mehtod MoveToChildren lets you move in the tree, negative numbers move up, 0 is the first of the children and a positive number is the n-th children; MoveToFather let s you move in the tree, 0 means remain where you are, greater then 0 to brothers on the right below 0 on the left. The method CopyFrom and CopyTo copy from the CDB to another one. MoveToRoot moves back to the root of the tree. The method FindSubTree searches on the right of the tree for the subtree identified by the string configName; on success nodes points to the node containing the subtree or leaf; it will not follow links; the only argument it supports is CDBAM_SkipCurrent that allow a search in the current subtree excluding current node any other flag are NOT SUPPORTED. The method Size gives the total number of nodes; argument can be CDBAM_SubTreeOnly allowing to measure the current subtree, or CDBAM_LeafsOnly allowing counting only the data nodes. TreePosition returns the position of the node in the left to right bottom to top order the absolute location of a node; CDBAM_LeafsOnly allows counting only the data nodes. The method TreeMove moves to a location within the whole (sub)tree; the nodes are numbered from left to right and from subnode to supernode; if the node does not exist returns False and remains in the start position.

```
virtual bool SubTreeName(Streamable& name,const char* sep = ".") = 0;
virtual bool NodeName(BString& name) = 0;
virtual bool NodeType(BString& name) = 0;

virtual bool AddChildAndMove(const char* subTreeName,
   SortFilterFn* sorter=NULL) = 0;
virtual int  NumberOfChildren()=0;
```

```
virtual bool Move(const char* subTreeName) = 0;
virtual bool MoveToChildren(int childNumber=0) = 0;
virtual bool MoveToBrother(int steps = 1) = 0;
virtual bool MoveToFather(int steps = 1) = 0;
virtual bool CopyFrom(CDBVirtual* cdbv) = 0;
inline bool  CopyTo(CDBVirtual* cdbv);
inline  bool MoveToRoot();

virtual bool FindSubTree(const char* configName,
   CDBAddressMode cdbam=CDBAM_None)=0;
virtual int  Size(CDBAddressMode cdbam) = 0;
virtual int  TreePosition(CDBAddressMode cdbam=CDBAM_LeafsOnly) = 0;
virtual bool TreeMove(int index,
   CDBAddressMode cdbam=CDBAM_LeafsOnly) = 0;
```

The method `GetArrayDims` if doesn't fail returns the `size`, `maxDim` and `configName` of the CDB. The method `ReadArray` read the array, if `size` is `NULL` it treats the input as a monodimensional array of size `nDim`, if `nDim` is zero this is a standard variable equivalent of a vector of size 1, if size does not agree with the actual dimensions of the matrix the routine will fail. We can tell the same for the `WriteArray` method.

The method `Delete` removes an entry or subtree (position is relative) to delete a link use the `linkTo` as the leaf name to delete a subtree simply specify the group node. The method `Exists` tell you whether a certain entry exists. The method `Link` make a link, `linkFrom` is the path where the link is created from.

The method `ReadFromStream` reads a database from a stream; `EnableParserReports` enables reports of parser during `ReadFromStream` into error stream. `WriteToStream` writes the database to stream without any ordering; `LoadFromEnvironment` loads from environment or from any `NULL` terminated list of chars.

The method `ReadStructure` read from CDB to memory and the method `WriteStructure` either copies or references a memory structure, at address `address` and of type `className` CDB transforms the memory.

```
virtual bool GetArrayDims(int* size, int& maxDim, const char* configName,
   CDBArrayIndexingMode cdbaim=CDBAIM_Flexible) = 0;
virtual bool ReadArray(void* array, const CDBTYPE& valueType,
   const int* size, int nDim, const char* configName) = 0;
virtual bool WriteArray(const void* array, const CDBTYPE& valueType,
   const int* size, int nDim, const char* configName,
   SortFilterFn* sorter=NULL) = 0;

virtual bool Delete(const char* configName) = 0;
virtual bool Exists(const char* configName) = 0;
virtual bool Link(const char* linkFrom,
   const char* linkTo,
   SortFilterFn* sorter=NULL) = 0;

virtual bool ReadFromStream(StreamInterface& stream,
   StreamInterface* err=NULL,SortFilterFn* sorter=NULL) = 0;
virtual void EnableParserReports(bool flag) = 0;
virtual bool WriteToStream(StreamInterface& stream,
   StreamInterface* err=NULL,CDBWriteMode mode=CDBWM_Tree) = 0;
virtual bool LoadFromEnvironment(char** env) = 0;

virtual bool ReadStructure (const char* className,
   char* address,
   Streamable* err=NULL) = 0;
virtual bool WriteStructure(const char* className,
```

```
      char* address,
      const char* variableName=NULL,
      Streamable* err=NULL) = 0;
```

CDBNull

[CDBNull.h]
Class CDBNull is a dummy CDB just to create valid references. Each method is implemented returning zero or False.

ConfigurationDataBase

[ConfiguratonDataBase.h]
Differently from other data structures that we have founded during this chapter, like ORDB and GODB, you can instantiate how many *Configuration Database*s you need and there is no static and globally defined CDB. A ConfigurationDataBase contains a reference to CDB of type CDBVirtual because it inherits from GCRTemplate<CDBVirtual>. Forces call to CDB to be made via the virtual table.

The method Load copy the CDBVirtual pointer in the GCRTemplate templatized attribute typeTObjectPointer after type checking. The first constructor creates a new database if base paramenter is NULL, if not NULL it creates a reference to an existing one. The second constructor creates a CDB from a generic base type.

```
protected:
   void Load(CDBVirtual *p);
public:
   inline ConfigurationDataBase(ConfigurationDataBase &base,
      CDBCreationMode cdbcm=CDBCM_CopyAddress);
   inline ConfigurationDataBase(const char* coreClassName="CDB");
   inline ConfigurationDataBase(GCReference cdb,
      CDBCreationMode cdbcm=CDBCM_CopyAddress);
   inline void operator=(ConfigurationDataBase &base);

   virtual  ~ConfigurationDataBase();
   CDBVirtual *operator->();
```

2.4.1 Design Notes

The class ConfigurationDataBase declared with all inline methods in *level1/ConfigurationDataBase.h* is included in many BaseLib's files, this way the code will be compiled to many times wasting space. Class CDBVirtual looks so heavy could be stripped off?

Chapter 3

BaseLib Level 2

A series of classes are dedicated to strings and data streaming. These allow to perform a series of advanced operations in character sequences and to perform input/output (I/O) operations without requiring the concept of beginning and end of transmission. Most of the I/O classes are abstracted to a level where the knowledge of the target media (socket, memory, ...) isn't required. Such classes come from BaseLib Level 2 and inherits from `level0::StreamInterface` and its implementation `level2::Streamable`.

The main concept in this chapter is the `Streamable` class that is presented before every other class. As usual we discuss each section in a logical order; Level2 asa said before implements the concept of stream, a common concept in C++ and in Java, slighty new to C programmers that are used to work with files, sockets or strings. All this concepts are groupped togheter and are accesible via the `level0::StreamInterface`. `Streamable` implements `StreamInterface` and adds some metods and attributes.

Analysis is carried on examining the differents stream object grouped in two sets: memory (buffer, string handling) and files (also sockets that are no more than files in Unix view). The whole BaseLib Level 2 could be divided in the following sections:

- Streamable (memory, files)
- CDB
- IO devices (serial line, keyboard)

3.1 Streamable

Before reading this section is really important that one has read *BaseLib Level0 Streams* section. Such a section introduces streams and the `StreamInterface` from which the `Streamable` inherits from. This section treats first the class `Streamable` and then we have a look first to memory streams and then to file streams. We take a look first to memory streams because there is an important class on which the `Streamable` class relies on. To have a pictorial idea of the `Streamable` class introduced below refer to Figure 3.1 and Figure 3.2.

Streamable

`[Streamable.h, Streamable.cpp]`
Most of the methods in `Streamable` class are an implementation of what is simply declared in the `StreamInterface` class in Level0. `Streamable` is the prototype for all streams, it adds to StreamInterface some buffering and token searching functions.

The class `Streamable` relies on the `CStreamBuffering` class described in the following sections, of such type are the first two attributes, the first is the *input* buffering area and the second is the *output* buffering area (`csbIn` and `csbOut`). Last attribute is a stream identifier, if hold the value `0` means the default stream and if is `-1` it means no stream.

The constructor simply set to `NULL` each attribute and the destructor does nothing.

```
protected:
   CStreamBuffering* csbIn;
   CStreamBuffering* csbOut;
   uint32 selectedStream;
public:
   Streamable();
   virtual ~Streamable();
```

The first two methods we encounter, `SSRead` and `SSWrite` are two virtual methods not implemented here but to be implemented by any subclass. `CanRead` and `CanWrite` simply return `False` because methods `SSRead`, `SSWrite` are not implemented for the class. Methods `Read` and `GetC` call `SSRead`; methods `Write` and `PutC` call `SSWrite`.

```
protected:
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault)=0;
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault)=0;

public:
   virtual bool Read(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);

   virtual bool CanRead();
   virtual bool CanWrite();

   inline bool GetC(char& c);
   inline bool PutC(char c);
```

The method `Size` must return the size of the file, in this case it will return`-1` because `Streamable` is an abstract class; the same happens for the method `Position`. Methods `SetSize`, `Seek` and `CanSeek` return `False`. All those methods must be implemented in a subclass.

The method `NOfStreams` return `1`, such method return how many streams are available, override it with a different value if multiple stream are supported. The first `Switch` method select the stream to read from, switching may reset the stream to the start, `0` is the default stream, `false` is only returned if the switch was not performed. Also the second `Switch` selects the stream to read from, must be overriden with a method providing name to number mapping. The method `SelectedStream` returns what stream has been selected and `StreamName` returns the name of the stream we are using. The method `AddStream` adds a new stream to write in and `RemoveStream` removes an existing stream. Those last two methods are not implemented in this class and return `false`.

The method `CompleteRead` performs the job of a the read function but guarantees the completion; in case of failure, `size` returns the actual data read; `timeOutMs` is the total allowed wait time checked using `level0::HRT`. The method `CompleteWrite` performs the job of a write function but guarantees the completion; in case of failure size returns the actual data written; those functions are implemented in the *Streamable.cpp* file.

```
   virtual int64 Size();
   virtual bool SetSize(int64 size);
   virtual int64 Position(void);
   virtual bool Seek(int64 pos);
   virtual bool CanSeek();

   virtual uint32 NOfStreams();
   virtual bool Switch(uint32 selectedStream);
```

```
virtual bool Switch(const char* name);
virtual uint32 SelectedStream();
virtual bool StreamName(uint32 n,char* name,int nameSize);
virtual bool AddStream(const char* name);
virtual bool RemoveStream(const char *name);

virtual bool CompleteRead(void* buffer,uint32& size,
   TimeoutType msecTimeout = TTInfiniteWait);
virtual bool CompleteWrite(const void* buffer,uint32& size,
   TimeoutType msecTimeout = TTInfiniteWait);
```

Now we introduce three functions firstly implemented in the `Streamable` class, those function make use of the attributes in the `Streamable` class.

The method `BufferedRead` writes the buffer from the stream; it's better to not remap this function unless the class needs to deal with buffering in a different way. The method `BufferedWrite` reads and copies data into the stream; it is important to not remap this function unless the class needs to deal with buffering in a different way, like the previous method. `Flush` forces to complete buffered operations.

```
virtual bool BufferedRead(void* buffer,uint32& size,
   TimeoutType msecTimeout = TTDefault);
virtual bool BufferedWrite(const void* buffer,uint32& size,
   TimeoutType msecTimeout = TTDefault);
inline void Flush();
```

The method `VPrintf` will write any text using printf format (is buffered if buffering is activated), the others `Print` functions print, respectively, an integer, a flot or a string with desidered size and padding to the output stream registered with the `Streamable` class (attribute `csbOut`).

```
bool VPrintf(const char* format,va_list argList);
inline bool Print(int32 n,int desiredSize=0,
   char desiredPadding=0,char mode = 'i');
inline bool Print(double f,int desiredSize=0,
   int desiredSubSize=6,char desiredPadding=0,char mode='f');
inline bool Print(const char* s,int desiredSize=0,
   char desiredPadding=0);
```

First `GetToken` method extracts a token from the stream into a string data until a terminator or 0 is found; `maxSize` is the buffer size, the maximum string size is `maxSize-1`; it skips all `skipCharacters` chars even if calssified also as terminators if at the beginning returns `true` if some data was read before any error or file termination. `false` only on error and no data available; the terminator (just the first encountered) is consumed in the process and saved in `saveTerminator` if provided. This is a buffered method. Second `GetToken` method behave in the same way as before but takes as the first argument an `StreamInterface` object. A character can be found in the terminator or in the `skipCharacters` list in both or in none. See Table 3.1 for an explanation on how it work. The same things can be said for the third `GetToken` but it takes a string as a `Streamable` reference.

char in	action preformed
none	the character is copied
terminator	the character is not copied the string is terminated
skip	the character is not copied
skip and terminator	the character is not copied, the string is terminated if not empty

Table 3.1: Level2 GetToken actions

`SkipTokens` skips a series of tokens delimited by terminators or 0. `GetLine` will skip an empty line or any part of a line termination (the first working on a `Streamable` input and the second on a `char*` input extracting the substring after skipping the required line).

Next two methods are `static`, the first one `GetCStringToken` extracts a token from the string into a string until a terminator or 0 is found, the maximum string size is $maxSize - 1$, it returns `true` if some data was read. False only on no data available. The method `DestructiveGetCStringToken` extracts a token from the string into a string until a terminator or 0 is found and affects the input by placing 0 at the end of each token.

Last method, `FindPattern` extracts data from the stream until the matching string is found the pointer is to the first character after the pattern.

```
virtual bool GetToken(char* buffer,const char* terminator,
   uint32 maxSize,char* saveTerminator=NULL,const char* skipCharacters=NULL);
virtual bool GetToken(StreamInterface& output,const char* terminator,
   char* saveTerminator=NULL,const char* skipCharacters=NULL);
virtual bool GetToken(Streamable& output,const char* terminator,
   char* saveTerminator=NULL,const char* skipCharacters=NULL);

virtual bool SkipTokens(uint32 count, const char* terminator);
virtual bool GetLine(Streamable& output, bool skipTerminators=True);
virtual bool GetLine(char* buffer,uint32 maxSize,bool skipTerminators=True);

static inline bool GetCStringToken(const char*& input,char* buffer,
   const char* terminator,uint32 maxSize);
static inline char* DestructiveGetCStringToken(char*& input,const char* terminator,
   char* saveTerminator=NULL,const char* skipCharacters="");

inline bool FindPattern(const char *pattern);
```

3.1.1 Memory Streams

Memory streams are all streams involving memory operations like strings and buffers. All classes that depend on this group are depicted in the UML diagram of Figure 3.1.

Class involved are listed below. Most of the classes are quite similar, the only notable difference is the `CStreamBuffering` class that is not a subclass of the `Streamable` class. We start analysing this class.

- CStreamBuffering
- BufferedStream
- SXMemory, SXNull
- FString, GCNString

CStreamBuffering

`[CStreamBuffering.h, CStreamBuffering.cpp]`
This class basically maps a `StreamInterface` to a `CStream` class providing buffering to the `CStream`.

The class has infact as attributes a `StreamInterface` pointer and a `CStream`. The constructor let the user set those attributes. The working mode, attribute `mode`, can be set with `UseAsOutput` and `UseAsInput`.

```
public:
   char* saveBuffer;
   uint32 bufferSize;
   StreamInterface* stream;
   CStream cs;
   int mode;
public:
   CStreamBuffering(StreamInterface* stream,char* buffer,int bufferSize);
```

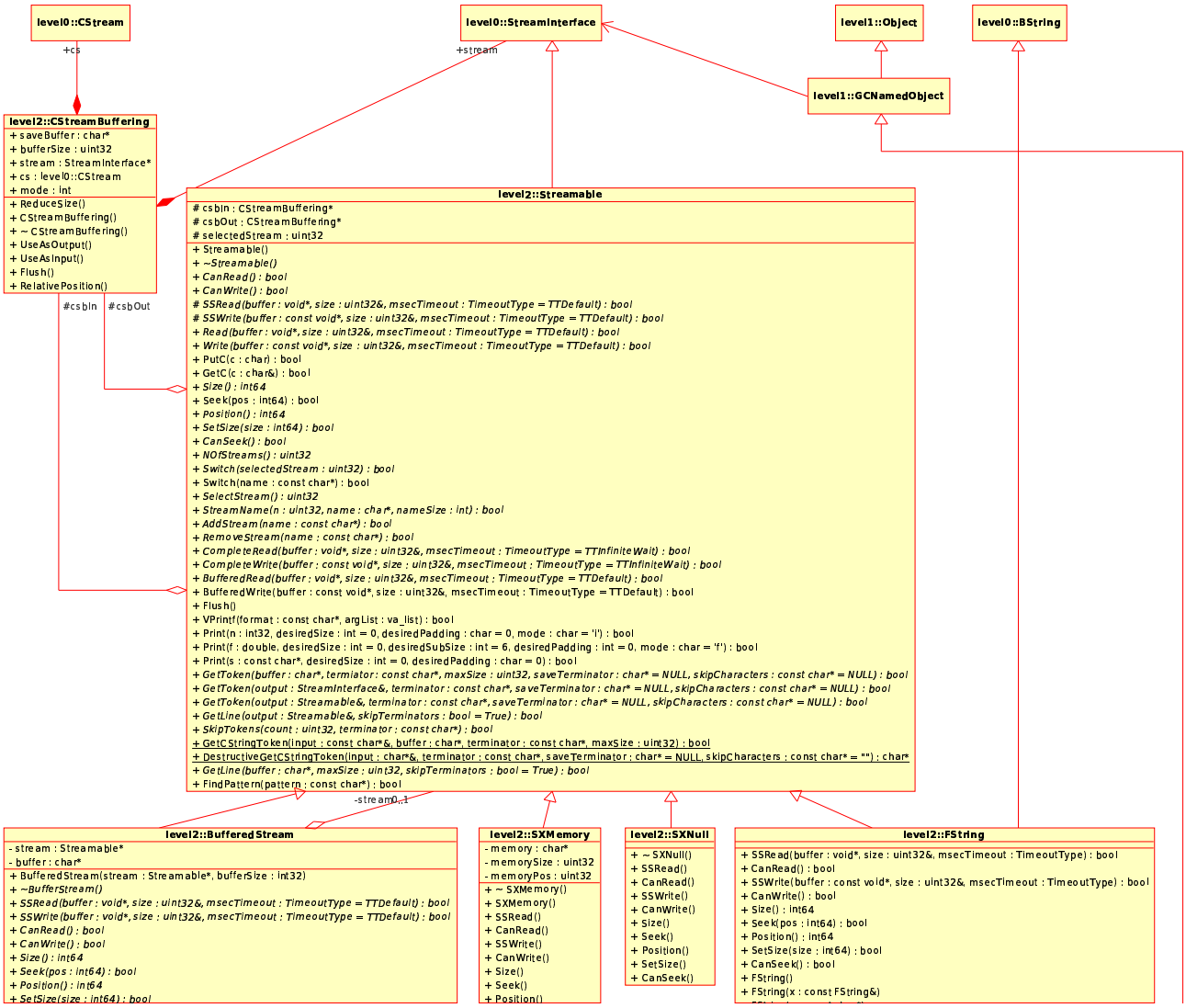

Figure 3.1: BaseLib Level2 memory stream classes

```
virtual ~CStreamBuffering();

void ReduceSize(int bufferSize);
CStream* UseAsOutput();
CStream* UseAsInput();
void Flush();
int32 RelativePosition();
```

BufferedStream

[BufferedStream.h, BufferedStream.cpp]
Class BufferedStream provides full buffering support to a Streamable class by wrapping the non buffering functions. Those functions are basically the Streamable protected SSRead and SSWrite.

To buffer a Streamable it has attributes a Streamable and a char* that will be the buffering area. The constructor require a Streamable that will be firtly saved in the associated attribute and then buffered, the buffering area will be required via the argument bufferSize. Take care because the constructor if initialized with a NULL Streamable pointer will exit without creating a valid object.

```
private:
   Streamable* stream;
   char* buffer;
public:
   BufferedStream(Streamable* stream,int32 bufferSize);
   virtual ~BufferedStream();
```

This not so difficult implementation in SSRead and SSWrite call BufferedRead and BufferedWrite on the saved Streamable attribute to achieve buffered operations.
All other methods call the respective method of the saved Streamable attribute.

```
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool CanRead();
   virtual bool CanWrite();

   virtual int64 Size();
   virtual bool  Seek(int64 pos);
   virtual int64 Position(void);
   virtual bool  SetSize(int64 size);
   virtual bool  CanSeek();

   virtual uint32 NOfStreams();
   virtual bool Switch(uint32 n);
   virtual bool Switch(const char* name);
   virtual uint32 SelectedStream();
   virtual bool StreamName(uint32 n,char* name,int nameSize);
   virtual bool AddStream(const char* name);
   virtual bool RemoveStream(const char* name);
```

SXMemory, SXNull

[SXMemory.h, SXNull.h]
The class SXMemory is a *Streamable eXtension Memory* for a buffer; i.e.it provides read and write operation from/to a buffer in memory. With this class it is possible to use a piece of memory user allocated like a stream.

The memory to be used must be previously allocated and then, you have to use the obtained pointer as an argument to the class's constructor. The constructor doesn't check for arguments consistency and simply store the arguements in the attributes. memoryPos will be set to zero.

```
   char* memory;
   uint32 memorySize;
   uint32 memoryPos;
public:
   SXMemory(char* mem,uint32 size);
   virtual ~SXMemory();
```

Also in this class the main methods are SSRead and SSWrite that mainly make use of memcpy to do its work. Other methods are straightforward.

```
virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
virtual bool CanRead();
virtual bool CanWrite();

virtual int64 Size();
virtual bool Seek(int64 pos);
virtual int64 Position(void);
virtual bool SetSize(int64 size);
virtual bool CanSeek();
```

Like you can see in the UML diagram of Figure ?? there is also a class `SXNull` who's name stand for *Streamable eXtension Null* that is a null or no stream class, so it not implement any stream but is a dummy one.

FString, GCNString

`[FString.h, FString.cpp, GCNString.h, GCNString.cpp]`
The class `FString` extends `BString` and `Streamable` and concretely is a class containing a string simulating a file, or better a streamable interface of a string. Reading a and writing are done on a specific position.

It holds no attributes and implements, like in the previous classes, protected `SSRead` and `SSWrite` methods. Strings operator concatenation and copy are overridden.

```
public:
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool CanRead();
   virtual bool CanWrite();

   virtual int64 Size();
   virtual bool Seek(int64 pos);
   virtual int64 Position(void);
   virtual bool SetSize(int64 size);
   virtual bool CanSeek();

   FString();
   FString(const FString& x);
   FString(const char* s);
   virtual ~FString();

   inline bool operator=(const FString& s);
   inline bool operator=(char c);
   inline bool operator=(const char* s);
   inline bool operator=(const BString& s);
   inline bool operator+=(const char c);
   inline bool operator+=(const char* s);
   inline bool operator+=(BString& s);
   inline bool operator==(BString& s) const;
   inline bool operator==(const char* s) const;
```

The class `GCNString` is an `FString` with a `GCNamedObject` parenthood. The class implements no further methods.

3.1.2 File Streams

File streams are all streams involving file operations. File operations are intended in the UNIX way, i.e. for example in UNIX sockets but also console are files. All classes that depend on this group are depicted in the UML diagram of Figure 3.2.

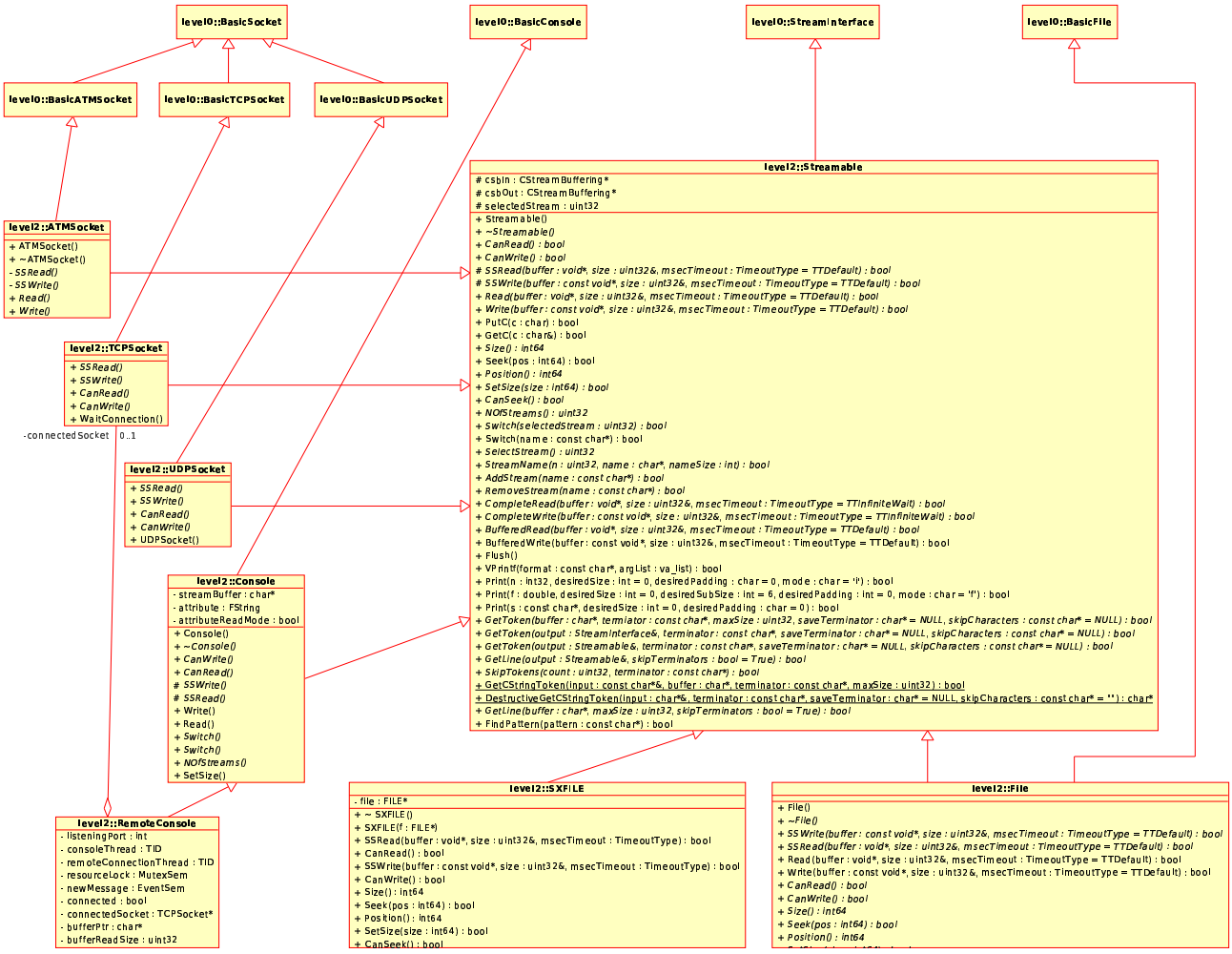

Figure 3.2: BaseLib Level2 file streams classes

Class involved are listed below. Most of the classes are quite similar, classes can be groupped in three subcategories: network (sockets), console and file. But we doesn't follow this distintiction.

- ATMSocket
- TCPSocket
- UDPSocket
- Console
- RemoteConsole
- File
- SXFILE

ATMSocket

[ATMSocket.h, ATMSocket.cpp]
The class ATMSocket adds the streamable capability to a BasicATMSocket, i.e. a BasicATMSocket can also be treat as a stream of data. In detail such class implements ATM sockets AAL5 layer.

The method SSRead reads a block of data from the socket, size is the maximum size, on return size is what was read, timeout is not supported yet. The method SSWrite writes a block of data: size is its size. Those methods call BasicATMSocket methods.

Last two methods call superclass methods.

```
public:
   ATMSocket(int32 socket=0);
   ~ATMSocket();

private:
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);

public:
   virtual bool Read(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
```

TCPSocket

[TCPSocket.h, TCPSocket.cpp]
The class TCPSocket adds the streamable capability to a BasicTCPSocket, i.e. a BasicTCPSocket can also be treat as a stream of data.

Like in the previous ATMSocket class methods SSRead and SSWrite are overridden to make operations on the BasicTCPSocket. Methods CanRead and CanWrite are also overridden and return true.

The class adds also a new method to the Streamable class, the method is called WaitConnection and return a TCPSocket*, it simply wait for a new TCP connection and return the object holding the connection.

```
public:
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool CanRead();
   virtual bool CanWrite();

   TCPSocket* WaitConnection(TimeoutType msecTimeout = TTInfiniteWait,TCPSocket *client = NULL);
```

UDPSocket

[UDPSocket.h, UDPSocket.cpp]
The class UDPSocket adds the streamable capability to a BasicUDPSocket, i.e. a BasicUDPSocket can also be treat as a stream of data. This is quite uncommon to UDP based communication that are usually studied as a connectionless socket media.

Like in the previous classese methods SSRead and SSWrite are overridden to make operations on the BasicUDPSocket. Methods CanRead and CanWrite are also overridden and return true.

```
public:
   UDPSocket(int32 socket=0);

   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
```

```
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool CanRead();
   virtual bool CanWrite();
```

Console

[Console.h, Console.cpp]
The class `Console` adds streaming capability to the class `BasicConsole`.

The first attribute, `streamBuffer`, is a small character array to buffer the console operations. On creation the constructor initialise che attribute `Streamble::csbIn` with the character array `streamBuffer` used as an buffered input.

```
private:
   char streamBuffer[32];
   FString attribute;
   bool attributeReadMode;
public:
   Console(ConsoleOpeningMode openingMode=ConsoleDefault,
      int numberOfColumns=-1,int numberOfRows=-1,TimeoutType msecTimeout=TTInfiniteWait);
   virtual ~Console();
```

Methods `CanWrite` and `CanRead` return both `true` because on a console you can either write and read. Methods `SSWrite` and `SSRead` supports different streams and are implemented to call `BasicConsole` methods or string related methods (`FString`).

Methods `Write` and `Read` call the superclass's method.

The first method `Switch` changes stream to write to, second method `Switch` is used to change the console attributes, valid names are "colour", two integers first is the fg second the bg; "cursor" the cursor position, two integers X,Y; "size" the buffer size, two integers DX,DY and also "window" the window size, two integers DX,DY. The method `NOfStreams` returns how many streams are available, and `SetSize` sets the size of the buffer.

```
   virtual bool CanWrite();
   virtual bool CanRead();
protected:
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
public:
   inline bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   inline bool Read(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);

   virtual bool Switch(uint32 newSelectedStream);
   virtual bool Switch(const char *name);
   virtual uint32 NOfStreams();
   inline bool SetSize(int numberOfColumns,int numberOfRows);
```

RemoteConsole

[RemoteConsole.h, RemoteConsole.cpp]
The class `RemoteConsole` simulates a `Console` for all purposes but allows a remote connection to take over. The `RemoteConsole` adds the possibility to connect a console via a TCP channel, inthis way we obtain a remote console; the remote connection is made available via a `TCPSocket` object hold as an attribute.

The `RemoteConsole` works with the aim of two threads: one that transmit and receive data on the TCP path and the other that handle local console commands (have a look at *level2/RemoteConsole.cpp*.

There are many attributes, most of them are semaphore or mutexes and are needed to keep synchronized the two threads.

```
private:
   int listeningPort;
   TID consoleThread;
   TID remoteConnectionThread;
   MutexSem resourceLock;
   EventSem newMessage;
   bool connected;
   TCPSocket* connectedSocket;
   char* bufferPtr;
   uint32 bufferReadSize;
   EventSem bufferRequest;
   bool hasToContinue;
```

The constructor initialize the object and bind the TCP socket listening on port 32769 (default) for incoming `RemoteConsole` connections, in this way an extarnal user can connect itself to the system simply connecting to that port on an host using a program like `telnet` and has a remote console to interact with BaseLib.

```
public:
   RemoteConsole(int listeningPort=32769);
   virtual ~RemoteConsole();

   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool CanWrite();
   virtual bool CanRead();

   virtual bool Switch(uint32 selectedStream);
   virtual bool Switch(const char* name);
   virtual uint32 NOfStreams();
```

File

`[File.h, File.cpp]`
The class `File` adds the streamable capability to a `BasicFile`, i.e. a `BasicFile` can also be treat as a stream of data.

Methods `SSRead` and `SSWrite` call `BasicFile`'s methods to read and write to the file; `Read` and `Write` are overridden but do the same as declared in the `Streamable` superclass, we can said the same for all other methods except `CanRead` and `CanWrite` that simply return `true` and the last method.

The method `StreamName` returns the file name or the name of any other substream we are using.

```
public:
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);

   inline bool Read(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   inline bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);

   virtual bool CanRead();
   virtual bool CanWrite();

   virtual int64 Size();
   virtual bool SetSize(int64 size);
   virtual bool Seek(int64 pos);
   virtual int64 Position(void);
   virtual bool CanSeek();

   virtual bool StreamName(uint32 n,char* name,int nameSize);
```

SXFILE

[SX_File.h]
Class SXFILE inherits only from Streamable (not also from BasicFile like class File). The class SXFILE is a *Streamable eXtension* for the standard C FILE routines.

Basically this class has as an attribute an integer that is nothing more than a file descriptor, on this file descriptor, that must be passed as an argument to the constructor, the class acts on standard C library function as fwrite, fread, ftell and fseek to implement a file streaming interface. The class must be completed with a new constructor that handle also file creation or opening.

```
private:
   int file;
public:
   virtual ~SXFILE();
   SXFILE(int f);

   bool SSRead(void* buffer, uint32 &size,TimeoutType msecTimeout);
   bool SSWrite(const void* buffer, uint32 &size,TimeoutType msecTimeout);
   bool CanRead();
   virtual bool CanWrite();

   int64 Size();
   bool SetSize(int64 size);
   bool  Seek(int64 pos);
   int64 Position(void);
   virtual bool CanSeek();
```

3.1.3 Design Notes

Some notes about constructors, if a constructor fail it must throw an exception, if doesn't throws an exception one can think that all its going well and after a while exeprience a bug. It is probably necessary to write some exception mechanism for example in BufferedStream if the argument is NULL then the object doesn't create itself and exit from the constructor. In class SXMemory the constructor doesn't check arguments.

There is a security hole in the RemoteConsole, is a TCP remote connection and doesn't check against password and autentication, use it with care because if the port is publically exposed some DOS attack can be experience.

3.2 ConfigurationDataBase

As we said in the previous chapters the ConfigurationDataBase is one of the key elements in the BaseLib software infrastructure. In Figure 3.3 is depicted the UML related to the classes in this level.

Basically there is only one class: CDBExtended. This class is an extension to the ConfigurationDataBase class; contains a reference to CDB of type CDBVirtual Forces call to CDB to be made via the virtual table. Sources, in this group, regarding level2 are:

- CDBExtended.h
- CDBDataTypes.h
- CDBDataTypes.cpp

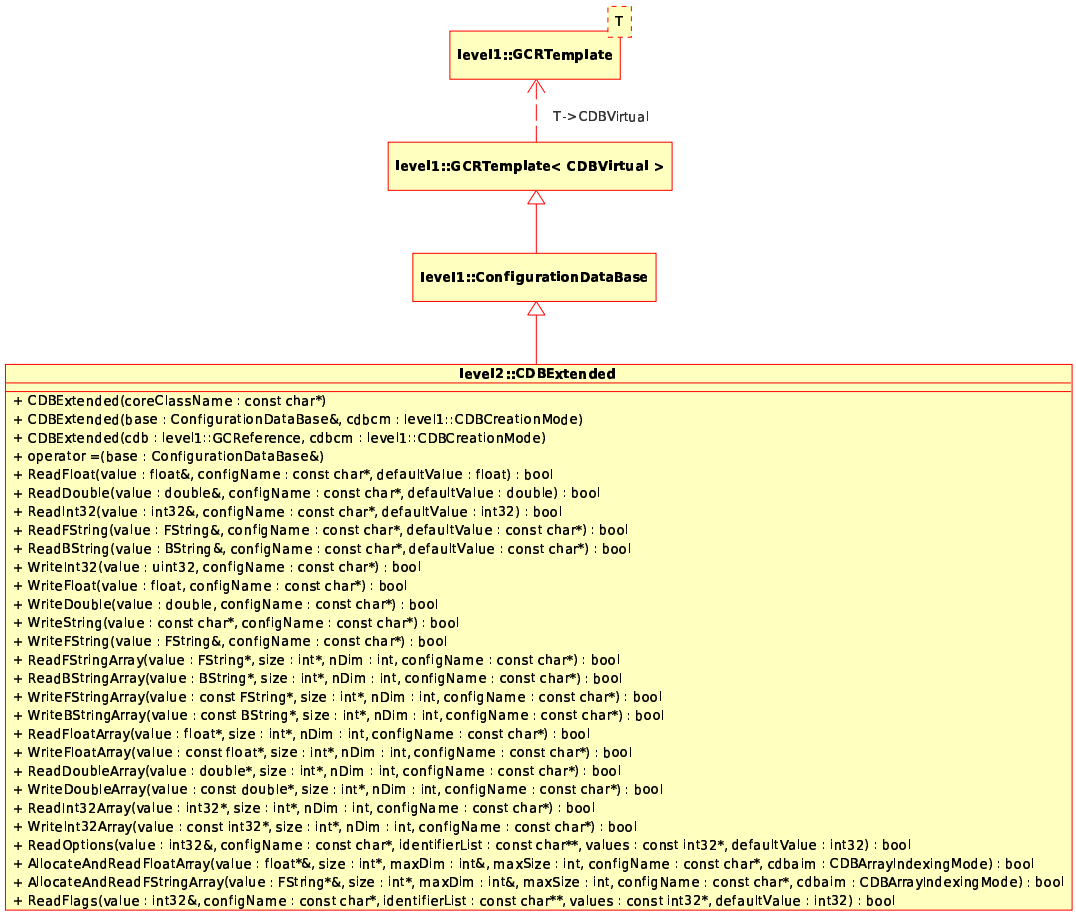

Figure 3.3: BaseLib Level2 CDB classes

In the following we analyse the `CDBExtended` class that will simply extends the `ConfigurationDataBase` with many specialized read and write methods. Such class is written as a collection of inline methods, also constructors are declared as `inline`.

The first constructor create a CDB from a generic base type, i.e. it will create a new `CDBNull` and than sets some value. The second constructor if `base` is specified it creates a new reference to an existing database, `cdbcm` attribute if is set to `CDBCM_CopyAddress` copies the address from the reference. The last constructor is similar to the second one. The copy operator of a CDB creates a new reference to an existing database.

```
public:
   inline CDBExtended(const char* coreClassName="CDB")
   inline CDBExtended(ConfigurationDataBase& base,CDBCreationMode cdbcm=CDBCM_CopyAddress);
   inline CDBExtended(GCReference cdb,CDBCreationMode cdbcm=CDBCM_CopyAddress);

   inline void operator=(ConfigurationDataBase& base);
```

Next methods address the read and write activity, read methods simply call `CDBVirtual::ReadArray` and write methods call `CDBVirtual::WriteArray`. All that with different arguments to read/write `Float`, `Double`, `Int32`, `FString`, `BString`, `FStringArray`, `BStringArray`, `FloatArray`, `DoubleArray` and `Int32Array`.

```
   inline bool ReadFloat(float &value,const char* configName,float defaultValue = 0);
   inline bool ReadDouble(double &value,const char* configName,double defaultValue = 0);
   inline bool ReadInt32(int32 &value,const char* configName,int32 defaultValue = 0);
   inline bool ReadFString(FString &value,const char* configName,const char *defaultValue = "");
```

```
inline bool ReadBString(BString &value,const char* configName,const char *defaultValue = "");
inline bool ReadFStringArray(FString* value,int* size,int nDim,const char* configName);
inline bool ReadBStringArray(BString* value,int* size,int nDim,const char* configName);
inline bool ReadFloatArray(float* value,int* size,int nDim,const char* configName);
inline bool ReadDoubleArray(double* value,int* size,int nDim,const char* configName);
inline bool ReadInt32Array(int32* value,int* size,int nDim,const char* configName);

inline bool WriteFloat(float value,const char* configName);
inline bool WriteDouble(double value,const char* configName);
inline bool WriteInt32(uint32 value,const char* configName);
inline bool WriteFString(FString &value,const char* configName);
inline bool WriteString(const char *value,const char* configName);
inline bool WriteFStringArray(const FString* value,int* size,int nDim,const char* configName);
inline bool WriteBStringArray(const BString* value,int* size,int nDim,const char* configName);
inline bool WriteFloatArray(const float* value,int* size,int nDim,const char* configName);
inline bool WriteDoubleArray(const double* value,int* size,int nDim,const char* configName);
inline bool WriteInt32Array(const int32* value,int* size,int nDim,const char* configName);
```

The method `ReadOptions` translates an identifier to a value, `dentifierList` is a zero terminated vector of `const char*`. The method `ReadFlags` translates a vector of identifiers to a bitset value, `identifierList` is a zero terminated vector of `const char*` the values associated to each identifier is in `values`.

The method `AllocateAndReadFloatArray` finds the matrix dimension first then allocates memory and finally read data; `size` parameter is a vector of dimensions of size `maxDim`; `value` argument will be allocated with at maximum `maxSize` floats, this routine uses malloc. The method `AllocateAndReadFStringArray` does exactly the same but handles `FString` objects; `value` argument will be allocated with at maximum `maxSize` `FString`, this routine used `new[]`.

```
inline bool ReadOptions(int32& value,
   const char* configName,
   const char** identifierList,
   const int32* values,
   int32 defaultValue);
inline bool ReadFlags(int32& value,
   const char* configName,
   const char** identifierList,
   const int32* values,
   int32 defaultValue);

inline bool AllocateAndReadFloatArray(float*& value,int* size,int& maxDim,int maxSize,
   const char* configName,CDBArrayIndexingMode cdbaim=CDBAIM_Strict);
inline bool AllocateAndReadFStringArray(FString*& value,int* size,int& maxDim,int maxSize,
   const char* configName,CDBArrayIndexingMode cdbaim=CDBAIM_Strict)
```

3.2.1 Design Notes

—

3.3 IO devices

In Level2 there are also two input device classes. Those input devices classes are: the serial line and the keyboard. In Figure 3.4 the UML schema is depicted.

We speak brefily about the `Keyboard` class than we take a tour of the serial line implementation that is much more interesting and better developed.

Keyboard

`[Keyboard.h]`
The `Keyboard` class manages the direct access to keyboard. The access is done throught a library call,

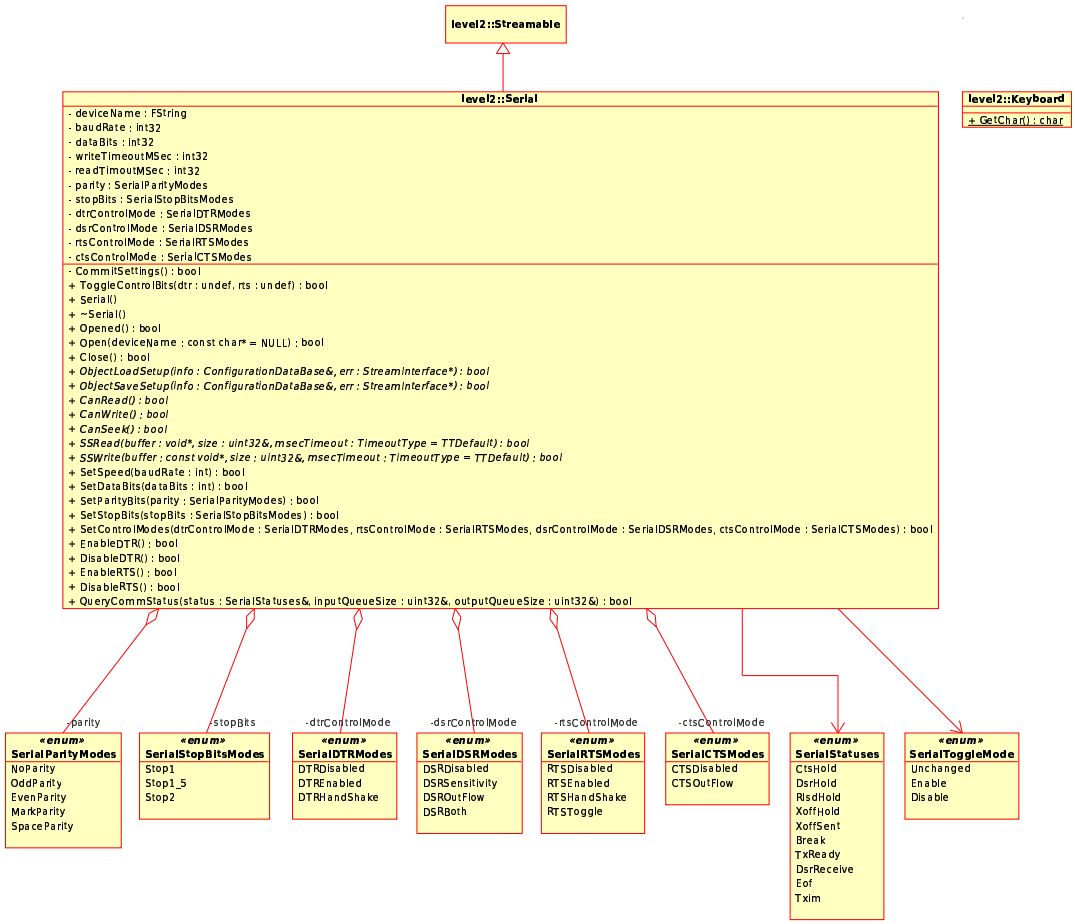

Figure 3.4: BaseLib Level2 Input devices classes

in most UNIX like system the call is `getchar()`, from the standard C library, in all other system `_getch` is used. The result is a read operation of a char without echoing. The straightforward implementation follow.

Note that this class doesn't interface directly with a driver but relay on the Operating System and C libraries.

```
class Keyboard{
public:
#if !(defined(_VXWORKS)||defined(_RTAI)|| defined(_LINUX) || defined(_SOLARIS))
    static  char GetChar() {return _getch();}
#else
    static char GetChar() {return getchar();}
#endif
};
```

Serial

`[Serial.h, Serial.cpp]`
The class `Serial` inherits from `Streamable`, i.e. the serial line can also be treated as a stream of data.

The class `Serial` in some OSes needs a file handle, attribute `port`, in the class is just defined for OS/2 and Microsoft Windows. The name of the device, the path in some systems, is hold by the `deviceName` attribute, all other attributes are serial line common properties and settings.

```
#if defined (_OS2)
```

```
    HFILE port;
#elif defined (_WIN32)
    HANDLE port;
#endif
   FString deviceName;
   int32 baudRate;
   int32 dataBits;
   int32 writeTimeoutMSec;
   int32 readTimeoutMSec;
   SerialParityModes parity;
   SerialStopBitsModes stopBits;
   SerialDTRModes dtrControlMode;
   SerialDSRModes dsrControlMode;
   SerialRTSModes rtsControlMode;
   SerialCTSModes ctsControlMode;
```

The method `CommitSettings` implements all the programmed changes, some methods only affects class's attributes and calling the method doesn't directly affect the serial line. `ToggleControlBits` changes the status of *dtr* and *rts* control lines in the serial interface, a value of `0` means untouched, `1` means to turn on and `2` means to turn off.

The method `Opened` query if the serial line device is just opened or not; `Open` open the device specified in the `name` argument, the name can be also a path in UNIX, in Microsoft Windows is something like `COM1`, if `name` argument is `NULL` than it will open what was setup using `ObjectLoadSetup`; `Close` closes the device.

The method `ObjectLoadSetup` accept from the configuration file the following parameters:

```
DeviceName COM1:
   BaudRate   int
   DataBits   5 6 7 8 9
   WriteTimeoutMSec   int
   ReadTimeoutMSec   int
   Parity   NoParity OddParity EvenParity MarkParity SpaceParity
   StopBits   Stop1 Stop1_5 Stop2
   RTSControlMode   RTSDisabled RTSEnabled RTSToggle
   DTRControlMode   DTRDisabled DTREnabled DTRHandShake
   DSRControlMode   DSRDisabled DSRSensitivity DSROutFlow DSRBoth
   CTSControlMode   CTSDisabled CTSOutFlow
```

The method `ObjectSaveSetup` is the standard object save function.

Methods`CanRead` and `CanWrite` return `true`; the method `CanSeek` is not available and return `false`. The method `SSRead` reads a block of data, `size` is the maximum size, on return `size` is what was read, the timeout is not supported yet. `SSWrite` writes a block of data, `size` is its size, on return `size` is what was written.

```
private:
   inline bool CommitSettings();
   inline bool ToggleControlBits(SerialToggleMode dtr, SerialToggleMode rts);

public:
   Serial();
   ~Serial();

   bool Opened();
   bool Open(const char *deviceName=NULL);
   bool Close();

   virtual bool ObjectLoadSetup(ConfigurationDataBase &info,StreamInterface *err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   virtual bool CanRead();
```

```
virtual bool CanWrite();
virtual bool CanSeek();

virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
virtual bool Write(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
```

The following function lets the user set the serial line parameters and to enable them. Last method `QueryCommStatus` retrieves the status of communication port.

```
bool SetSpeed(int baudRate);
bool SetDataBits(int dataBits);
bool SetParityBits(SerialParityModes parity);
bool SetStopBits(SerialStopBitsModes stopBits);

bool SetControlModes(
     SerialDTRModes dtrControlMode,
     SerialRTSModes rtsControlMode,
     SerialDSRModes dsrControlMode,
     SerialCTSModes ctsControlMode);

inline bool EnableDTR(void);
inline bool DisableDTR(void);
inline bool EnableRTS(void);
inline bool DisableRTS(void);

inline bool QueryCommStatus(SerialStatuses& status,
   uint32& inputQueueSize,
   uint32& outputQueueSize);
```

3.3.1 Design Notes

The class `Serial` is a `Streamable` object, but, is not likely to be a real stream of data, it is more an i/o device (why not an IOGAM..) or something architectural dependent, the same can be said for the `Keyboard` class.

The class `Keyboard` must be better developed without the need of the C standard library; an architectural dependent keyboard code can be developed to doesn't relay on the OS itself.

Chapter 4

BaseLib Level 3

BaseLib Level 3 deals about the problem of configuration that is really similar to serialization in Object Oriented Programming (OOP) in BaseLib. The basic issue addressed here is the reding and writing of a configuration file from a file or generally from a stream, the stream is a formatted document where it is saved a configuration of the system, a configuration must also be thinked as a backed up set of instances of all the classes in a particular life moment of the system (OOP serialization).

The main tool that lets the user select which class to load and setting the parameters is the *Configuration Database*, a *Configuration Database*, that was just partially covered in the previous chapters, is a tree data structure that reflect the content of a configuration file (or stream). Using this philosopy every component of the framework that need configuration parameters must ask to this central entity that address configuration issues.

The whole BaseLib Level 3 could be divided in the following sections:

- CDB
- other CDBs (stream oriented CDBs)
- Parser

The fundamental component of this level is the CDB, i.e. an implementation of a `Configuration Database` interface in */level1*. A CDB addresses the problem of which classes and which paramenters loading during the framework startup and reflect the system status (i.e. instantiated classes and attributes value) not only at the initialization but also during runtime; i.e. implements serialization.
Objects type registration and object instantiation is done by the previously analized GODB and ORDB; configuration database loading is the last step for the system inizialization.

The flow chart in Figure 4.1 shows the order in which the logical components that compose the objects infrastructure, i.e. type registration, instantiation and serialization; are loaded by the BaseLib framework at application startup.
Loading an application that runs under the BaseLib framework lets the code first execute the compile time C++ code that create static and globally defined classes; part of this classes create the ORDB. Then a configuration file (or a stream) will be read and a configuration tree will be created, i.e. a CDB. When the ORDB and the CDB are ready a GODB is then filled up using the configuration of a CDB loading objects registered in the ORDB. This is the whole story; we try to better explain all this stuff with an example in the following.

Example We now have a look to a simple example that make use of BaseLib and instantiate its objects via the CDB, the example is located in *BaseLibTests/TestMessageBroker.cpp*. The first thing

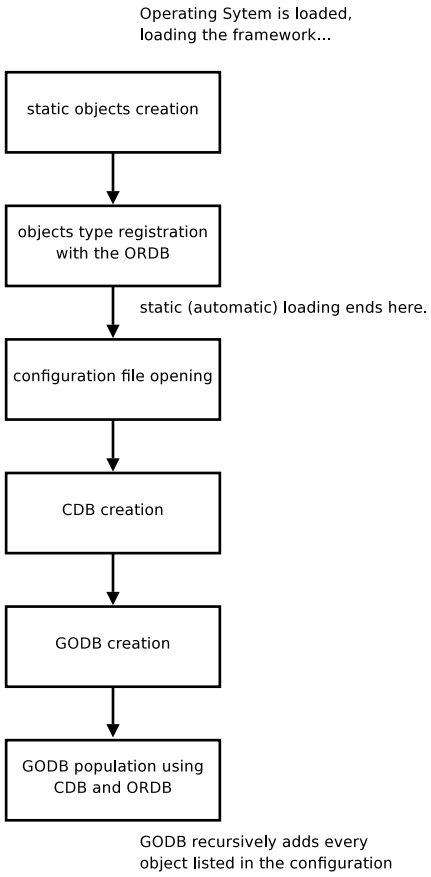

Figure 4.1: BaseLib objects loading and instantiation sequence

that follow is a snippets of a C++ source where there is the configuration data as a `char*` string. Don't try to understand what kind of objects are represented, they are no meaning in this place, have a look to the string format and try to understand the insights.

```
const char* testMessageBroker=
   "+TREES={"
   "␣␣␣␣Class␣=␣GCReferenceContainer\n"
   "␣␣␣+BRANCH1={\n"
   "␣␣␣␣␣␣␣␣Class␣=␣GCReferenceContainer\n"
   "␣␣␣␣␣␣␣+EP1={\n"
   "␣␣␣␣␣␣␣␣␣␣␣Class␣=␣MessageEndPoint\n"
   "␣␣␣␣␣␣␣}\n"
   "␣␣␣}\n"
   "␣␣␣+BRANCH2={\n"
   "␣␣␣␣␣␣␣␣Class␣=␣GCReferenceContainer\n"
   "␣␣␣␣␣␣␣+EP2={\n"
   "␣␣␣␣␣␣␣␣␣␣␣Class␣=␣MessageEndPoint\n"
   "␣␣␣␣␣␣␣}\n"
   "␣␣␣}\n"
   "␣␣␣+BRANCH3={\n"
   "␣␣␣␣␣␣␣␣Class␣=␣MessageBroker\n"
   "␣␣␣␣␣␣␣␣PeerIpAddress␣=␣localhost\n"
   "␣␣␣␣␣␣␣␣PeerPort␣=␣8882\n"
   "␣␣␣}\n"
   "}\n"
   "+MC={\n"
   "␣␣␣␣Class␣=␣MessageSenderTest\n"
   "}\n"
```

```
"+MS={\n"
"␣␣␣␣Class␣=␣MessageServer\n"
"␣␣␣␣ServerPort␣=␣8881\n"
"}\n";
```

Now we follow the process of the objects creation starting from the previous string configuration. All the process, in the source behind is depicted, using a UML diagram, in Figure 4.2. The basic steps are: creating a stream of data, in this case an `SXMemory` object, create a `ConfigurationDataBase` object, that is a `GRTemplate` templatized on any `CDBVirtual`, in this case a `CDB` that we will analize in this chapter and then the `cdb` is passed to the GODB for class instantiation.

```
SXMemory config((char*)testMessageBroker,strlen(testMessageBroker));
ConfigurationDataBase cdb;
if(!cdb->ReadFromStream(config,NULL)){
   CStaticAssertErrorCondition(ParametersError,"Init:␣cdb.ReadFromStream␣failed");
   return -1;
}

GCRTemplate<GCReferenceContainer> godb = GetGlobalObjectDataBase();
godb->ObjectLoadSetup(cdb,NULL);
```

Creating a automatic class `ConfigurationDataBase` as we can see in Figure 4.2, let an automatic chain of constructor take place that construct defaulty a ConfigurationDataBase of type `CDB` that is the first issue addressed in this chapter.

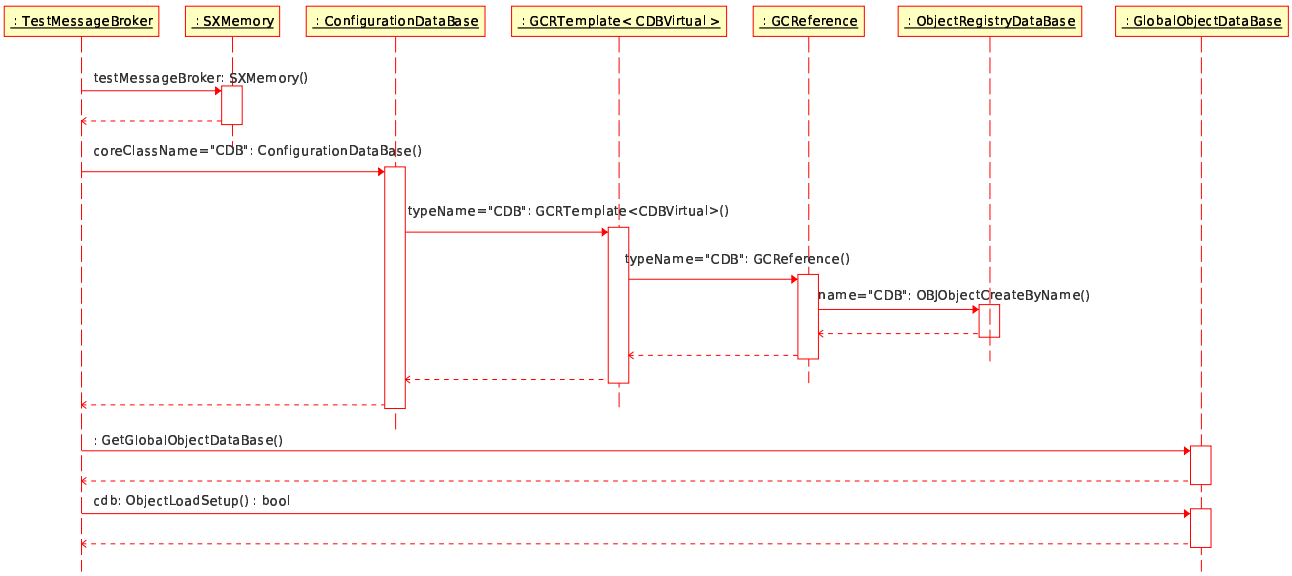

Figure 4.2: BaseLib TestMessageBroker example

We end this introductory paragraph with another example of a configuration file. Those lines can be cut and pasted in a single file ending in `.cfg`.

```
+HttpServer = {
   Class = HttpService
   Port = 8084
}
+Control = {
   Class = ControlGAM
   Controller = {
      NoPlasmaVelocityGain = 0.0
      NoPlasmaCurrentGain = 40.0
         IPWaveform = {
            Times = {0 120}
```

```
            Amplitudes = {0.5 0.5}
            Rounding = 50
         }
   }
}
+DAM = {
   Class = DAMGAM
   // Projection matrix
   ProjectionMatrix = {
      0 = { 1.0 0.0 0.0 0.0 }
      1 = { 0.0 1.0 0.0 0.0 }
      2 = { 0.0 0.0 1.0 0.0 }
      3 = { 0.0 0.0 0.0 1.0 }
   }
}
```

4.1 CDB

We have just spend some words about the `Configuration Database` that will be presented in this section, this is the first real implementation of a CDB that inherits from the class `CDBVirtual`. Figure 4.3 depict the UML diagram behind the classes in this section.

Class in this section are listed below.

- CDBNode
- CDBDataNode
- CDBStringDataNode
- CDBLinkNode
- CDBGroupNode
- CDBObjectNode
- CDBCore
- CDBNodeRef
- CDB
- CDBCInterface
- CDBTreePurger
- CDBDataTypeInterface

The CDB tree data structure is designed with two basic data structures: leafs and internal nodes. Each internal node has at least a leaf (or another internal node). An internal node is a `CDBroupNode` and a leaf is a `CDBDataNode`. Figure 4.4 depicts the structure described.

CDBNode

`[CDBNode.h, CDBNode.cpp]`
The class `CDBNode` is the basic node interface of the CDB tree data structure. Each type of node is build up the `CDBNode` class. In Figure 4.4 the tree data structure of a CDB is depicted, each class is a `CDBNode` or a derived class of a `CDBNode`, class `level0::LinkedListHolder` is necessary to hold a linked list of `CDBNode` objects (of type `LinkedListable`).

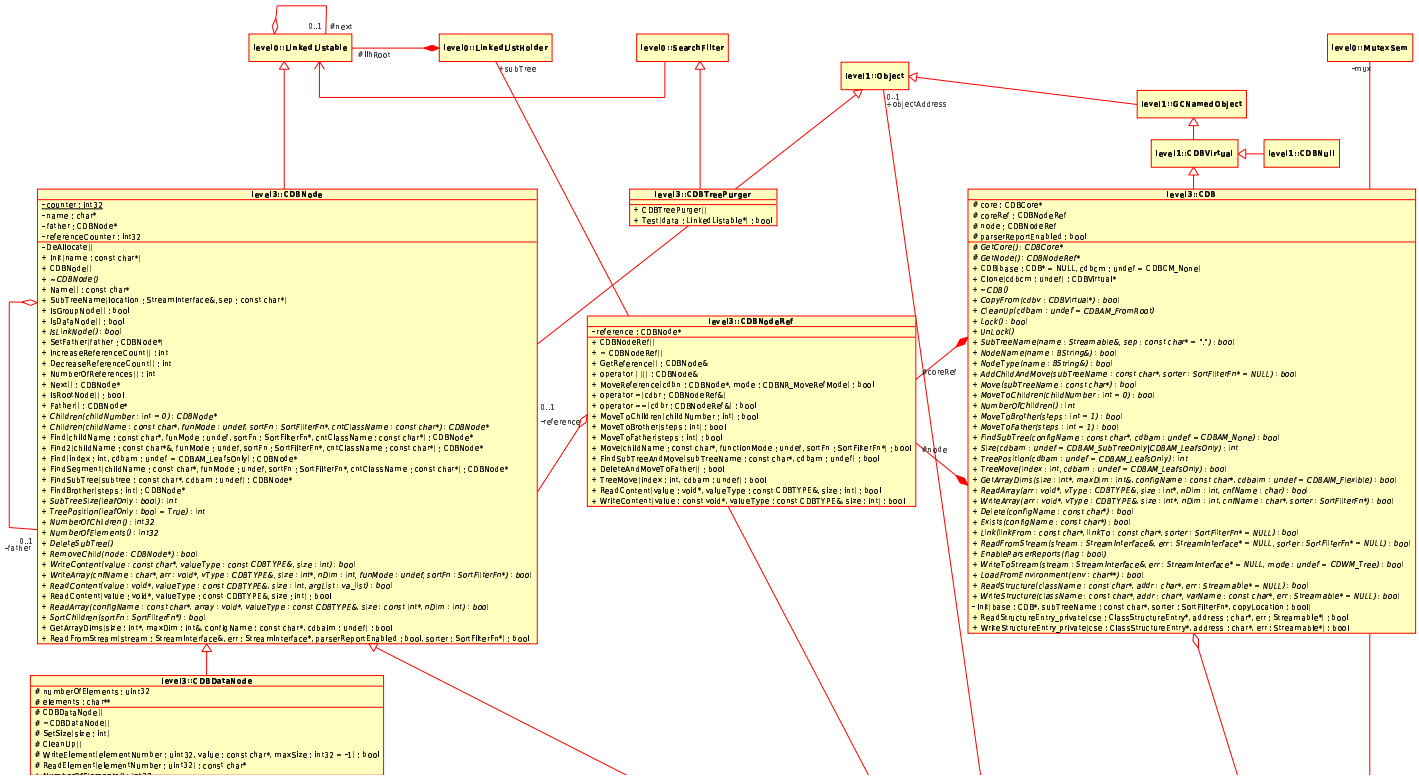

Figure 4.3: BaseLib Level3 CDB (a Configuration Database implementation)

We now switch to speak about attributes and methods of the `CDBNode` class. A `CDBNode` as the first attribute has a `static counter`, this variable lets hold the total number of nodes there are in the system, i.e. in the CDB, the constructor increment it and the deconstructor decrement it. Every node has a `name` that is really the name of the field, an object that it is its `father` in the tree and a `referenceCounter`, each node has its own `referenceCounter` for coherent destruction.

```
private:
    static int32 counter;

    char* name;
    CDBNode* father;
    int32 referenceCounter;
```

The first method we analysed is the private `DeAllocate` it deallocates memory used in a node (`name` allocated buffer); `Init` is public and inits the node and adds a name to it, then came the constructor and the destructor, that removes the node from its container also.

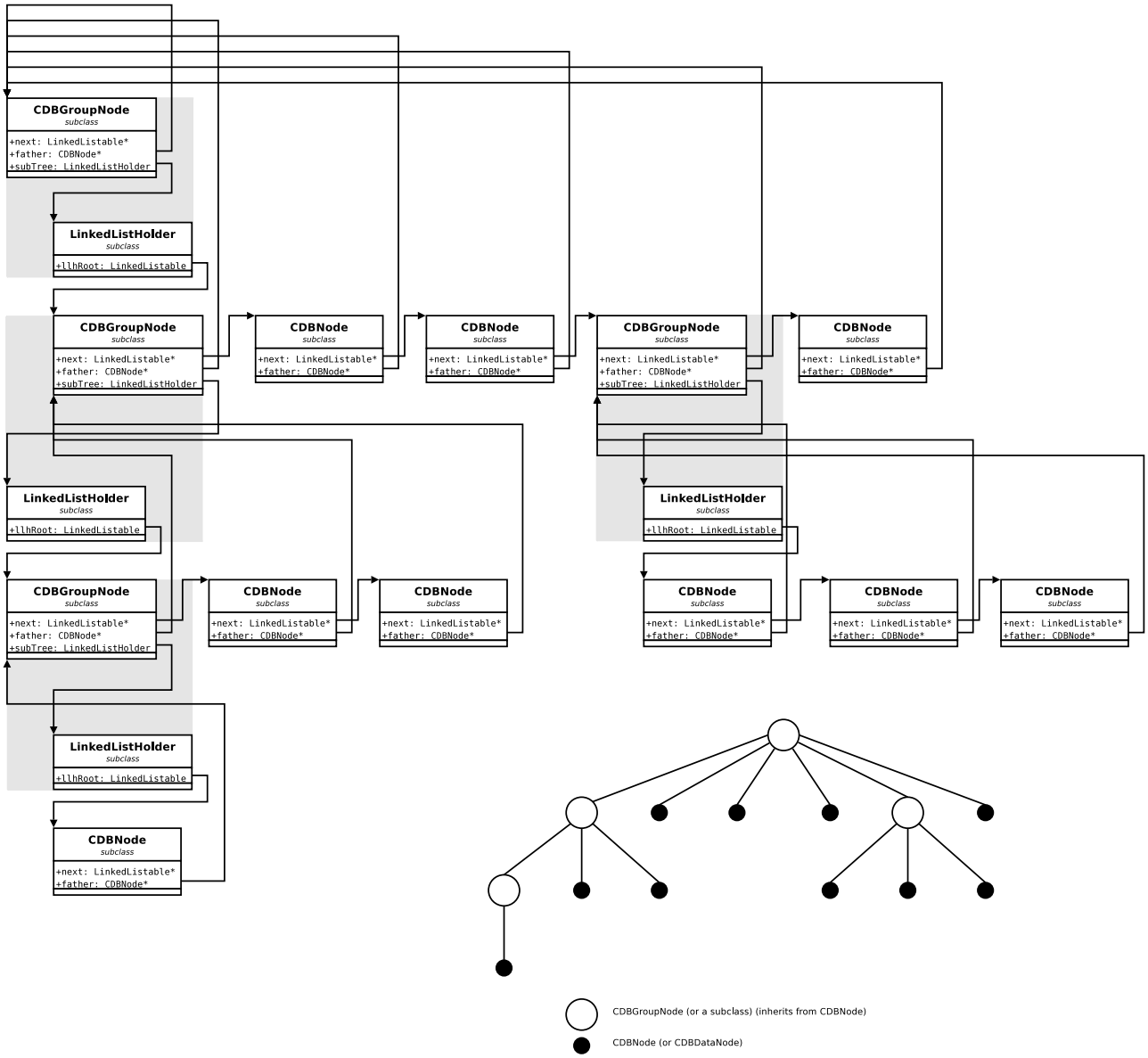

Figure 4.4: BaseLib Level3 CDB tree implementation

The method `Name` gets the `name` attribute of the class and `NumberOfElements` returns `-1` because such function must return how much data elements are contained by this node but a `CDBNode` is an abstraction of two node types so if is a leaf doesn't have elements but if it is an internal node it must have it. In this case it returns `-1` as an error, this function must be implemented at subclasses level.

The method `IsRootNode` check if the node is the root node of the tree, i.e. is a root node if the `father` attribute points to itself (is the same as the `this` pointer). The method `IsGroupNode` in this case return `false` bacause only the subclass `CDBGroupNode` and derived return `true`; method `IsDataNode` returns `false` because `CDBNode` is not a `CDBDataNode` or a subclass of it. The methods `IsLinkNode()` return `true` whether it is a link to a different subtree, in the case of a `CDBNode` simply return `false`.

Methods `IncreaseReferenceCount`, `DecreaseReferenceCount` and `NumberOfReferences` increment, decrement and return atomically the attribute `referenceCounter`, such attribute show the

number of users of this node. The method `Next` gets the next node, method `Father` gets the father of the node that you can sets via the method `SetFather`.

```
   void DeAllocate();
public:
   void Init(const char* name);

   CDBNode();
   virtual ~CDBNode();

   const char* Name() const;
   virtual int32 NumberOfElements() const;

   bool IsRootNode() const;
   inline bool IsGroupNode() const;
   inline bool IsDataNode() const;
   virtual bool IsLinkNode() const;

   int IncreaseReferenceCount();
   int DecreaseReferenceCount();
   int NumberOfReferences() const;

   CDBNode* Next() const;
   CDBNode* Father() const;
   void SetFather(CDBNode* father);
```

The first `Find` method finds a location within the whole (sub)tree as the object invoked on; remember that nodes are nombered from left to right and from subnode to supernode, if the node does not exist returns `false` and remains in the start position.

The second `Find` method simply call `Find2` method that has a sligtly different argument signature. `Find2` moves recursively in the tree from current position; `childName` can also be "UpNode" if you search is one level up, or "RootNode" if it is all the way up. Any movement onto unexisting subtree will fail; any movement out of unexisting subtree will erase temporary holders; movement is relative to current location. `childName` is returned modified to reflect the depth of matching.

The method `FindSegment` moves one level in the tree, we must remember that the CDB is a configuration tree that holds configuration data, as `Find2` `childName` argument if "UpNode" means one level up and if means "RootNode" is all the way up; as before any movement onto unexisting subtree will fail,any movement out of unexisting subtree will erase temporary holders; movement is relative to current location. The method `FindSubTree` moves to a node pointing to the specified subtree. `FindBrother` allows accessing brothers; negative and positive numbers allow to move relative to start position.

The method `SubTreeName` returns the name of the subtree starting from this node, this call is safe, does not require locking of tree. `SubTreeSize` returns `1` for a `CDBNode`, subclass of such class returns different values. `DeleteSubTree` will remove all unreferenced subtrees, in `CDBNode` doesn't take any action. `TreePosition` returns the position of the node within the tree.

The method `NumberOfChildren` returns how many childrens of this node, `-1` means it is not a node that has children, in this implementation `CDBNode` returns `-1`. The first `Children` allows accessing the subtrees and uptrees (using negative indexes -1 -2), `CDBNode` implements only access to uptrees. Second `Children` method allows accessing the subtrees by name, `childName` is the name of the node, it cannot be a full subtree; `functionMode` is of type `CDBNMode`, argument `sortFn` is used only if adding a node; this method is not implemented in `CDBNode` and simply return `NULL`. `SortChildren` apply sorting to the subtree; `RemoveChild` removes the specified child if exists.

```
   CDBNode* Find(int index,CDBAddressMode cdbam=CDBAM_LeafsOnly);
   inline CDBNode* Find(const char* childName,CDBNMode functionMode=CDBN_None,
      SortFilterFn* sortFn=NULL,const char* containerClassName=NULL);
```

```
CDBNode* Find2(const char*& childName,CDBNMode functionMode=CDBN_None,
   SortFilterFn* sortFn=NULL,const char* containerClassName=NULL);
CDBNode* FindSegment(const char* childName,CDBNMode functionMode=CDBN_None,
   SortFilterFn* sortFn=NULL,const char* containerClassName=NULL);
CDBNode* FindSubTree(const char* subtree,CDBAddressMode cdbam);
CDBNode* FindBrother(int steps);

void SubTreeName(Streamable& location, const char* sep);
virtual int SubTreeSize(bool leafOnly);
virtual void DeleteSubTree();
virtual int TreePosition(bool leafOnly = True);

virtual int32 NumberOfChildren() const;
virtual CDBNode* Children(int childNumber=0);
virtual CDBNode* Children(const char* childName,CDBNMode functionMode=CDBN_SearchOnly,
   SortFilterFn* sortFn=NULL,const char* containerClassName=NULL);
virtual bool SortChildren(SortFilterFn* sortFn);
virtual bool RemoveChild(CDBNode* node);
```

Methods `WriteContent`, `WriteArray`, `ReadContent` and `ReadArray` are all implemented in subclasses and in the class `CDBNode` simply return `false`. Read and write methods are implemented in `CDBDataNode` and subclasses.

The method `GetArrayDim` reads the dimensions of a matrix, it expects all the indexes to be found form 0 to N. `ReadFromStream` parses the stream and build a configuration database according to the simple CDB syntax. Those methods are implemented in `CDBNode`.

```
virtual bool WriteContent(const void* value,const CDBTYPE& valueType,int size);
virtual bool WriteArray(const char* configName,const void* array,
   const CDBTYPE& valueType,const int* size,int nDim,
   CDBNMode functionMode=CDBN_CreateStringNode,SortFilterFn* sortFn=NULL);

virtual bool ReadContent(void* value,const CDBTYPE& valueType,
   int size,va_list argList);
bool ReadContent(void* value,const CDBTYPE& valueType,int size,...);
virtual bool ReadArray(const char* configName,void* array,
   const CDBTYPE& valueType,const int* size,int nDim);

bool GetArrayDims(int* size,int& maxDim,const char* configName,
   CDBArrayIndexingMode cdbaim);
bool ReadFromStream(StreamInterface& stream,StreamInterface* err,
   bool parserReportEnabled,SortFilterFn* sorter);
```

CDBDataNode

[CDBStringDataNode.h, CDBStringDataNode.cpp]
The class `CDBDataNode` is the low level abstraction of all nodes containing data. It basically adds few methods and ttributes to the `CDBNode` class.

The first attribute `numberOfElements` can be 1 if the other attribute, `elements`, points to a string directly; greater then 1 if `elements` is a pointer to an array of pointers. As we said `elements` is a vector of pointers or a direct pointer to data if `numberOfElements` is 1, if `numberOfElements` is 0 `elements` should be `NULL`.

Constructor set `elements` to `NULL` and `numberOfElements` to 0; deconstructor call the method `CleanUp`.

The method `SetSize` resize data structure, i.e. it will call malloc on the attribute `elements` and copies the old data. `CleanUp` frees each elements and turn the data structure to 0 elemnts like the constructor.

The method `WriteElement` adds a new element in the `elements` array, such function handle also allocation of the memory, with the `ReadElement` method is also possible to read the entry just written.

The method `NumberOfElements` simply return the attribute `NumberOfElements`, i.e. how much data elements are contained by this node, -1 means it is not a node that has data. `HasData` return `true` whether it is to be considered a leaf node.

```
protected:
   uint32 numberOfElements;
   char** elements;

   CDBDataNode();
   ~CDBDataNode();

   void SetSize(int size);
   void CleanUp();
   bool WriteElement(uint32 elementNumber,const char* value,int32 maxSize=-1);
   const char* ReadElement(uint32 elementNumber) const;

   virtual int32 NumberOfElements() const;
   virtual bool HasData() const;
```

CDBStringDataNode

[CDBStringDataNode.h, CDBStringDataNode.cpp]
The class `CDBStringDataNode` is a csubclass of a `CDBDataNode` that holds data in string format, it is possible to convert from a string to each other type. Obviously if a string is a character array of letters can't be converted in a double. The class provides no attributes.

Following methods write (it makes a copy) of respectively double, float, integer, unsigned integer, char and strings from the first argument to the `CDBDataNode::elements` pointer. Each method accept in input not only one value but an array of data with the same format. Read methods have the same behaviour but copy the `elements` pointer to the `values` argument. All the write methods make use of the superclass method's `CDBDataNode::WriteElement` and write methods of `CDBDataNode::ReadElement`.

```
protected:
   bool WriteDouble(const double* values,uint32 size);
   bool WriteFloat(const float* values,uint32 size);
   bool WriteInteger(Const int32* values,uint32 size);
   bool WriteUnsignedInteger(const uint32* values,uint32 size);
   bool WriteChar(const char* values,uint32 size);
   bool WriteString(const char** values,uint32 size);
   bool WriteString(const FString* values,uint32 size);
   bool WriteString(const BString* values,uint32 size);

   bool ReadDouble(double* values,uint32 size);
   bool ReadFloat(float* values,uint32 size);
   bool ReadInteger(int32* values,uint32 size);
   bool ReadUnsignedInteger(uint32* values,uint32 size);
   bool ReadChar(char* values,uint32 size);
   bool ReadString(char** values,uint32 size);
   bool ReadString(FString* values,uint32 size);
   bool ReadString(BString* values,uint32 size);
```

The method `WriteFormatted` takes a formatted string as an input and write it in the texttttCDB-DataNode::elements attribute; probably the formatted input can contain data of different type. The method `ReadFormatted` read outputting on the `Streamable` argument all the elements in the `CDBDataNode` array, inserting around the formatted stream.

```
   bool WriteFormatted(const char* formatted,const char* seps="␣{}\n\t\r");
   bool ReadFormatted(Streamable* formatted,uint32 indentChars,uint32 maxElements);
```

Now comes public methods. The method `WriteContent` writes data on a node the value can be of any `CDBTYPE` the `size` argument specifies how many elements. The class `CDBTYPE` is declared in *level1*. Table shows registered `CDBTYPE`s, from *level1/CDBTypes.h*.

name	CDBDataType	sizeof()
CDBTYPE_float	CDB_float	float
CDBTYPE_double	CDB_double	double
CDBTYPE_int32	CDB_int32	int32
CDBTYPE_uint32	CDB_uint32	uint32
CDBTYPE_char	CDB_char	char
CDBTYPE_hex	CDB_hex	uint32
CDBTYPE_octal	CDB_octal	uint32
CDBTYPE_String	CDB_String	char*
CDBTYPE_BString	CDB_BString	BString
CDBTYPE_NULL	CDB_None	NULL

Table 4.1: Level1 declared `CDBTYPE`s in *level1/CDBTypes.h*

`ReadContent` reads data from a node, `valueType` specifies data type, `size` specifies how many elements and `value` contains a pointer to memory where to write the data.

The method `WriteArray` writes content (one to many elements) into a data node, creates a data node or modifies an existing, `configName` is the address of the parameter relative to the current node, `array` is the data in whatever form specified by `valueType` and `size` is a vector of matrix dimensions; if `size` is `NULL` it treats the input as a monodimensional array of size `nDim`; `sortFn` allows inserting newly created nodes in a specific order. The method `ReadArray` reads content from a data node to the `configName` pointer argument.

```
public:
   CDBStringDataNode(const char* name="");
   ~CDBStringDataNode();

   virtual bool WriteContent(const void* value,const CDBTYPE& valueType,int size);
   virtual bool ReadContent(void* value,const CDBTYPE& valueType,int size,va_list argList);

   virtual bool WriteArray(const char* configName,const void* array,
      const CDBTYPE& valueType,const int* size,int nDim,
      CDBNMode functionMode=CDBN_CreateStringNode,SortFilterFn* sortFn=NULL);
   virtual bool ReadArray(const char* configName,void* array,
      const CDBTYPE& valueType,const int* size,int nDim);
```

CDBLinkNode

`[CDBLinkNode.h, CDBLinkNode.cpp]`
Such `CDBLinkNode` should implements hard link between nodes. The constructor calls the superclass constructor, this class is a subclass of a `CDBStringDataNode`.

Methods `IsLinkNode` and `IsDataNode` return `true` each. `Children` allows accessing the subtrees by name but in this case simply return `NULL`; if argument `followLink` is `true` links are followed. `ReadContent` reads data from a node, this is the only real method implemented, it should find what to read in the `value` argument and then treat the same argument as an output stream. Not really well implemented.

```
public:
   CDBLinkNode(const char *name);
   virtual bool IsLinkNode() const;
```

```
virtual bool IsDataNode() const;
virtual CDBNode* Children(const char* childName,bool followLink=False);
virtual bool ReadContent(void* value,const CDBTYPE& valueType,int size,va_list argList);
```

CDBGroupNode

[CDBGroupNode.h, CDBGroupNode.cpp]
A CDB tree is constituited of two basic types of nodes: `CDBDataNode` and `CDBGroupNode` respectively leafs and internal nodes. Internal nodes are `CDBGroupNode` and basically are containers of `CDBNode`s, i.e. the branches of the tree. So the only, but the most important, attribute of this class is `subTree` a `LinkedListHolder` that holds all the nodes underneath in the tree.

```
public:
   LinkedListHolder subTree;
```

The constructor invokes the superclass's constructor with the passed by name; destructor call `LinkedListHolder::Reset` method. `NumberOfChildren` returns the number of elements contained in the linked list of the attribute, i.e. the number of children of this node. The first `Children` method allows accessing the subtrees and uptrees (with negative values, -1 -2...),the links are not expanded by this function; second `Children` method allows accessing the subtrees by name, `childName` is the name of the node, it cannot be a full subtree; `followLink` if `true` links are followed, `sortFn` used only if adding a node. The method `RemoveChild` removes a child by name; `DeleteSubTree` removes all unreferenced subtrees; `SortChildren` apply sorting. `SubTreeSize` returns how many elements in the sub tree, note that is far different from `NumberOfChildren` because counts elements in all the sub tree from the node.

```
public:
   CDBGroupNode(const char* name="");
   virtual ~CDBGroupNode();

   virtual int32 NumberOfChildren() const;
   virtual CDBNode* Children(int childNumber);
   virtual CDBNode* Children(const char* childName,
      CDBNMode functionMode=CDBN_SearchOnly,SortFilterFn* sortFn=NULL,
      const char* containerClassName=NULL);
   virtual bool RemoveChild(CDBNode* child);
   virtual void DeleteSubTree();
   virtual bool SortChildren(SortFilterFn* sortFn);
   virtual int SubTreeSize(bool leafOnly);
```

Methods that follow are just be analysed before and are not treated here. All those three methods, `ReadContent`, `WriteArray` and `ReadArray` are implemented here.

```
   virtual bool ReadContent(void* value,const CDBTYPE& valueType,
      int size,va_list argList);

   virtual bool WriteArray(const char* configName,const void* array,
      const CDBTYPE& valueType,const int* size,int nDim,ù
      CDBNMode functionMode=CDBN_CreateStringNode,SortFilterFn* sortFn=NULL);
   virtual bool ReadArray(const char* configName,void* array,
      const CDBTYPE& valueType,const int* size,int nDim);
```

CDBObjectNode

[CDBObjectNode.h, CDBObjectNode.cpp]
From the sources it is possible to read that a `CDBObjectNode` is "a container of CDBNodes and of an Object"; most important is that the `CDBObjectNode` class inherits from `CDBGroupNode` and has as attributes a `const char*` and a `Object*` so it has an `Object` associated with it. The constructor

register the name of the node, the pointer to the object and also the class type as a string.

The method `ReadArray` reads content from a data node, `configName` is the address of the parameter relative to the current node, `array` is the data in wahtever form specified by `valueType`, `size` is a vector of matrix dimensions, if `size` is `NULL` it treats the input as a monodimensional array of size `nDim`; `nDim` specifies how many dimensions the array possesses or the vector size if `size` is `NULL`. `ReadContent` reads also data from a node argument `value` contains a pointer to memory where to write the data.

```
public:
   const char* classType;
   Object* objectAddress;

   CDBObjectNode(const char* name,Object* object,const char* type);
   virtual ~CDBObjectNode();

   virtual bool ReadArray(const char* configName,void* array,
      const CDBTYPE& valueType,const int* size,int nDim);
   virtual bool ReadContent(void* value,const CDBTYPE& valueType,
      int size,va_list argList);
```

CDBCore

`[CDBCore.h]`
Probably the class `CDBCore` is the most simplest class of *level3*, it only use a `level0:MutexSem` guaranteeing shared access to a CDB context safe. The class code follow.

```
class CDBCore: public CDBGroupNode{
private:
   MutexSem mux;
public:
   CDBCore(){ mux.Create();}
   virtual ~CDBCore(){ }
   virtual bool Lock(){
      if (!mux.Lock()){
         CStaticAssertErrorCondition(FatalError,"CDBCore␣Lock␣failed!");
         return False;
      }
      return True;
   }
   virtual void UnLock(){
      mux.UnLock();
   }
};
```

CDBNodeRef

`[CDBNodeRef.h, CDBNodeRef.cpp]`
The class `CDBNodeRef` is a reference to a `CDBNode` that remaps some methods. The only attribute is infact a `CDBNode*`. The class is a kind of iterator to move between nodes.

The constructor takes no arguments and initialize the object to a dummy `CDBNode` reference. The distructor move the reference away. There is also a copy operator overridden and a compare operator redefinition.

```
   CDBNode* reference;
public:
   CDBNodeRef();
   ~CDBNodeRef();

   inline CDBNode &GetReference();
   inline CDBNode &operator();
```

```
inline void operator=(CDBNodeRef &cdbr);
inline bool operator==(CDBNodeRef &cdbr);
```

The method `MoveReference` changes the reference to the node, if argument `cdbn` is `NULL` than it will refer to the globally allocated `nullCDBNode`.

The method `MoveToChildren` allows accessing the subtrees (argument `childNumber>=0`) and uptree (-1, -2, -3, ...); `MoveToBrother` allows accessing the brothers and `MoveToFather` allows acceesing brothers.

The method `Move` is the general method that allows accessing the subtrees and uptrees, like in `CDBNode` "UpNode" is one level up and "RootNode" is all the way up, any movement onto unexisting subtree will fail; any movement out of unexisting subtree will erase temporary holders; finally movements are relative to current location.

The method `FindSubTreeAndMove` moves to a node pointing to the specified subtree; `DeleteAndMoveToFather` remove current node and move up, the operation will complete only if the number of users is 0, the up movement will happen anyway. `TreeMove` moves to a location within the whole (sub)tree.

The method `ReadContent` reads data on a node, `valueType` specifies data type and `size` specifies how many elements; `WriteContent` writes data on a node argument `valueType` specifies data type and `size` specifies how many elements.

```
bool MoveReference( CDBNode *cdbn, CDBNR_MoveRefMode mode);
inline bool MoveToChildren(int childNumber=0);
inline bool MoveToBrother(int steps = 1);
inline bool MoveToFather(int steps = 1);
inline bool Move(const char* childName,CDBNMode functionMode=CDBN_None,
   SortFilterFn* sortFn=NULL);

inline bool FindSubTreeAndMove(const char* subTreeName,
   CDBAddressMode cdbam=CDBAM_None);
inline bool DeleteAndMoveToFather();
inline bool TreeMove(int index,CDBAddressMode cdbam=CDBAM_LeafsOnly);

bool ReadContent(void* value,const CDBTYPE& valueType,int size,...);
inline bool WriteContent(const void* value,const CDBTYPE& valueType,int size);
```

CDB

`[CDB.h, CDB.cpp]`
Like a `CDBNull` a `CDB` class extends a `CDBVirtual`. A `CDB` is an implementation of a hierarchical database. It works similarly to a filesystem having directory nodes (internal nodes or branches) and file nodes (leaves).

Each leaf node (`CDBDataNode`) is a container of a sequence of characters, both ASCII and not ASCII. `CDB` is infact a reference to an instance of the database. Copying `CDB`s does not duplicate the database but increases a reference counter. Operation to the same database by different threads is safe.

We start with the attributes as usual. The first attribute is a `CDBCore*` that is used to acquire concurrent access to a `CDB`, the first `CDBNodeRef` is a pointer to the root node of the *Configuration DataBase*, second `CDBNodeRef` is a pointer to the current node, obviously it can point to branches or leaves. Last argument `parserReportEnabled` said whether parsing reports faults.

The method `GetCore` basically return the `core` attribute and `GetNode` return a reference to the `node` attribute.

```
protected:
   CDBCore* core;
   CDBNodeRef coreRef;
   CDBNodeRef node;
```

```
   bool parserReportEnabled;

   virtual CDBCore* GetCore();
   virtual CDBNodeRef* GetNode();
```

The first method is the constructor of a CDB, if `base` is specified it creates a new reference to an existing database; if `NULL` creates a new database; argument `cdbcm` must be `CDBCM_CopyAddress` to copy the address from the reference; in case of two or more flags, after oring one must cast back to the enum type. `Clone` simply call the constructor with `this` as `base` argument creating a new reference to a database, or if that is not possible it creates a copy. Then comes the destructor that does nothing, but will destroy database if instance count goes to zero. The method `CopyFrom` copy all the `CDB` passed by to the current `CDB`. `CleanUp` removes all the content from the database unless some other database instance exists and points to a subtree one can request to delete only the current subtree.

The method `Lock` locks the main database for exclusive access: use if a group of transactions should be atomic, `UnLock` unlocks the main database: remember unlocking a locked database after use.

The method `SubTreeName` finds the overall path leading to the current node, `NodeName` returns the name of the current node and `NodeType` gets the type of the current node.

The method `AddChildAndMove` can be used to create a new subtree, `Move` moves the `node` attribute to the specified location, the movement is relative to the current location. `MoveToChildren` moves the `node` attribute, negative number move up, zero is the first of the children; `NumberOfChildren` returns how many branches there are from this node; negative number implies that the location is a leaf. The method `MoveToBrother` moves to a brother's node, with 0 means remain where you are, greater then 0 brothers on the right below 0 on the left. `MoveToFather` moves up, `-1` means `Root`, one level up for each positive.

The method `FindSubTree` search a node identified by `configName` attribtue from node search on the right of the tree for the subtree identified by the string name, on success `node` attribute points to the node containing the subtree or leaf, such method will not follow links. `Size` return the total number of nodes, param `cdbam` if `CDBAM_SubTreeOnly` allows to measure the current subtree and if is `CDBAM_LeafsOnly` allows counting only the data nodes. The method `TreePosition` return the absolute location of a node in the tree in left to right bottom to top order. `TreeMove` moves to a location within the whole (sub)tree; nodes are numbered from left to right and from subnode to supernode; if the node does not exist returns `False` and remains in the start position.

```
public:
   CDB(CDB* base=NULL,CDBCreationMode cdbcm=CDBCM_None);
   CDBVirtual* Clone(CDBCreationMode cdbcm);
   virtual ~CDB();
   virtual bool CopyFrom(CDBVirtual* cdbv);
   virtual void CleanUp(CDBAddressMode cdbam=CDBAM_FromRoot);

   virtual bool Lock();
   virtual void UnLock();

   virtual bool SubTreeName(Streamable& name,const char* sep=".");
   virtual bool NodeName(BString& name);
   virtual bool NodeType(BString& name);

   virtual bool AddChildAndMove(const char* subTreeName,SortFilterFn* sorter=NULL);
   virtual bool Move(const char* subTreeName);
   virtual bool MoveToChildren(int childNumber=0);
   virtual int NumberOfChildren();
   virtual bool MoveToBrother(int steps=1);
   virtual bool MoveToFather(int steps=1);
```

```
virtual bool FindSubTree(const char* configName,CDBAddressMode cdbam=CDBAM_None);
virtual int Size(CDBAddressMode cdbam=CDBAM_SubTreeOnly|CDBAM_LeafsOnly);
virtual int TreePosition(CDBAddressMode cdbam=CDBAM_LeafsOnly);
virtual bool TreeMove(int index,CDBAddressMode cdbam=CDBAM_LeafsOnly);
```

The method `GetArrayDims` is inherited from `CDBVirtual`, if argument `cdbaim` is `CDBAIM_Rigid`, it expects all the indexes to be found form 0 to $n-1$ and the same number in all subtrees. Also methods `ReadArray` and `WriteArray` are from `CDBVirtual`.

The method `Delete` removea a leaf or a subtree (note that position is relative), if a leaf is removed then the pointer is moved to the father; if a specified node's children are deleted, but not the node, the pointer still points at the node; to delete a link use the `linkTo` as the leaf name; to delete a subtree simply specify the group node. `Exist` tells whether a certain entry exists. Finally method `Link` lets you make a link, `linkFrom` is the path where the link is created from.

```
virtual bool GetArrayDims(int* size,int& maxDim,const char* configName,
   CDBArrayIndexingMode cdbaim=CDBAIM_Flexible);
virtual bool ReadArray (void* array,const CDBTYPE& valueType,const int* size,
   int nDim,const char* configName);
virtual bool WriteArray(const void* array,const CDBTYPE& valueType,
   const int* size,int nDim,const char* configName,SortFilterFn* sorter=NULL);

virtual bool Delete(const char* configName);
virtual bool Exists(const char* configName);
virtual bool Link(const char* linkFrom,const char* linkTo,
   SortFilterFn* sorter=NULL);
```

Finally we analysed some complex load/save functions. All those functions override `CDBVirtual`'s methods.

The method `ReadFromStream` reads a database from a stream, `WriteToStream` writes the database to stream without any ordering.

The method `EnableParserReports` enables reports of parser during `ReadFromStream` into error; `LoadFromEnvironment` loads from environment or from any `NULL` terminated list of chars.

`ReadStructure` reads from CDB to memory and `WriteStructure` either copies or references a memory structure, at address `address` and of type `className` `CDB` transforms the memory, `MMCDB` just references it.

```
virtual bool ReadFromStream(StreamInterface& stream,StreamInterface* err=NULL,
   SortFilterFn* sorter=NULL);
virtual bool WriteToStream(StreamInterface& stream,StreamInterface* err=NULL,
   CDBWriteMode mode=CDBWM_Tree);

virtual void EnableParserReports(bool flag);
virtual bool LoadFromEnvironment(char** env);

virtual bool ReadStructure (const char* className,char* address,
   Streamable* err=NULL);
virtual bool WriteStructure(const char* className,char* address,
   const char* variableName=NULL,Streamable* err=NULL);

private:
   void Init(CDB* base=NULL,const char* subTreeName=NULL,SortFilterFn* sorter=NULL,
      bool copyLocation=False);
   bool ReadStructureEntry_private(ClassStructureEntry* cse,char* address,
      Streamable* err);
   bool WriteStructureEntry_private(ClassStructureEntry* cse,char* address,
      Streamable* err);
```

CDBCInterface

[CDBCInterface.h, CDBCInterface.cpp]
The CDBCInterface is not a class but a set of C functions. These functions enable an user to access a CDB within C language, not all CDB's methods are available.

The code start with a type redefinition that follow.

```
typedef void* CDBReference;
```

The method CDBCICreate creates a reference to a CDB, if argument cdbr is NULL just creates, otherwise it copies; CDBCIDestroy destroy a CDB reference, CDBCIClean clean the current subtree, it returns 1 on succes and 0 on failure.

CDBCILoad loads from a memory buffer into the current subtree, it returns 1 on success 0 on failure. CDBCISave saves from the current subtree into a memory buffer, it returns 1 on success and 0 on failure, buffer is a zero terminated string, it must be freed using CDBCIFree. The method CDBCIMove moves to a specific node, returns 1 on success 0 on failure; argument command is a dot separated list of commands, each command can be:

- a node name to move to
- root -> move to root
- father -> move to father
- brothernn -> $nn > 0$ move to the brothers on the right < 0 on the left
- childnn -> $nn >= 0$ move to the *nnth* child

if location is not NULL then the current location is returned, location is a zero terminated string. It must be freed using CDBCIFree.

The method CDBCIReadArray reads a value in the specified format, float, double and int data are in the C standard order; char* data is stored as a single array separated by 0s; arraySizes is a 0 terminated array of ints and it lists the array dimensions; arguments arrayValue and arraySizes are allocated by the library and **must** be deallocated with CDBCIFree. The method CDBCIWriteArray writes a value in the raw format, again float, double and int data are in the C standard order; char* data is stored as a single array separated by 0s and arraySizes is a 0 terminated array of ints.

The method CDBCIExist checks the existance of a node, given the node name returns 1 if the node exists, 0 otherwise. CDBCINumberOfChildren returns the number of children of the present node, a negative number is returned if the cdbr is invalid or the present node is a leaf. The method CDBCINodeName returns the name of the node, partial or full depending on the value of the command parameter; if the full parameter is 1 the whole path will be returned, otherwise the node name will be returned; the pointer to the string will be allocated by the function and must be freed by the user; if the cdbr reference is invalid a NULL pointer will be returned.

Finally the method CDBCIFree is used to deallocate any of the allocated data.

```
CDBReference CDBCICreate(CDBReference cdbr);
void CDBCIDestroy(CDBReference* cdbr);
int CDBCIClean(CDBReference cdbr);

int CDBCILoad(CDBReference cdbr,const char* buffer);
int CDBCISave(CDBReference cdbr,char** buffer);
int CDBCIMove(CDBReference cdbr,const char* command,char** location);

int CDBCIReadArray(CDBReference cdbr,CDBCDataType type,void** arrayValues,int** arraySizes);
int CDBCIWriteArray(CDBReference cdbr,const char* entryName,CDBCDataType type,void* arrayValues,int*
```

```
int CDBCIExist(CDBReference cdbr,const char* entryName);
int CDBCINumberOfChildren(CDBReference cdbr);
char* CDBCINodeName(CDBReference cdbr,int full);

void CDBCIFree(void* data);
```

CDBTreePurger

[CDBGroupNode.h, CDBGroupNode.cpp]
The class `CDBTreePurger` is a filter to cleanup a tree from empty branches. The class inherits from `level0::SearchFilter` and is pasted below for its semplicity and understanding.

```
class CDBTreePurger: public SearchFilter {
public:
   CDBTreePurger(){ }
   bool Test(LinkedListable* data){
      CDBNode* cdbn = (CDBNode*)data;
      cdbn->DeleteSubTree();
      return ((cdbn->NumberOfReferences()<=0)&&(cdbn->NumberOfChildren()<=0));
    }
};
```

CDBDataTypeInterface

[CDBDataTypeInterface.h]
The class `CDBDataTypeInterface` is an interface, infact it has only pure virtual methods and no attributes. The interface does not inherits from any other class. As the time of writing there is no other class in BaseLib that inherits from this interface.

The class describes how to handle a certain data type in the database. Allows storing a certain type of data in CDB. The method `ElementBySize` should return the size of an element of the managed class as bytes. `ManagedClass` (?) returns the type of object that is managed.

The method `EncodeSubTree` converts the data in a tree of classes which is allocated with `new`; argument `sizes` are the sizes of each dimension, `dataVector` is a packed array of `ManagedClass` (?) whose size is described by the following parameters: `numberOfDimensions` the number of dimensions, 0 means a scalar 1 is a vector 2 is matrix, finally `sizes` is an array of ints of size `numberOfDimensions`, the order of data is that used in C.

The method `DeCodeSubTree` tries to convert a node or a subtree into the data type requested, returns the array size; if `dataVector` is `NULL` then it will be allocated using `new[]`, if it is not `NULL` then the result will be written in the memory pointed by it, numberOfDimensions if as input if `dataVector` is not `NULL` is the size of `dataVector`, as an output it contains the number of dimensions in the data; will be allocated using malloc.

```
public:
   virtual int ElementByteSize()=0;
   virtual const char* ManagedClass()=0;

   virtual CDBNode* EncodeSubTree(const void* dataVector,int numberOfDimensions,
      const int*& sizes)=0;
   virtual bool DeCodeSubTree(CDBNode* node,void*& dataVector,int& numberOfDimensions,
      int* sizes)=0;
```

Remarks

TODO
TODO
TODO
esempio d'uso

Design Notes

Software design practise was not applied to this data structure. `CDBNode` declare many methods most of those are only implemented by subclasses. OOP doesn't teach to declare all the possible methods in the superclass but to define them in the subclass that specialized them.

The behaviour of a `CDBLinkNode` is not known, probably has never been tested yet; for sure is not a soft link, i.e. in the source code somewhere is written that it is an hard link. A `CDBLinkNode` is a `CDBStringDataNode` without more attributes with only one method overridden (`ReadContent`) the constructor call `CDBNode`'s `Init` method.

4.2 Other CDB implementations (CDBOS, SCDB)

In the following we analysed two flavours of *Configuration Database*, `CDBOS` and `SCDB` that rispectively extends `CDBVirtual` and make use of a `CDBExtended`. So the new elements of the BaseLib we going to analysed are:

- CDBOS
- SCDB

In Figure 4.5 those new elements are depicted in a UML schema, better UMLs are coming in the following sections making a good picture of the machanisms involved in this section.

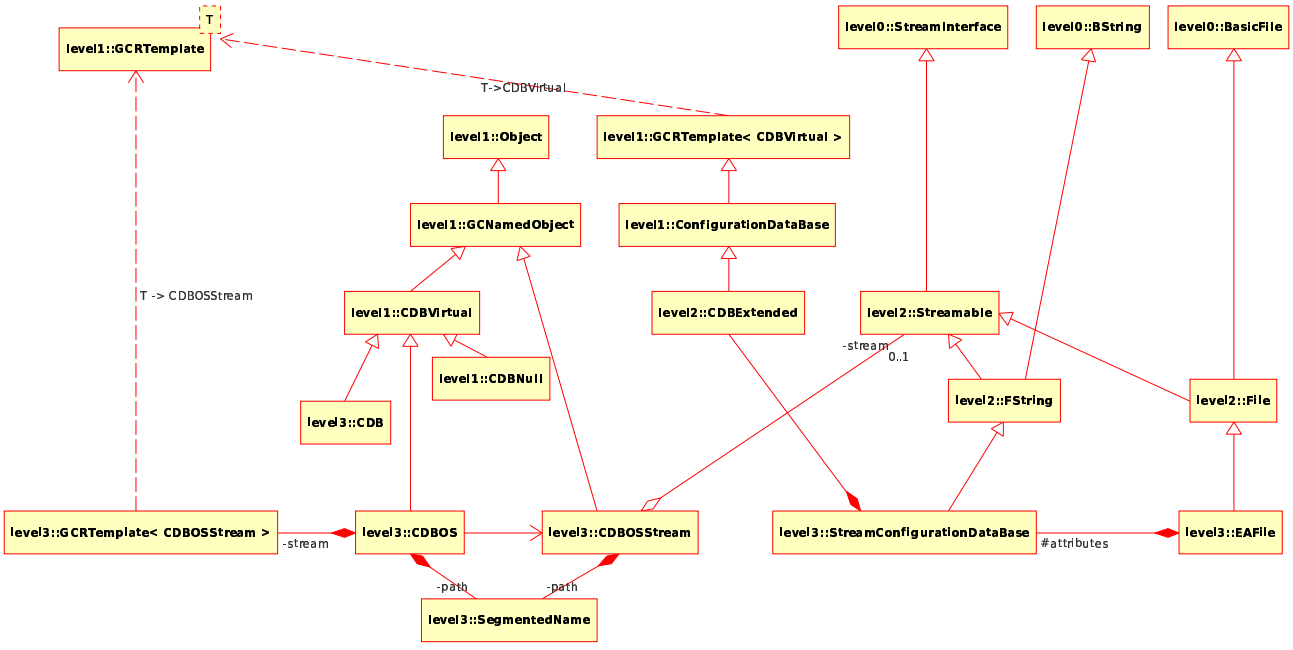

Figure 4.5: BaseLib Level3 CDBOS and SCDB

4.2.1 CDBOS

TODO
TODO
TODO
CHE COSA E e cosa fa

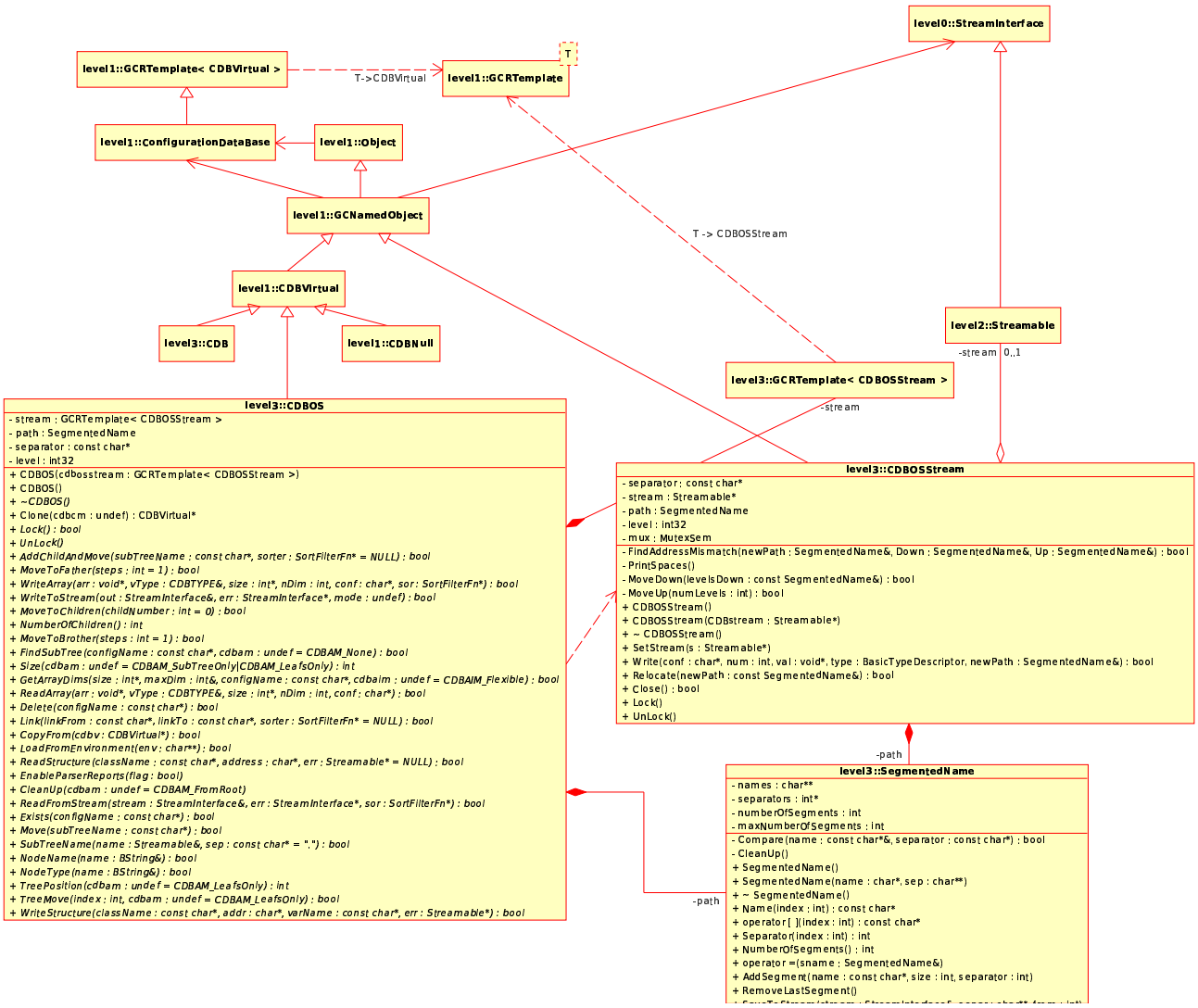

Figure 4.6: BaseLib Level3 CDBOS (another Configuration Database implementation)

- SegmentedName
- CDBOSStream
- CDBOS

SegmentedName

`[SegmentedName.h]`
The class `SegmentedName` doesn't inherit from any other class. Such class parses a name with segments like paths or structure naming. For example it can be used to segment a UNIX path like this:

`/usr/include/asm-generic/bitops/atomic.h`

in two data structures: an array of character strings, the segments (attribute `names` see below the source), and an array of index of separators (attribute `separators`). The parsing of the path above load the class with the following contents:

index	`char** names`	`int* separators`
0	usr	/
1	include	/
2	asm-generic	/
3	bitops	/
4	atomic	/
5	h	.

Table 4.2: Level3 `SegmentedName` class attributes

From the source we know the four attribtues on which the class is build up, the first attribute, `names` is a zero separated names array, `separators` is an index in the separators list provided at object construction (note that in the attribute list there is no such list). `numberOfSegments` is the current number of segments added to the list, `maxNumberOfSegments` is the actually max number of segments in the class.

```
private:
   char** names;
   int* separators;
   int numberOfSegments;
   int maxNumberOfSegments;
```

The method `Compare` is similar to the well known `strcmp` but limit search to size of separators, `CleanUp` simply free all associated structures (char strings) of a class.

Teh first constructor create a new `null SegmenetedName`, the second one take as arguments a character string containing a name to be segmented and a separators list, i.e. a `NULL` terminated vector of separators strings.

The method `Name` returns the `index`-th segment, `Separator` returns the index of the `index`-th separator in the string. `NumberOfSegments` returns the currently available segments in the class. Then two operator ridefinition are coming. `AddSegment` simply let you to add a segment manually, that you can remove with `RemoveLastSegment`. With the method `SaveToStream` it is possible to recompone the segmented name saved starting from any segment (see `fromSegment` argument).

```
   bool Compare(const char*& name,const char* separator);
   void CleanUp();

public:
   SegmentedName();
   SegmentedName(const char* name,const char** separatorNames);
   ~SegmentedName();

   const char* Name(int index) const;
   int Separator(int index);
   int NumberOfSegments() const;
   const char* operator[](int index) const;
   void operator=(SegmentedName& sname);

   void AddSegment(const char* name,int size,int separator);
   void RemoveLastSegment();
   void SaveToStream(StreamInterface& stream,const char** separatorNames=NULL,
      int fromSegment=0);
```

CDBOSStream

[CDBOSStream.h]
The class CDBOSStream is used from the class CDBOS, so it is presented before.

The first attribute is a character string separator that holds a list of separator, single character separators. The second attribute is a pointer to a Streamable, i.e. an interal pointer to the Streamable to write the CDB content. The attribute path is a SegmentedName that holds the current path in the CDB, level stores the current level in the hierarchy and the MutexSem protects against concurrent operations on the data structure.

```
private:
   const char* separator;
   Streamable* stream;
   SegmentedName path;
   int32 level;
   MutexSem mux;
```

We take a look at the methods of the class; method FindAddressMismatch finds differences between two CDB addresses; PrintSpaces prints spaces to respect hierarchy, MoveDown moves **n** levels down the structure and MoveUp moves **n** levels up the structure.

The constructor initialize the class by default with separator "/". The method SetStream sets the stream attribute; Write writes to the output one value at the current address. Relocate lets move in the CDB and update internal path; Close closes the parenthesis opened untill now; methods Lock and UnLock locks and unlock the stream's semaphore.

```
   bool FindAddressMismatch(const SegmentedName& newPath,int& levelsUp,
      SegmentedName& levelsDown);
   void PrintSpaces();
   bool MoveDown(const SegmentedName& levelsDown);
   bool MoveUp int numLevels);

public:
   CDBOSStream(): separator("/"), path("",&separator);
   CDBOSStream(Streamable* CDBstream) : separator("/"),path("",&separator);
   ~CDBOSStream();

   void SetStream(Streamable* s);
   bool Write (const char* configName,int numOfElements,const void* value,
      BasicTypeDescriptor type,const SegmentedName& newPath);

   bool Relocate (const SegmentedName& newPath);
   bool Close();
   void Lock();
   void UnLock();
```

The class CDBOSStream is used to templatize a GCRTemplate that is then used as an attribute in the following CDBOS class.

CDBOS

[CDBOS.h, CDBOS.cpp]
The class CDBOS inherits from CDBVirtual, the CDBOS implement direct writing of a *Configuration Database* to a stream. The first attribute is a GCRTemplate< CDBOSStream >, i.e. an object that has a pointer to a CDBOSStream. The attribute path save the current path inside the CDB, separator is a character array of separators and level stores the current level in the hierarchy in the same way as they do in the CDBOSStream class.

```
private:
```

```
   GCRTemplate<CDBOSStream> stream;
   SegmentedName path;
   const char* separator;
   int32 level;

   CDBOS(GCRTemplate<CDBOSStream> cdbosstream) : separator("/"), path("",&separator);
   CDBOS() : stream(GCFT_Create), separator("/"), path("",&separator);
   virtual ~CDBOS();
```

The method `Clone` returns a `CDBVirtual` pointer creating a new reference to a database, or if that is not possible it creates a copy of it; if `cdbcm` is `CDBCM_CopyAddress` ensures that the new object points at the same location.

The method `Lock` locks the main database for exclusive access: use if a group of transactions should be atomic; `UnLock` unlocks the main database: remember to unlock a locked database after using it.

The method `AddChilAndMove` can be used to create a new subtree, `MoveToFather` move the current pointer to the father's node, -1 means moving to the root node.

The method `WriteArray` calls the method `GCRTemplate< CDBOSStream >::Write`, if `size` is `NULL` it treats the input as a monodimensional array of size `nDim`. The method `WriteToStream` is used to write to the stream assigned.

```
public:
   CDBVirtual* Clone(CDBCreationMode cdbcm);
   virtual bool Lock();
   virtual void UnLock();

   virtual bool AddChildAndMove(const char* subTreeName,SortFilterFn* sorter=NULL);
   virtual bool MoveToFather(int steps=1);

   virtual bool WriteArray(const void* array,const CDBTYPE& valueType,const int* size,
      int nDim,const char* configName,SortFilterFn* sorter=NULL);
   virtual bool WriteToStream(StreamInterface& streamout,StreamInterface* err=NULL,
      CDBWriteMode mode=CDBWM_Tree);
```

Then follow a huge list of overridden methods that simply returns an error because are not implemented in this class.

```
    virtual bool MoveToChildren(int childNumber=0);
    virtual int  NumberOfChildren();
    virtual bool MoveToBrother(int steps=1);
    virtual bool FindSubTree(const char* configName,CDBAddressMode cdbam=CDBAM_None);
    virtual int  Size(CDBAddressMode cdbam=CDBAM_SubTreeOnly | CDBAM_LeafsOnly)
    virtual bool GetArrayDims(int* size,int& maxDim,const char* configName,CDBArrayIndexingMode cdba
    virtual bool ReadArray(void* array,const CDBTYPE& valueType,const int* size,int nDim,const char*
    virtual bool Delete(const char* configName);
    virtual bool Link(const char* linkFrom,const char* linkTo,SortFilterFn* sorter=NULL);
    virtual bool CopyFrom(CDBVirtual* cdbv);
    virtual bool LoadFromEnvironment(char** env);
    virtual bool ReadStructure (const char* className,char* address,Streamable* err=NULL);
    virtual void EnableParserReports(bool flag);
    virtual void CleanUp(CDBAddressMode cdbam=CDBAM_FromRoot);
    virtual bool ReadFromStream(StreamInterface& stream,StreamInterface* err=NULL,SortFilterFn* sort
    virtual bool Exists(const char* configName);
    virtual bool Move(const char* subTreeName);
    virtual bool SubTreeName(Streamable& name,const char* sep = ".")
    virtual bool NodeName(BString& name);
    virtual bool NodeType(BString& name);
    virtual int  TreePosition(CDBAddressMode cdbam=CDBAM_LeafsOnly);
    virtual bool TreeMove(int index,CDBAddressMode cdbam=CDBAM_LeafsOnly);
    virtual bool WriteStructure(const char* className,char* address,const char* variableName=NULL,St
```

4.2.2 Stream Configuration Database (SCDB)

The *Stream Configuration Database* is another kind of *Configuration Database* that is really similar to the `CDBOS` structure. Figure 4.7 represent the UML schema of the *Stream Configuration Database* infrastructure. Classes in this section are:

- StreamConfigurationDataBase
- EAFile

The `EAFile` is grouped in this section because it is the only user of the SCDB in this section.

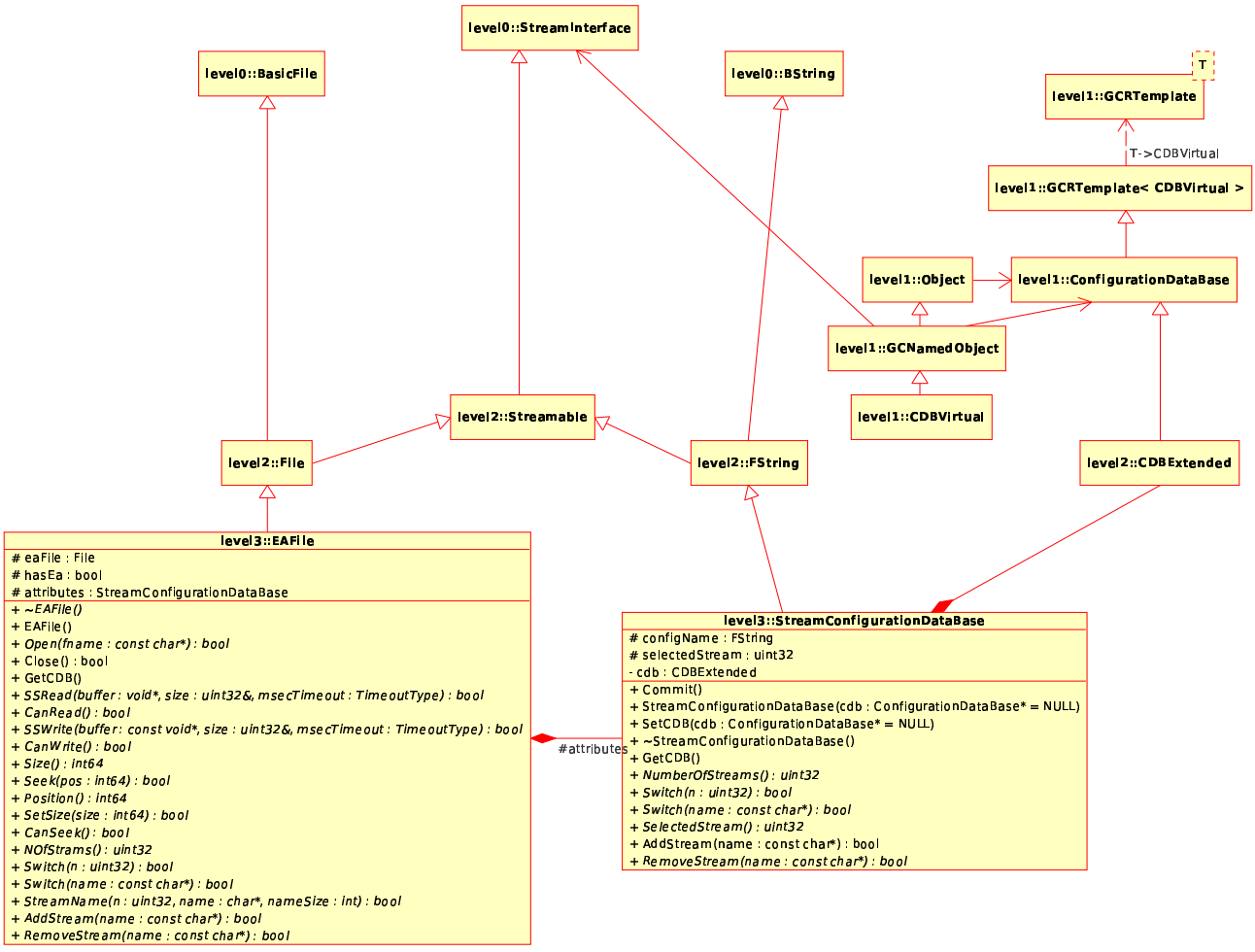

Figure 4.7: BaseLib Level3 SCDB (another Configuration Database implementation)

StreamConfigurationDataBase

`[StreamConfigurationDataBase.h, StreamConfigurationDataBase.cpp]`
The class `StreamConfigurationDataBase` implement a *Configuration Database* using the multi stream interface to read and write data into, so it is far different from the `CDBOS` because is a `Streamable`, more precisely a `FString`, that has a `CDBExtended` as an attribute. With a SCDB it is possible to write and read to a CDB like a stream.

The first attribute `cdb` is the `CDBExtended` associated to the SCDB, i.e. the CDB being modified; `configName` is the current location pointer and `selectedStream` is the order of current location within tree.

```
protected:
   CDBExtended cdb;
   FString configName;
   uint32 selectedStream;
```

The method `Commit` copies the work done to the stream this far to the database, the constructor initialises with a reference to a *CDB* the destructor basically calls `Commit` and `SetCDB` like the constructor initialises with a reference *CDB*; `GetCDB` gets the `cdb` attribute.

The method `NumberOfStreams` returns how many streams are available in total, counts from root all the leaves. `SelectedStream` returns which is the selected stream. Methods `Switch` select the stream to read from, switching may reset the stream to the start. The method `AddStream` adds a new stream to write to and `RemoveStream` removes an existing stream.

```
public:
   void Commit();

   StreamConfigurationDataBase(ConfigurationDataBase* cdb = NULL):cdb("CDB");
   ~StreamConfigurationDataBase();

   void SetCDB(ConfigurationDataBase* cdb = NULL);
   CDBExtended& GetCDB();

   virtual uint32 NumberOfStreams();
   virtual uint32 SelectedStream();
   virtual bool Switch(uint32 n);
   virtual bool Switch(const char *name);

   bool AddStream(const char *name);
   virtual bool RemoveStream(const char *name){
```

EAFile

`[EAFile.h, EAFile.cpp]`
The class `EAFile` stands for *Extended Attributes File*, it accounts to store extended attributes of files that a File Systema of an OS doesn't account for in a separate file; such files have the following extension, we paste the definition below.

```
#define EA_EXT ".!EA!"
```

The class inherits from `File` so its a `Streamable`. The first atrribute, `eaFile` is a `File` so it not only inherits from `File` but has also an attribute of the same type, and such attribute refers to the file containing the attributes. `hasEa` is `true` if any attributes have been set; `attributes` is a stream interface to the attributes.

```
protected:
   File eaFile;
   bool hasEa;
   StreamConfigurationDataBase attributes;
```

We skip destructor and constructor because are without arguments. The `Open` method opens for read/write or create if missing the *Extended Attribute File*, `Close` closes it. `GetCDB` get the `attribute` attribute content. All other methos are from the `Streamable` interface.

The method `SSRead` reads from the selected stream, `SSWrite` writes to the selected stream, methods `CanRead` and `CanWrite` return the capability of reading and writing of the `attribute`.

```
public:
   virtual ~EAFile();
   EAFile()
```

```
virtual bool Open(const char *fname,...);
bool Close();

CDBExtended& GetCDB();

virtual bool SSRead (void* buffer,uint32& size,TimeoutType msecTimeout);
virtual bool CanRead();
virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout);
virtual bool CanWrite();
```

The method `Size` returns the size of the file; `SetSize` clips the file size to a specified point; `Position` returns the current position in the file; `Seek` moves within the file to an absolute location, `CanSeek` return `true` if seeking is possible.

The method `NOfStreams` gets how many streams are available; methods `Switch` select the stream to read from, switching may reset the stream to the start. `StreamName` returns the name of the stream `n`, stream `0` is the file and therefore you get the file name. `AddStream` adds a new stream to write to, `selectedStream` is resetted to `0`; `RemoveStream` removes a stream.

```
virtual int64 Size();
virtual bool SetSize(int64 size);
virtual int64 Position(void);
virtual bool Seek(int64 pos);
virtual bool CanSeek();

virtual uint32 NOfStreams();
virtual bool Switch(uint32 n);
virtual bool Switch(const char* name);
virtual bool StreamName(uint32 n,char* name,int nameSize);
virtual bool AddStream(const char* name);
virtual bool RemoveStream(const char* name)
```

4.2.3 Design Notes

The class `SegmentedName` supports separators of only one character. How it is possible to handle separators made up of one or more characters, i.e. how it is possible to handle an URL like

`http://www.google.it`

4.3 Parser

The following classes address the problem of parsing a string, all classes create a framework to build up a parser. In Figure 4.8 is depicted the UML schema of the parser infrastructure. The following classes are involved:

- LA_TokenInfo
- LA_TokenData
- LexicalAnalyzer
- Parser

Lexical Analyzer Token Information

`[LexicalAnalyzer.h, LexicalAnalyzer.cpp]`
The class `LA_TokenInfo` stores information about a lexical element. The first attribute, `token` is the code identifying the lexical meaning of this part of the text, `description` holds the meaning of the

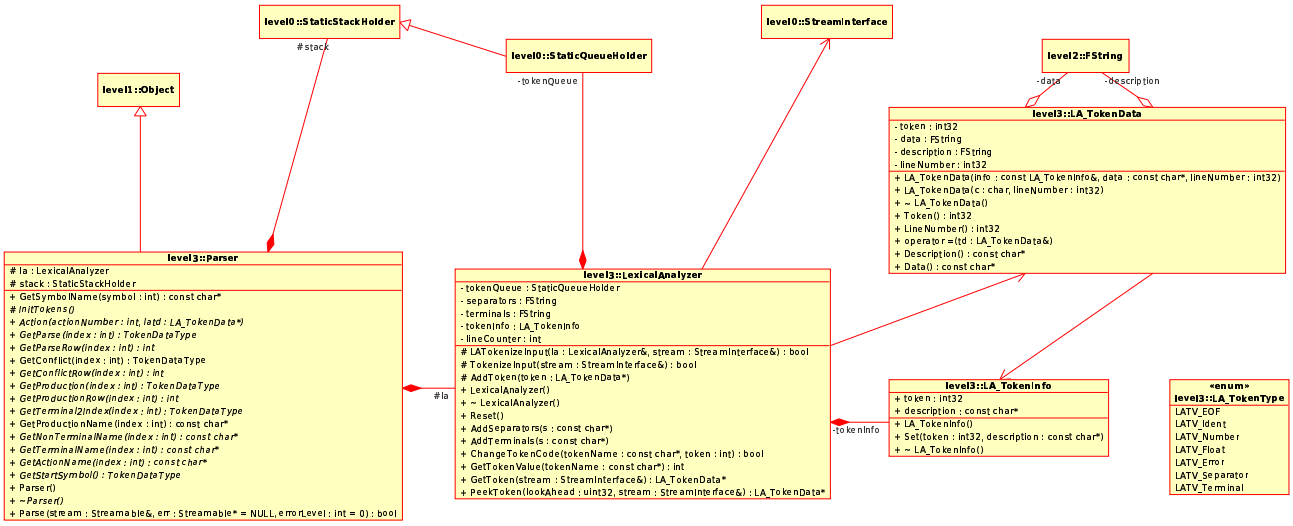

Figure 4.8: BaseLib Level3 parser, lexical analyser and tokenizer

token. The constructor initialize `token` to zero and `description` to `NULL`, the method `Set` lets you setting the attributes.

```
public:
   int32 token;
   const char* description;

   LA_TokenInfo();
   ~LA_TokenInfo();
   void Set(int32 token,const char* description);
```

Lexical Analyzer Token Data

[LexicalAnalyzer.h, LexicalAnalyzer.cpp]
The class `LA_TokenData` is an element produced by the `LexicalAnalyzer`. The attribute `token` is the code identifying the lexical meaning of this part of the text; `data` is the parsed part of the text; `description` is copied from a `LA_TokenInfo`, it holds the meaning of the token; `lineNumber` saves at what line the token was found.

```
   int32 token;
   FString data;
   FString description;
   int32 lineNumber;
```

The first constructor builds a token from a token description constant and the data content, the second constructor builds a token from a simple character.

The method `Token` returns the attribute `token`; `LineNumber` returns the attribute `lineNumber`; the `operator=` redefinition copies content, `Description` return the description of the content and `Data` return the content, i.e. `data` attribute.

```
public:
   LA_TokenData(const LA_TokenInfo &info, const char *data,int32 lineNumber);
   LA_TokenData(char c,int32 lineNumber);
   virtual ~LA_TokenData();

   int32 Token();
   int32 LineNumber();
   void  operator=(LA_TokenData& td);
   const char* Description();
   const char* Data();
```

Lexical Analyzer

[LexicalAnalyzer.h, LexicalAnalyzer.cpp]
The class `LexicalAnalyzer` is a slightly programmable lexical analyzer; It recognizes *identifiers*, *numbers* and *floats*. It allows to browse ahead without consuming the token, tokenisation is performed on demand.

Before exploring the class we paste an enumeration below. This enumeration is a set of codes to list the complex terminals recognised by the analyser. The source code is self explained.

```
enum LA_TokenType{
   /** indicates EOF */
   LATV_EOF=0x100,
   /** indicates an identifier or a string */
   LATV_Ident=0x101,
   /** indicates an integer */
   LATV_Number=0x102,
   /** indicates a float */
   LATV_Float=0x103,
   /** indicates a wrongly constructed element */
   LATV_Error=0x104,
   /** indicates an element that separates parts of the phrase */
   LATV_Separator=0x105,
   /** indicates an element that is a token on its own */
   LATV_Terminal=0x106
};
```

The first attribute, `tokenQueue` is a queue of pre-cooked tokens; `separators` is a string made of separators; `terminals` is a string composed of the terminal characters; `tokenInfo` list of token informations.

```
   StaticQueueHolder tokenQueue;
   FString separators;
   FString terminals;
   LA_TokenInfo tokenInfo[LATV_Error+2-0x100];
   int lineCounter;
```

The method `TokenizeInput` is the main method that takes as an argument a `StreamInterface` and with the `this` pointer calls a friend C method `LA_TokenizeInput` that make the tokenization of the input stream; every token is also added to the `tokenQueue`.

The method `AddToken` adds a token into the `tokenQueue`. `Reset` resets the status of the class; `AddSeparators` sets these characters as separators; `AddTerminals` sets these characters as separators; `ChangeTokenCode` changes the token code associated with a given complex terminal, valid `tokenNames` are:

- "EOF"
- "IDENT"
- "NUMBER"
- "FLOAT"
- "ERROR"

The method `GetToken` takes one token from the stack or processes the input for a new one, moves the token into last token. The class allocates the data but does not provide to the deallocation once the structure has been extracted. The method `PeekToken` reads in the stack at position `lookAhead` argument or increases the stack to allow for it; it returns the token but it still keeps hold of it.

```
protected:
   bool TokenizeInput(StreamInterface &stream);
   void AddToken(LA_TokenData* token);

public:
   LexicalAnalyzer(): tokenQueue(sizeof(LA_TokenData*)/sizeof(int));
   ~LexicalAnalyzer()

   void Reset();
   void AddSeparators(const char* s);
   void AddTerminals(const char* s);
   bool ChangeTokenCode(const char* tokenName,int token);

   int GetTokenValue(const char* tokenName);

   LA_TokenData* GetToken(StreamInterface& stream);
   LA_TokenData* PeekToken(uint32 lookAhead,StreamInterface& stream);
```

Parser

[Parser.h, Parser.cpp]
The class `Parser` allows implementation of parsers. It is a base class for parsers. The class inherits from `Object`. The first attribute, of type `LexicalAnalyzer`, is the lexical analyzer; second attribute is a `StaticStackHolder`.

The method `GetSymbolName` gets the symbol name from an index. The method `Parse` parses a `Streamable` passed by argument, `errorLevel` can be one of the following:

- `0` means do not display
- `1` means show actions
- `2` means show productions
- `-1` means show all

```
protected:
   LexicalAnalyzer la;
   StaticStackHolder stack;
private:
   const char* GetSymbolName(int symbol);
public:
   Parser():stack(sizeof(TokenDataType)/sizeof(int));
   virtual ~Parser();
   bool Parse(Streamable& stream,Streamable* err=NULL,int errorLevel=0);
```

Then follows a great set of pure virtual methods that must be implemented in `Parser`'s subclasses. Actually in BaseLib there is no class that make use or extends such class.

```
protected:
   virtual void InitTokens()=0;
   virtual void Action(int actionNumber,LA_TokenData* latd)=0;
   virtual TokenDataType GetParse(int index)=0;
   virtual int GetParseRow(int index)=0;
   virtual TokenDataType GetConflict(int index)=0;
   virtual int GetConflictRow(int index)=0;
   virtual TokenDataType GetProduction(int index)=0;
   virtual int GetProductionRow(int index)=0;
   virtual TokenDataType GetTerminal2Index(int index)=0;
   virtual const char* GetProductionName(int index)=0;
   virtual const char* GetNonTerminalName(int index)=0;
```

```
virtual const char* GetTerminalName(int index)=0;
virtual const char* GetActionName(int index)=0;
virtual TokenDataType GetStartSymbol()=0;
```

4.3.1 Design Notes

Never used in BaseLib.

Chapter 5

BaseLib Level 4

BaseLib Level 4 integrate full HTTP protocol, server and client side. The server can manage more than one client at a time and the stream is usually coded in HTML version 1.0 with support to the *keep alive* functionality. There is also an approach to manage secure connection letting the developer which pages are for public domain and which are only for selected users defining a sort of policy security.

5.1 HttpStream and URLs

Figure 5.1 depicts an UML schema of the classes involved in this section. Classes in this section are:

- URL
- HttpURL
- HttpStream
- HtmlStream

HttpStream

`[HttpStream.h, HttpStream.cpp]`
The class `HttpStream` implements an HTTP language multiple stream. The main stream *read* and *write* channels, operate with the client (or on a cache until method `CompleteReadOperation` is not called); the other streams allow read/write access to special information like `HttpOptions` or HTML form commands. The HTTP options are accessed using the streams whose name begins with `InputHttpOtions` or `OutputHttpOptions`. `InputCommands` contain the informations produced by submitting a form. The stream called `Peer` contains the name of the connecting peer.

The class `HttpStream` inherits from `GCNamedObject` and `StreamConfigurationDataBase`. The first attribute, `unsentReplyBody`, is the part of the reply that is not sent yet; the second, `clientStream` is the stream connected to the remote client that should implement a timeout for read operation and translation for specific HTTP hex char groups; `bosyCompletedEvent` is a semaphore on which to wait for body completed.

Then come the public attributes. The attribute `operationMode` represents one of the following the values: `writeToString`, `writeToClient`, `writeToCDB`. The attribute `httpCommand` stores what command was requested; this affects the working of `Read` and `Write` methods, i.e. `Write` method on the main channel does not work for HTTP HEAD tag; it can also be the reply, in which case it contains the reply code. `httpVersion` holds the HTTP version number, i.e. `1000` means *v1.0*, `2100` means *v2.1*. The attribute `keepAlive` is `true` if the communication should continue after transaction; the `url` is the URL of the requested page, the same information is also holded by the `path` attribute that store it

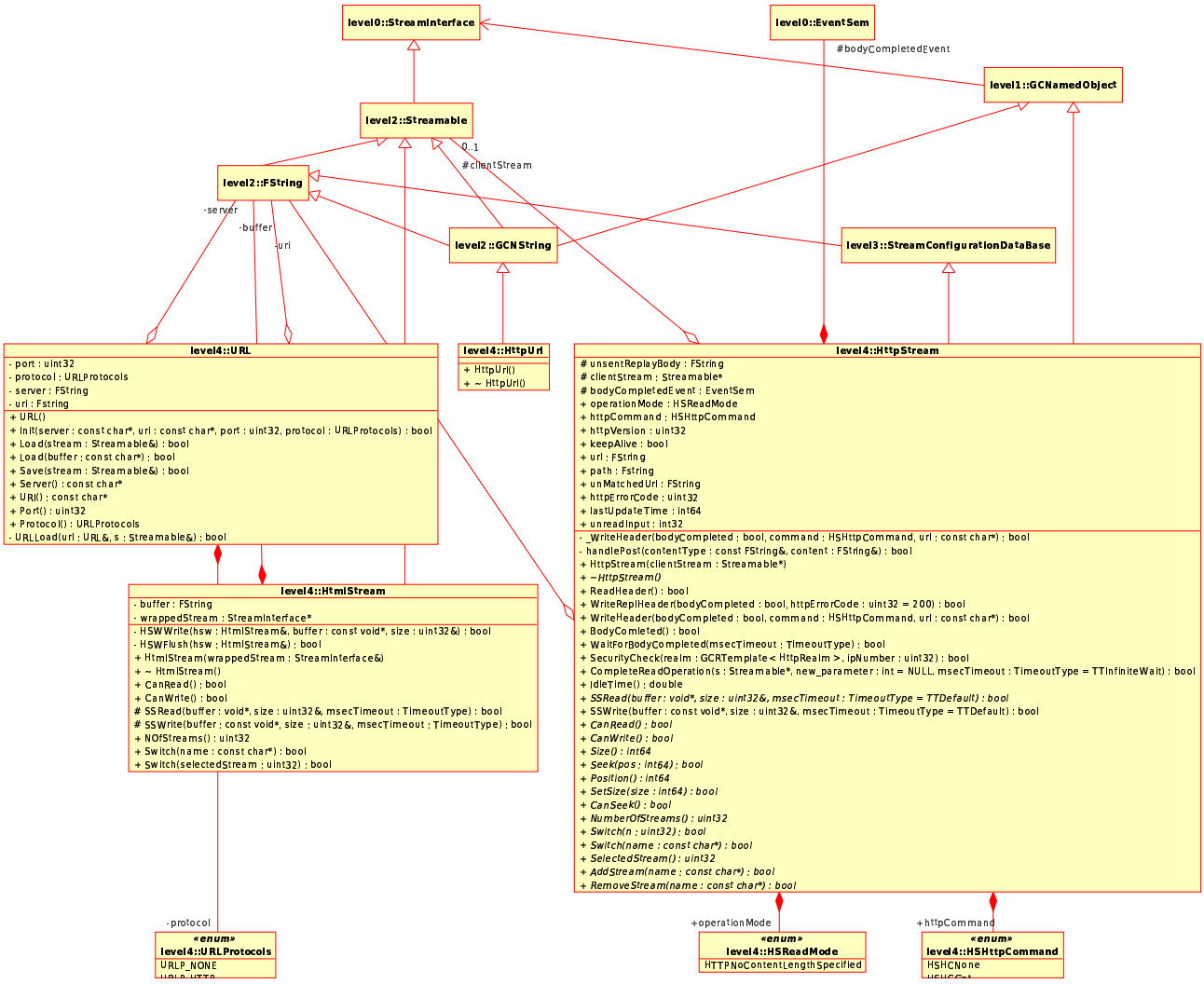

Figure 5.1: BaseLib Level 4 HttpStream and URLs infrastructure

with "." (dotted notation) instead of "". `unMatchedUrl` stores the remainder of url not matched in the search. `httpErrorCode` sves the HTTP return code; the attribute `lastUpdateTime` is the last time the body has been updated and `unreadInput` is how much data is still waiting in the input stream from the client.

```
protected:
   FString unsentReplyBody;
   Streamable* clientStream;
   EventSem bodyCompletedEvent;
public:
   HSOperatingMode operationMode;
   HSHttpCommand httpCommand;
   uint32 httpVersion;
   bool keepAlive;
   FString url;
   FString path;
   FString unMatchedUrl;
   uint32 httpErrorCode;
   int64 lastUpdateTime;
```

```
   int32 unreadInput;
```

The method `handlePost` handles a post request. The constructor builds an empty class, and the destructor simply destroies it.

The method `ReadHeader` reads the HTTP header and leaves body up to the user; `WriteReplyHeader` begins an HTTP reply message: sends out the HTTP header as currently formed and the outstanding body; any further write will operate directly on the stream; if `complete` is `true` then the whole body has been cached and therefore we can announce the body size. `WriteHeader` begins an HTTP message: sends out the HTTP header as currently formed and the outstanding body; any further write will operate directly on the stream if complete is `true` then the whole body has been cached and therefore we can announce the body size, if `command` is a reply we are generating a reply message.

The method `BodyCompleted` returns `true` if the writing of the body has been completed. `WaitForbodyCompletec` is a sender thread that waits for the completion of this activity.

The method `SecurityCheck` checks a request against a security module; `CompleteReadOperation` completes the reading of an HTTP message throwing away the content. `IdeleTime` returns how long since the last write operation.

```
   bool _WriteHeader(bool bodyCompleted,HSHttpCommand command,const char* url);
private:
   bool handlePost(const FString& contentType, FString& content);
public:
   HttpStream(Streamable *clientStream);
   virtual ~HttpStream();

   bool ReadHeader();
   bool WriteReplyHeader(bool bodyCompleted,uint32 httpErrorCode=200);
   inline bool WriteHeader(bool bodyCompleted,HSHttpCommand command,const char* url);

   bool BodyCompleted();
   bool WaitForBodyCompleted(TimeoutType msecTimeout);

   bool SecurityCheck(GCRTemplate<HttpRealm> realm,uint32 ipNumber);
   bool CompleteReadOperation(Streamable* s=NULL,TimeoutType msecTimeout=TTInfiniteWait);
   double IdleTime();
```

Other methods, that follow, override the `Streamable` ones, we report them for a documnentation purpose.

```
   virtual bool SSRead(void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool SSWrite(const void* buffer,uint32& size,TimeoutType msecTimeout=TTDefault);
   virtual bool CanRead();
   virtual bool CanWrite();

   virtual int64 Size();
   virtual bool Seek(int64 pos);
   virtual bool CanSeek();
   virtual int64 Position();
   virtual bool SetSize(int64 size);

   virtual uint32 NumberOfStreams();
   virtual bool Switch(uint32 n);
   virtual bool Switch(const char* name);
   virtual uint32 SelectedStream();
   virtual bool AddStream(const char* name);
   virtual bool RemoveStream(const char* name);
```

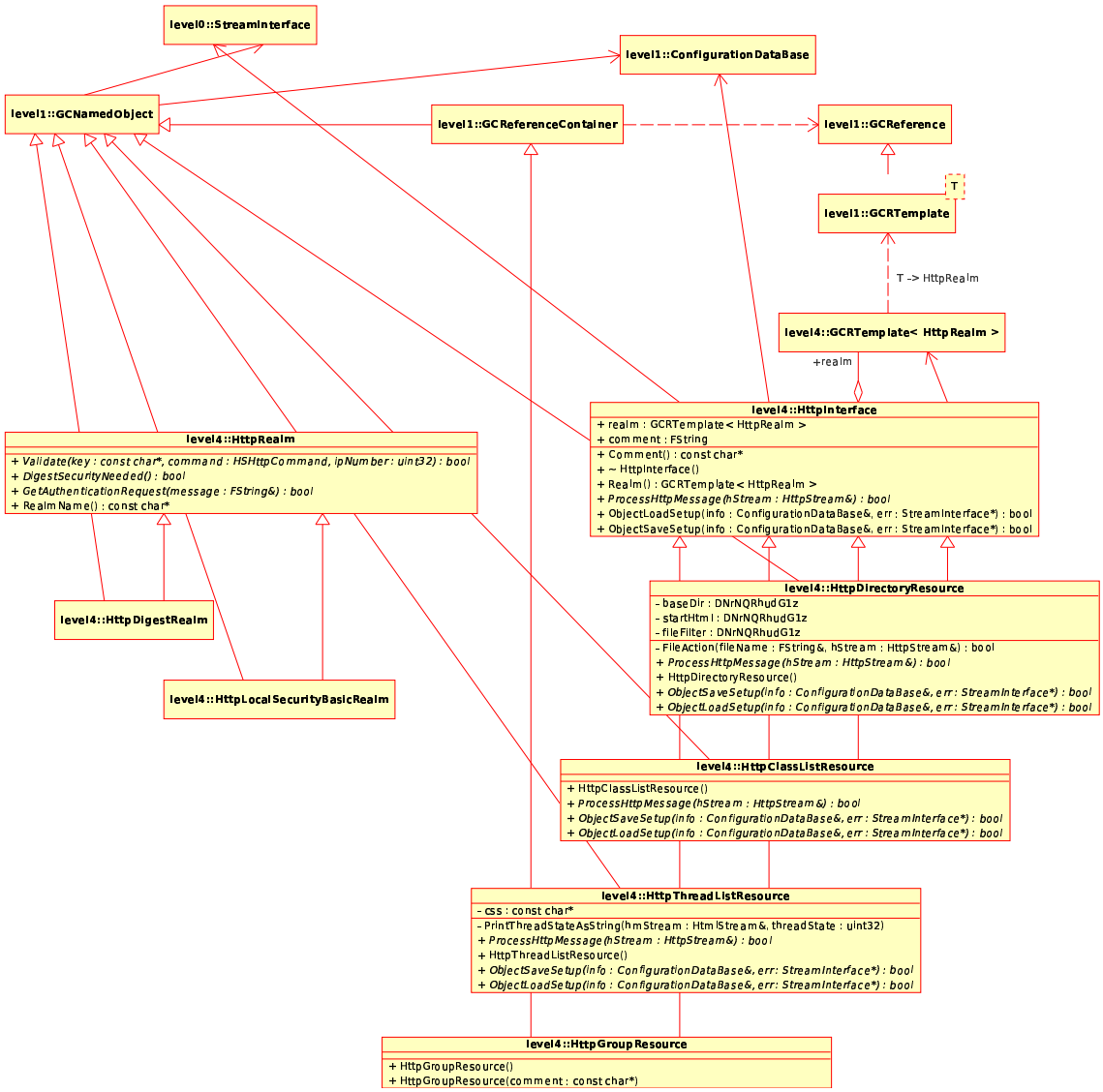

Figure 5.2: BaseLib Level 4 HttpInterface and HttpRealm

5.2 HttpInterface and HttpRealm

Figure 5.2 depict an UML diagram of classes in this section. Classes in this section are (the first three are from the security suite and the others comes from the `HttpInterface` hierarchy):

- HttpRealm
- HttpDigestRealm
- HttpLocalSecurityBasicRealm
- HttpInterface

- HttpDirectoryResource
- HttpClassListResource
- HttpThreadListResoource
- HttpGroupResource
- HttpService RelayResource

HttpInterface

[HttpInterface.h]
The class `HttpInterface` is the base class for a generic HTTP resource object; each resource object that wants to share its state or part of its state via the BaseLib HTTP server must inherits this interface and overriddes the `ProcessHttpMessage` method.

In the overridden `ProcessHttpMessage` the subclass must output an HTTP page about the state of the class.

The method `GCRTemplate<HttpRealm>` is a reference to the *realm* database, i.e. a container of security options; `comment` is the page name to be presented to the viewer.

The method `Comment` return `comment.Buffer()`; `Realm` method return a reference to the `realm` database. The method `ProcessHttpMessage` is the method that will be overridden to obtain an HTTP page for the subclassed object.

The method `ObjectLoadSetup` calls `realm`'s one that is inherited from the `GCRTemplate`; the method `ObjectSaveSetup` behave the same as `ObjectLoadSetup`.

```
public:
   GCRTemplate<HttpRealm> realm;
   FString comment;

public:
   virtual ~HttpInterface(){};
   const char* Comment();
   GCRTemplate<HttpRealm> Realm();
   virtual bool ProcessHttpMessage(HttpStream& hStream)=0;
   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

Chapter 6

BaseLib Level 5

BaseLib Level 5 deals about different software mechanisms that makes components communicate and execute. The first presented mechanism is the *Object's Message Passing*, it is infact possible to send and receive messages between objects, we are used to ear about messages between threads or processes, in BaseLib this concept goes much further and arrive to objects (i.e. a thread its an object in BaseLib). There is a further extension to the *Object's Message Passing* that let's the user manipulate also HTTP messages.

A second really usefull mechanism is the *State Machine*, i.e. the possibility to define and easily create a state machine in a similar fashion of Matlab StateFlow. Each flavour of state machine can be created.

A signal of data can be completely described by the *Signal* class and many signals are packed togheter, by the access mode, to form an *Interface*, more then one interface form a *Generic Application Module* (a GAM). A GAM can be thinked as a Matlab Simulink computational block. A set of blocks communicate between them using a memory buffer, this memory buffer is provided by the Dynamic Data Buffer (DDB). The last mechanism offered by this level are the *Menu*.

Follow a list of the sections in this chapter:

- Messages
- HTTP Messages
- State Machine
- Signals
- DDB, GAM
- Menu
- Others

6.1 Messages

This section introduces the object message passing infrastructure of BaseLib. This is another powerfull subsystem that let you send messages between different objects instances locally and on remote systems (using the UDP/IP protocol). To send remotely a message serializiation and deserialization, seen in the previous chapters, is exploited. Point to point messages and multicast messages are supported. Mainly messages are asynchronous but early and late replay can be sended so synchronous messages can also be created. Summarizing, messages can be:

- local, remote *send*

- point to point, multicast messages (*multiple destinations*)
- immediate, delayed *dispatching*
- early, late *replay* (also acknowledge)

Figure 6.1 try to graph the properties that a message can have and the dependencies between those properties. For example an immediate message doesn't answer to an early replay.

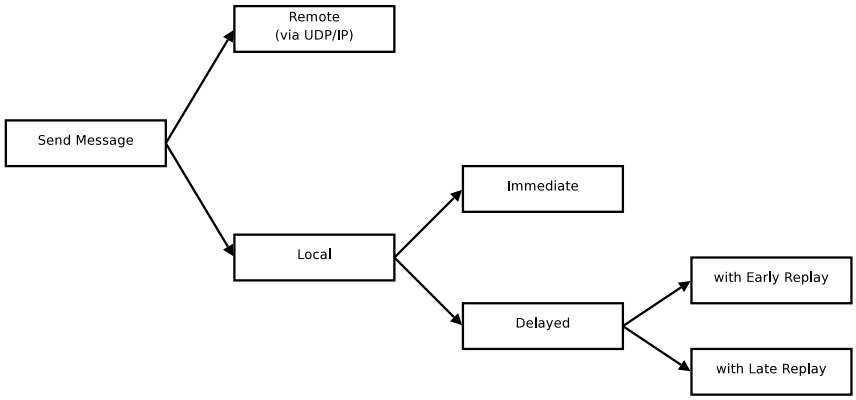

Figure 6.1: BaseLib Level 5 messages properties diagram

Using UDP/IP the policy for remote messages handling is to be Real Time, if it arrives it's ok otherwise doesn't mind, the timeout will save you.

Every message can be build up of any object type, a message is a list of objects that can also be a message itself. The whole UML diagram is depicted in Figure 6.2 and classes involved in this section are:

- MessageCode
- Message
- MessageQueue
- MessageInterface
- MessageHandler
- MDRFlags
- MessageEnvelope
- MessageDeliveryRequest
- MessageDispatcher
- MessageBroker
- StartStopMessageHandlerInterface
- MessageServer

A `MessageCode` is an integer code that let comparing messages on the base of it; otherwise a message can be distingued from a string that is holded by the `Message` class.

Basically the vital work is done by the `MessageHandler` class that depends by the `MessageQueue` that holds messages that can be dispatched not immediately and by the `MessageInterface` that require

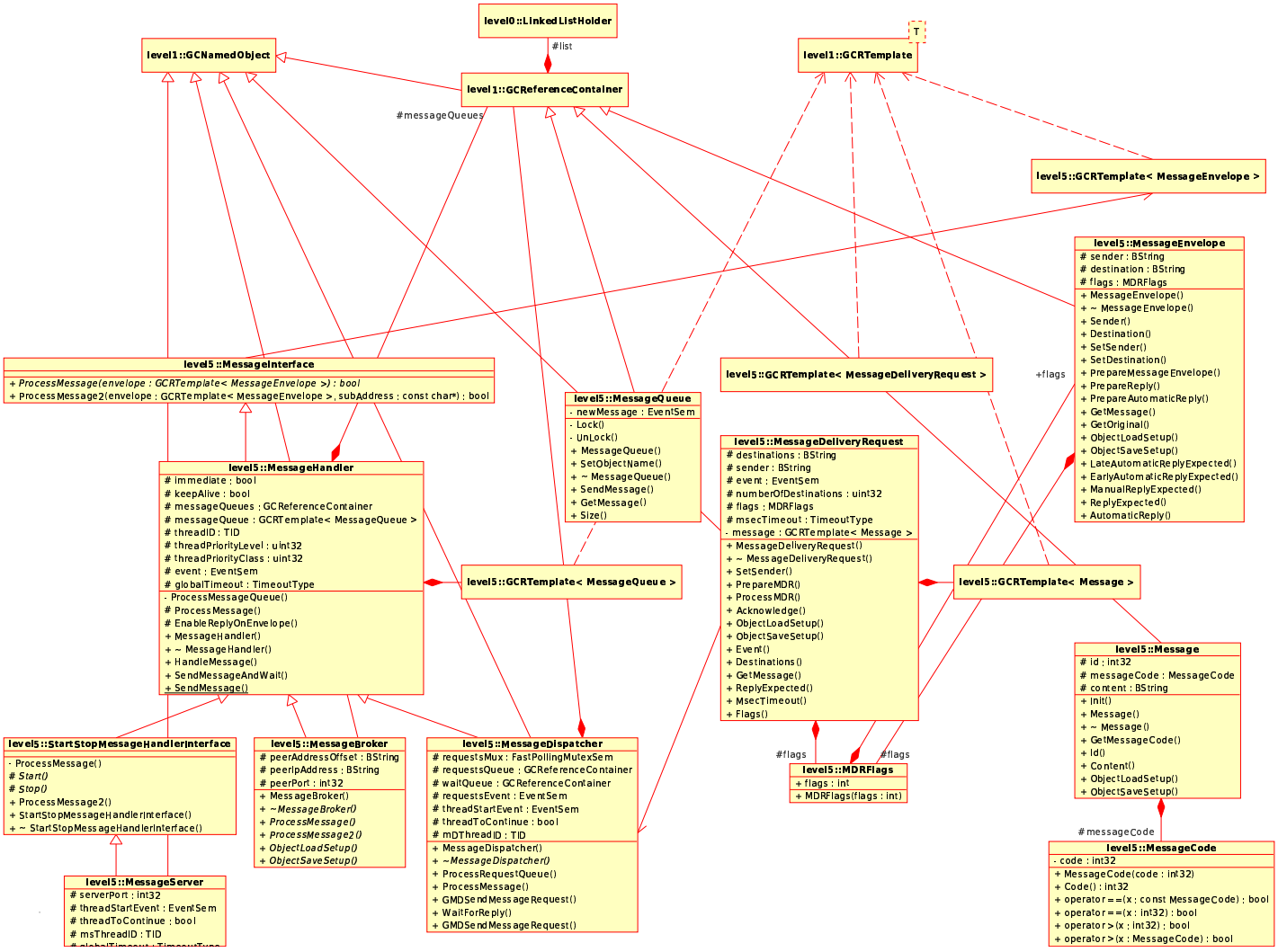

Figure 6.2: BaseLib Level 5 Messages

the implementation of the two processing methods for the messages.

The main method to send messages is the static `MessageHandler::SendMessage()`. This method is static so it is freely callable by any other classes that doesn't know how to send; if a class need a reply must extends the class `MessageHandler` or one of its subclasses. Any class that is a destination of a message must extend `MessageHandler`, and like before, or one of its subclasses.

Figure 6.3 is an UML activity diagram that depict the steps for sending a message thorught the message infrastructure; some comments are needed. To send a message, the message must be first created, so a new `Message` class must be instantiated and a `MessageEnvelope` must be setted up with the source object pointer and the name of the destination class; the final destination class must be a `MessageHandler` subclass, otherwise it can not receive the message. We write "final destination class" because a destiantion address can be a dotted separated address of class names, i.e. the message can be sended between classes in a serial way, this is usefull for example in cases when you need to communicate to a remote class in another machine. An example of a remote location can be "message-

Broker.synchronizationGAM" or "(@192.168.2.3:1234).synchronizationGAM".

After creating a `MessageEnvelope` it is possible to send it via the static `SendMessage` method; the first thing that is done is to check if it is a valid object, then the address is checked to know if it is a remote or a local message. If it is a remote message the `Message` is serialized and an UDP/IP socket is opened and the serialized object is sended.

If the message is local the first things that is done is to check against existance of the receiver object, if it exists in the GODB (the global registry of objects), it will be checked if it is of type `MessageHandler` otherwise the function returns with an error, infact if the receiver is not a `MessageHandler` subclass the message can not be delivered furthermore.

If the classes is founded then the control of the message is passed to class's methods, the first thing is to check against the form of dispatching, if it is *immediate* the class's `ProcessMessage2` is called, otherwise a detached thread is made running and the message is queued for dispatching and the method returns.

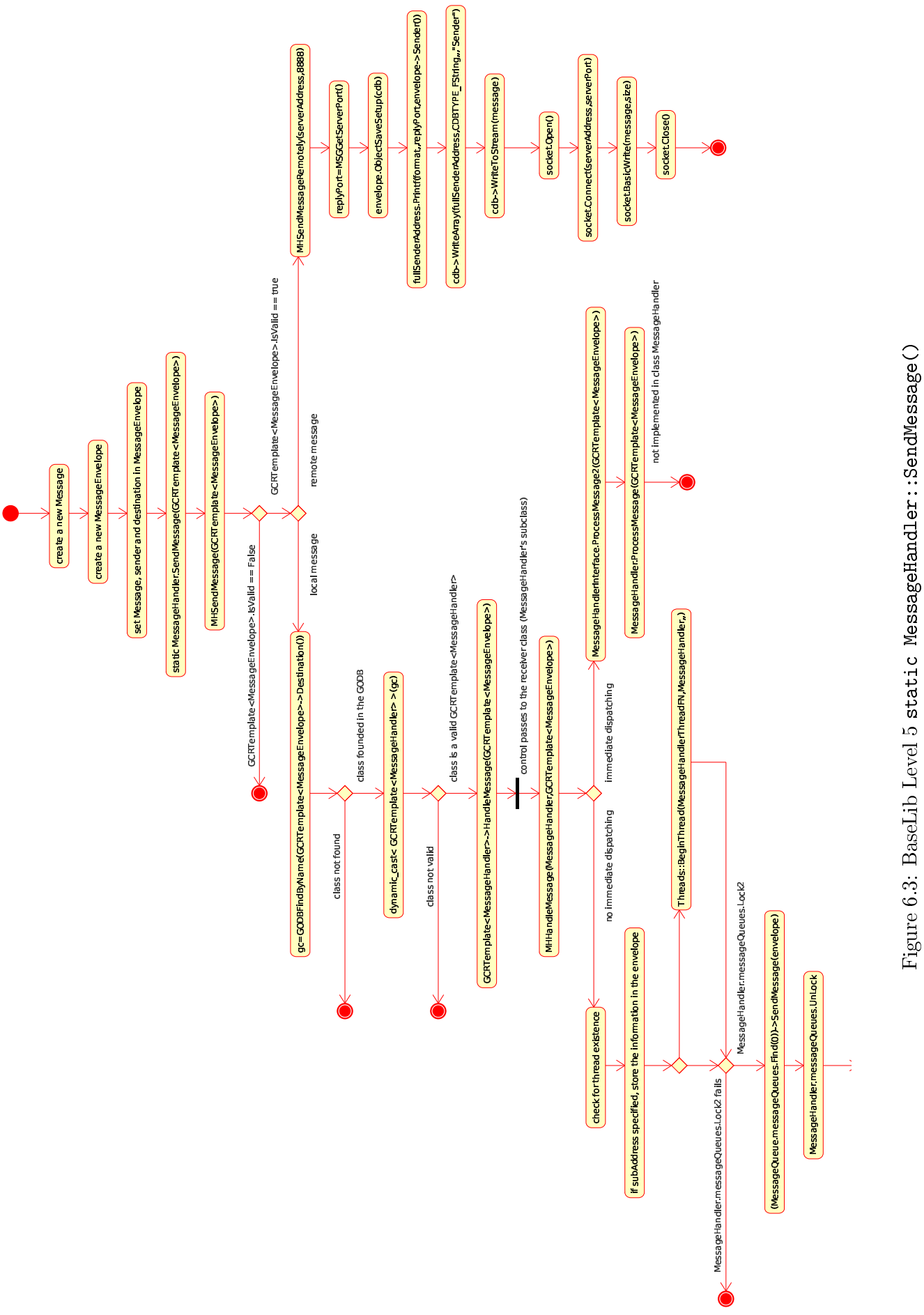

Figure 6.3: BaseLib Level 5 `static MessageHandler::SendMessage()`

The detached thread is a `MessageHandlerThreadFN`, refer to Figure 6.4, this thread is an asynchronous entity that delivers messages queued on the object. For each message an early reply and a late reply can be requested, the early reply is a message sended before the action of calling the `ProcessMessage2` method and the late reply after. There is a special treatment for `MenuMessage`s.

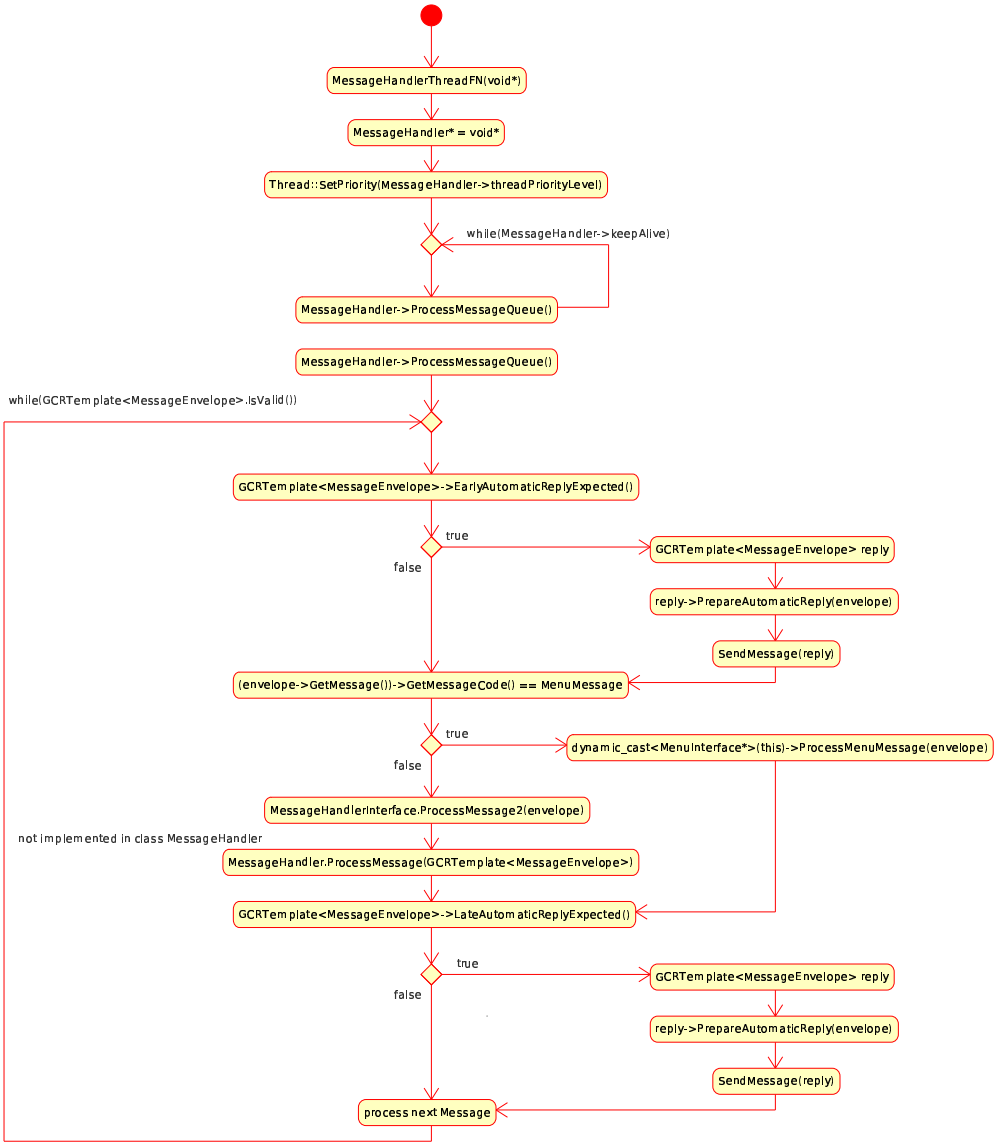

Figure 6.4: BaseLib Level 5 `MessageHandlerThreadFN`

MessageCode

`[MessageCode.h]`
The class `MessageCode` implements the codes of the messages and some already defined codes. Such class is used to describe the content of a message, is just an integer but it enforces type checks. User defined `MessageCode`s starts from `UserMessageCode` (defined in the following).

As before the unique attribute is an `int32` that define the code itself. The constructor creates a new `MessageCode` object with the code passed by argument. `Code` accesses the `code` attribute. Then a set of operator redefinitions comes that are able to compare between message codes.

```
private:
   int32 code;

public:
   MessageCode(int32 code);
   inline int32 Code() const;

   inline bool operator==(const MessageCode x) const;
   inline bool operator==(int32 x) const;
   inline bool operator>(int32 x) const;
   inline bool operator>(MessageCode x) const;
   inline bool operator<(MessageCode x) const
```

The following message codes are statically defined in the *level5/MessageCode.h* file. Each file that include *MessageCode.h* has a local copy of these objects. Is a waste of space.

description	object name	int32 code
message rejected by the original recipient	RejectedMessage	0xFFFFFFFF
used when there is nothing to say	NullMessage	0x0000
standard reply to fully acknowledge a message	FinishedMessage	0x0001
a message directed to a MenuInterface	MenuMessage	0x1001
a message directed to a HttpInterface	HttpMessage	0x1002
first user defined message	UserMessageCode	0x100000
how many user message codes	MaxUserMessageCode	0x100000

Table 6.1: Level5 Message codes

RejectedMessage is the result of a message rejected by the original recipient; NullMessage is an empty message used when there is nothing to say; FinishedMessage is returned when it has finished using the data in the original message, this is the standard reply to fully acknowledge a message. MenuMessage is a message directed to a MenuInterface, the message payload must contain an OutputStream object and possibly an InputStream one. Both must be streamable; HttpMessage is a message directed to a HttpInterface.

Message

[Message.h, Message.cpp]
The class Message is made up with an identification number, a code, a string and an object reference container thanks to the GCReferenceContainer that its inherits from.

The first attribute is an int32, i.e. a system unique message identification, messageCode is a message code of the object and content is the message string.

The Init method sets the messageCode and content attributes. The constructor that is coming sets the messageCode to zero and content to a zero length string and id using the function MSGGetUniqueId. GetMessageCode retrieves the message code as a MessageCode object; Id retrieves the message id; Content retrieves the message content.

ObjectLoadSetup initialises an object from a set of configs, the syntax is:

Code = <a number> or in alternative
UserCode = <a number> in this case the code will be offsetted with UserCode index
Content = <any text (within " if includes spaces)>
or standard GCReferenceContainer syntax

`ObjectSaveSetup` saves the object into a set of configs, the syntax is:

```
Code = <a number>
Id = <a number>
Content = <any text (within " if includes spaces)>
or standard GCReferenceContainer syntax
```

```
protected:
   int32 id;
   MessageCode messageCode;
   BString content;

public:
   void Init(MessageCode code,const char* content);

   Message() :messageCode(0);
   virtual ~Message();

   MessageCode GetMessageCode();
   int32 Id();
   const char* Content();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

MessageQueue

[MessageQueue.h]
A class `MessageQueue` is a `GCReferenceContainer` with a FIFO only policy and an `EventSem` to synchronise tasks to new data arrival. The `EventSem` wake up when a new message is in the queue. The superclass `GCReferenceContainer` mantain a list (a queue) of `GCNamedObject`s and this class exploits this functionality.

Methods `Lock` and `UnLock` guarantees concurrent access to the objects. The constructor create the semaphore `newMessage` and the destructor destroyes it. The method `SetObjectName` calls the `GCNamedObject::SetObjectName` via `GCReferenceContainer` class. The method `SendMessage` enqueue a `GCRTemplate<MessageEnvelope>` object on the `GCReferenceContainer`'s linked list; `GetMessage` returns the oldest message queued in the linked list. `Size` return the number of elements in the linked list.

```
private:
   EventSem newMessage;

   bool Lock(TimeoutType tt = TTInfiniteWait);
   bool UnLock();

public:
   MessageQueue();
   virtual ~MessageQueue();

   void SetObjectName(const char* name);
   bool SendMessage(GCRTemplate<MessageEnvelope> envelope,TimeoutType tt=TTInfiniteWait);
   GCRTemplate<MessageEnvelope> GetMessage(TimeoutType tt=TTInfiniteWait);
   int32 Size();
```

MessageInterface

[MessageInterface.h]
The class `MessageInterface` is formally the interface that can be used to create multiple message handling objects. This interface alone cannot implement a `MessageHandler`; to implement an object with multiple message handling capabilities simply have each component class inherit from this class and then the final class inherit from all the base classes and `MessageHandler`.

If each component class has registered itself with the `MessageHandler` then the standard message handling function will try all the interfaces.

The methods to be implemented by the user application follows.

```
public:
   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope)=0;
   virtual bool ProcessMessage2(GCRTemplate<MessageEnvelope> envelope,const char* subAddress);
```

MessageHandler

[MessageHandler.h, MessageHandler.cpp]
The class `MessageHandler` is the main class for subclassing objects that can handle messages. Based on the `immediate` attribute the `MessageHandler` object will handle the message immediately or it will put the message on a queue and spawn a thread to handle it later.

We now take a look at the attributes of the class. The `immediate` attribute is `true` if the message is processed immediately; note that the user handler function must be able to handle parallel requests to work, in the following we speak about it. The attribute `keepAlive` says if is to keep the thread alive; `messageQueues` are the pools of queues and `messageQueue` is the local message queue.

The attribute `threadID` is the thread identification if not zero it means that a thread is running. `threadPriorityLevel` is the level at which the handler task operates; `threadPriorityClass` is the level at which the handler task operates, priority classes are:

priority class	description
0	Idle
1	Regular
2	Server
3	RT

`event` is a semaphore on which to wait for the task start/stop events and `globalTimeout` is how long to wait for single action.

```
protected:
   bool immediate;
   bool keepAlive;
   GCReferenceContainer messageQueues;
   GCRTemplate<MessageQueue> messageQueue;

   TID threadID;
   uint32 threadPriorityLevel;
   uint32 threadPriorityClass;

   EventSem event;
   TimeoutType globalTimeout;
```

The method `ProcessMessageQueue` is the actual body of the managing thread, method `ProcessMessage` must be overridden in descendent classes; `envelope` should be a valid reference to a `MessageEnvelope`, it returns `true` if the message is considered to have been consumed `false` if the handler does not

recognise the message as its own, at this level it simply return `false`.

`EnableReplyOnEnvelope` if the envelope has no reply mechanism specified basically modify the `envelope::flags` attribute to `MDRF_EarlyAutomaticReply` see more on next classes.

The constructor by default set the `immediate` attribute to `false` requiring the creation of a thread but in the constructor no thread is created and the priority level is sets to zero. The `gloablTimeout` is of about 1000ms, a queue is also created and is queued on the `messageQueues` attribute. The destructor try to destroy a living thread that handle the queues completion if it exists.

The method `HandleMessage` is the function to be called to handle the message, so it must be written to be able to handle parallel requests. Such method calls the `ProcessMessage` method if is a local message. `SendMessageAndWait` sends a message and wait for reply; the last function `SendMessage`, is the only function that sends a message by any other class.

```
private:
   void ProcessMessageQueue();
protected:
   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);

   void EnableReplyOnEnvelope(GCRTemplate<MessageEnvelope> envelope);
public:
   MessageHandler();
   virtual ~MessageHandler();

   inline  bool HandleMessage(GCRTemplate<MessageEnvelope> envelope,
      const char* subAddress = NULL);

   inline bool SendMessageAndWait(GCRTemplate<MessageEnvelope> envelope,
      GCRTemplate<MessageEnvelope>&  reply,TimeoutType timeout=TTInfiniteWait);
   static inline bool  SendMessage(GCRTemplate<MessageEnvelope> gcrtme);
```

MDRFlags

`[MDRFlags.h]`
The `MDRFlags` class, i.e. Message Delivery Request Flags, is used to control `MessageEnvelope` and `MessageDeliveryRequest`. The class follows and is made by only one attribute and no methods.

```
class MDRFlags {
public:
   int flags;
};
```

`MDRFlags` global objects are:

MessageEnvelope

`[MessageEnvelope.h, MessageEnvelope.cpp]`
The class `MessageEnvelope` is a folder containing one or more `Message` objects (due to inherit from `GCReferenceContainer`), the destination and sender object's address and possibly the mail it was replying to. The destination and sender object's address are strings (BString attribute `sender` and `destination`) so destination and sender objects must be searched by name.

The constructor builds an empty envelope setting `sender` and `destination` to "`none`". The method `Sender` retrieves the message sender, `Destination` retrieves the message destination; `SetSender` sets the specified object as the sender of the message, note that you must use an `GCNamedObject` as a sender, i.e. must be an object with a name; `SetDestination` sets the specified object as the destination of a message, the destination object is not checked against existance in this method.

description	object name	int flags
no flags	MDRF_None	0x0000
no flags	MDRF_ReplyMask	0x0003
no flags	MDRF_ReplyNMask	0xFFFFFFFC
Create a reply soon before or after user	MDRF_AutomaticReply	0x0001
Create a reply after or as part of user	MDRF_LateReply	0x0002
Create a reply soon before user	MDRF_EarlyAutomaticReply	0x0001
Create a reply soon after user	MDRF_LateAutomaticReply	0x0003
No reply generated or expected	MDRF_NoReply	0x000
Manual reply expected	MDRF_ManualReply	0x0002
Mask to check for any form of replya expected	MDRF_ReplyExpected	0x003
allows using partialName in address	MDRF_MatchPartialName	0x0004
This is a reply with no new message	MDRF_Reply	0x10000

Table 6.2: Level5 Message Delivery Request flags

```
protected:
   BString sender;
   BString destination;
   MDRFlags flags;

public:
   MessageEnvelope();
   virtual ~MessageEnvelope();

   inline const char* Sender();
   inline const char* Destination();
   inline void SetSender(GCNamedObject& sender);
   inline void SetDestination(const char* destination);
```

The method `PrepareMessageEnvelope` adds the `message` passed by argument to the `GCReferenceContainer` list using `GCReferenceContainer::Insert` method and sets `sender`, `destination` and `flags` attributes; if `destination` starts with `::` the name search scope is the global object container otherwise it is the message receiver container; use `flags` to request a reply.

The method `PrepareReply` creates a `MessageEnvelope` as a reply to the `MessageEnvelope`, swaps sender and destination and copies all content; `maxHistory` regulates the number of old-messages left in the envelope.

The method `PrepareAutomaticReply` is the acknowledge to a message reception; it is automatically generated from the original enevelope, message's `flags` is `MDRF_Reply`.

The method `GetMessage` returns the first message in the envelope that can hold more then one message, it uses `GCReferenceContainer::Find`. `GetOriginal` gets the original message if this is a reply, `index` default `1` is the original message any higher number is the history of messages.

The usual `ObjectLoadSetup` initialises an object from a set of configs, the syntax is:

```
Sender = <any text (within "" if includes spaces) >
Destination = <any text (within "" if includes spaces) >
<Use syantax of GCReferenceContainer>
Add = {
    MessageName = {
      <See Message syntax>
    }
    [ReplyName = {
       <See Message syntax>
    }]
```

```
}
```

`ObjectSaveSetup` saves an object to a set of configs; the syntax is:

```
Sender = <any text (within "" if includes spaces) >
Destination = <any text (within "" if includes spaces) >
Content = {
        <Use syantax of GCReferenceContainer>
        Add = {
          Message = {
            <See Message syntax>
          }
          [Reply = {
             <See Message syntax>
          }]
        }
        ... (other message names are possible)
}
```

The following methods simply tests for if the `flags` attribute has a particular value. `LateAutomaticReplyExpect` is `true` if an automatic reply shall be generated after processing, `EarlyAutomaticReplyExpected` is `true` if an automatic reply shall be generated, `ManualReplyExpected` is `true` if a manual reply shall be generated by the user, `ReplyExpected` is `true` if a manual reply shall be generated by the user and `AutomaticReply` is `true` if an automatic reply.

```
   inline bool PrepareMessageEnvelope(GCRTemplate<Message> message,
      const char* destination,MDRFlags flags=MDRF_None,GCNamedObject* source=NULL);
   inline bool PrepareReply(GCRTemplate<MessageEnvelope> messageEnvelope,
      GCRTemplate<Message> replyMessage,MDRFlags flags=MDRF_None,int maxHistory=2);
   inline bool PrepareAutomaticReply(GCRTemplate<MessageEnvelope> envelope);

   inline GCRTemplate<Message> GetMessage();
   inline GCRTemplate<Message> GetOriginal(int index=1);

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   inline bool LateAutomaticReplyExpected() const;
   inline bool EarlyAutomaticReplyExpected() const;
   inline bool ManualReplyExpected() const;
   inline bool ReplyExpected() const;
   inline bool AutomaticReply() const;
```

MessageDeliveryRequest

`[MessageDeliveryRequest.h, MessageDeliveryRequest.cpp]`
The class `MessageDeliveryRequest` is a container that will hold a message and associated data; the class holds a sender reference, a list of recipients and a mechanism to synchronise a sender with the return receipts from all its recipients. The class `MessageDeliveryRequest` inherits from `GCNamedObject`.

Attributes are all protected, there is an explicit `GCRTemplate<Message>` attribute that holds a `Message` objects via templatization. `sender` is who made the request; `destination` is a comma/space separated list of destinations and `nuberOfDestinations` is the number of valid destinations created when sending messages by `ProcessMDR`. The attribute `event` is a sempahore on wich to wait for reply at least `msecTimeout` milliseconds; other options are stored in `flags`.

```
protected:
   GCRTemplate<Message> message;

   BString sender;
   BString destinations;
```

```
   uint32 numberOfDestinations;

   EventSem event;
   TimeoutType msecTimeout;
   MDRFlags flags;
```

The constructor builds an empty request with zero destinations and inifinte timeout creting also the semaphore; the destructor simply delete the semaphore. `SetSender` sets the specified object as sender of the message, like `MessageEnvelope` should be a `GCNamdeObject`.

The method `PrepareMDR` sets all the arguments passed by to the attributes in the class; the message is saved in the `message` attribute. `ProcessMDR` prepares envelopes and delivers them, the delivery is done via the `MessageHandler::SendMessage` static method for each destination in the `destinations` attribute. `Acknowledge` counts down the number of messages to acknowledge, returns `true` only when all acks are received.

The method `ObjectLoadSetup` initialises an object from a set of configs; the syntax is (the object name is taken from the current node name):

```
Sender = <any text (within "" if includes spaces) >
Destinations = <any text (within "" if includes spaces)
               multiple destinations are space or comma separated >
MsecTimeOut = how msec to wait for reply
Flags = [ EarlyAutomaticReply LateAutomaticReply NoReply ManualReply ]
Message = {
  Class = Message
  <See Message syntax>
}
```

`ObjectSaveSetup` saves an object to a set of configs; the syntax is as follows:

```
Sender = <any text (within "" if includes spaces) >
Destinations = <any text (within "" if includes spaces) >
Message = {
  Class = Message
  <See Message syntax>
}
```

`Event` returns `event` attribute, `Destinations` returns the destinations string and `GetMessage` returns the message.

`Flags` returns `flags` attribute, `ReplyExpected` is `true` if we want wait for reply, `MsecTimeout` is how long to wait for a reply.

```
public:
   MessageDeliveryRequest();
   ~MessageDeliveryRequest();

   inline void SetSender(GCNamedObject& sender);

   inline bool PrepareMDR(GCRTemplate<Message> message,
      const char* destinations,MDRFlags flags=MDRF_None,
      GCNamedObject* source=NULL,TimeoutType msecTimeout=TTInfiniteWait);
   bool ProcessMDR(MessageDispatcher *md);
   bool Acknowledge();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   inline EventSem& Event();
   inline const char* Destinations() const;
   inline GCRTemplate<Message> GetMessage() const;

   inline MDRFlags Flags() const;
```

```
   inline bool ReplyExpected() const;
   inline TimeoutType MsecTimeout() const;
```

MessageDispatcher

[MessageDispatcher.h, MessageDispatcher.cpp]
The class MessageDispatcher is a special global object that handles sending messages for third parties. Those class provide message sending to single/multiple recipients; provides also wait for acknowledge service from single/multiple destinations. Such class is teh first of the three implementations of the ProcessMessage method in this level (in MessageHandler and in MessageInterface simply return false or is not implemented).

It inherits from GCNamedObject and MessageHandler. There is a globally declared object globalMessageDispat that formally is a GCRTemplate<MessageDispatcher> and is returned by the global function GlobalMessageDispatc in *level5/MessageDispatcher.cpp*.

The first attribute, requestMux protects access to the requestsQueue attribute that is the next argument. waitQueue is the queue of elements to be acknowledge. The semaphore requestsEvent is used to mark a new request in the list, threadStartEvent to mark the start of the handling thread, the boolean threadToContinue requests the end of the processing thread, mDThreadID is the thread identifier, if not zero it means a thread is running.

```
protected:
   FastPollingMutexSem requestsMux;
   GCReferenceContainer requestsQueue;
   GCReferenceContainer waitQueue;
   EventSem requestsEvent;
   EventSem threadStartEvent;
   bool threadToContinue;
   TID  mDThreadID;
```

The constructor creates semaphores, sets threadToContinue to false and then create a new MessageDispatcherThreadFN thread. The destructor closes the semaphores and try to kill the associated thread.

The method ProcessRequestQueue processes requests on the requestsQueue, for every job on such queue it delegates the job to the static MessageHandler.SendMessage() or to the ProcessMessage method if is a MessageDeliveryRequest. The method ProcessMessage is the virtual function needed by MessageHandlerInterface that process the Messages, i.e. it sends them. What the method does is to simply acknowledge the messages received, is the subclass that has to interpret the message.

The method GMDSendMessageRequest queues the request and makes the task wait if acknowledge is requested, if waitForReply is true the method will wait for reply if any of the relevant MDRF_* flags are set. Last method WaitForReply will wait for reply if MDRF_AutomaticReply is set.

```
public:
   MessageDispatcher();
   virtual ~MessageDispatcher();

   inline bool ProcessRequestQueue();
   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);

   inline bool GMDSendMessageRequest(GCRTemplate<MessageDeliveryRequest> messageRequest,
      TimeoutType msecTimeout,bool waitForReply);
   inline bool GMDSendMessageRequest(GCRTemplate<MessageEnvelope> messageRequest,
      TimeoutType msecTimeout);

   inline bool WaitForReply(GCRTemplate<MessageDeliveryRequest> messageRequest,
      TimeoutType msecTimeout=TTInfiniteWait);
```

MessageBroker

[MessageBroker.h, MessageBroker.cpp]
A MessageBroker is a special object that allows to deliver message to a remote system; i.e. it permits to send messages to a remote program on a diffrent machine on which a MessageServer is installed and active. MessageBroker inherits from GCNamedObject and MessageHandler.

Starting as usual with attributes peerAddressOffeset is a part of the remote address that will be attached to the sub address, peerIpAddress holds the IP address of the machine to send the message to and peerPort is the UDP port on the remote machine.

```
protected:
   BString peerAddressOffset;
   BString peerIpAddress;
   int32 peerPort;
```

The constructor simply set the peerPort to zero and the destructor doesn't do anything. Then comes the two methods inherited from MessageHandlerInterface the first one ProcessMessage is not implemented and returns false and signal a fatal error on the error console; the second one, ProcessMessage2 is implemented; this method checks message's paramenters and calls the MHSendMessageRemotely (in *level5/MessageHandler.cpp*) using class's peerIpAddress and peerPort as arguments; see Figure 6.3.

The method ObjectLoadSetup initialise the object parameters, the first parameter in the configuration database can be PeerAddressOffset and can be omitted, if declared changes the address used as destination by replacing the part matched so far with this string, PeerIpAddress is the peer IP address and PeerPort is the peer port number; ObjectSaveSetup saves settings to the CDB.

```
public:
   MessageBroker();
   virtual ~MessageBroker();

   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
   virtual bool ProcessMessage2(GCRTemplate<MessageEnvelope> envelope,const char* subAddress);

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

StartStopMessageHandlerInterface

[StartStopMessageHandlerInterface.h, StartStopMessageHandlerInterface.cpp]
The class StartStopMessageHandlerInterface is rougly an interface, it will implements a MessageHandlerInterfa that is able to handle Start and Stop messages. The class StartStopMessageHandlerInterface inherits from MessageHandler.

The method ProcessMessage is the only implemented method in the class, and processes the message, if the message content is START or STOP it calls the Start or Stop function accordingly. If the message content is not START or STOP it calls the ProcessMessage2 function. Those methods are abstract methods in this class.

```
private:
   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);

protected:
   virtual bool Start() = 0;
   virtual bool Stop() = 0;

public:
```

```
   virtual bool ProcessMessage2(GCRTemplate<MessageEnvelope> envelope);
   StartStopMessageHandlerInterface();
   virtual ~StartStopMessageHandlerInterface();
```

MessageServer

[MessageServer.h, MessageServer.cpp]
The class `MessageServer` is a special object that allows two programs communicate seamlessly, precisely this is the message incoming server; the previous `MessageBroker` sends messages to the `MessageServer`. The class `MessageServer` inherits from `GCNamedObject` and from `StartStopMessageHandlerInterface`.

The first attribute `serverPort` is the UDP port to receive from, `threadStartEvent` marks the start of the handling thread, `threadToContinue` requests the end of the processing thread if `true`; `msThreadID` is the thread identifier, if not zero it means a thread is running; `globalTimeout` is the timeout in use: wait for messages for this amount then check for closure condition.

```
protected:
   int32 serverPort;
   EventSem threadStartEvent;
   bool threadToContinue;
   TID msThreadID;
   TimeoutType globalTimeout;
```

The constructor creates a new `MessageServer` without any thread and initialise the `EventSem` the destructor calls the `Stop` method.

The method `Start` implements the `StartStopMessageHandler::Start` function creating a new `MessageServerThreadFN` thread. `Stop` method kills any alive thread of the previous type created by the class.

```
public:
   MessageServer();
   virtual ~MessageServer();

   virtual bool Start();
   virtual bool Stop();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

The C function `MessageServerThreadFN` is implemented as follow.

```
void MessageServerThreadFN(void *arg){
   Threads::SetPriorityLevel(0);
   MessageServer *ms = (MessageServer *)arg;
   if (ms != NULL){
      ms->threadToContinue = True;
      ms->threadStartEvent.Post();
      MSProcessIncomingMessages(*ms);
      ms->msThreadID = (TID)0;
      ms->threadStartEvent.Post(); }
}
```

The function `MSProcessIncomingMessages` opens the UDP port setted and await for incoming message, when a message arrive it will be casted and sended using `MessageHandler::SendMessage`.

6.1.1 Remarks

TODO
TODO
TODO
esempio esempi

6.1.2 Design Notes

So what are the differences between `MessageEnvelope` and `MessageDeliveryRequest`? Basically a `MessageEnvelope` should be sended manually by the developer and has only **one** destination; a `MessageDeliveryRequest` instead has multiple destinations and creates and sends multiple `MessageEnvelope` one for each destination (with the same messages). The send operation is done automatically from a `MessageDeliveryRequest` by the `MessageDeliveryRequest::ProcessMDR`.

A message with a single destination usually is enbodyed within a `MessageEnvelope`, such object is sended using the static `MessageHandler::SendMessage()`. A message with many destinations is enbodyed within a `MessageDeliveryRequest` object that is sended using its own `ProcessMDR` that do the multicast send, sending the same message (or group of messages) to each destination. The `ProcessMDR` method expects a `MessageDispatcher` as an argument and this kind of object can be obtained at runtime by the function `GetGlobalMessageDispatcher` that return a global registered `MessageDispatcher`. It's the user of the `MessageDeliveryRequest` that must know how to do that to send a multicast message.

There are two ways to send a message to a remote object but only one way to receive it in the remote system: to add (in the CDB) the instantiation of a `MessageServer`. If on the remote server no instance of the `MessageServer` exists the remote machine doesn't receive messages. The most directed way to send a remote message explicitly writing down the ip address and the port is via the static method `MessageHandler::SendMessage()` the other way is using a `MessageBroker`, in this case is such class that knows the remote host address.

6.2 HTTP Messages

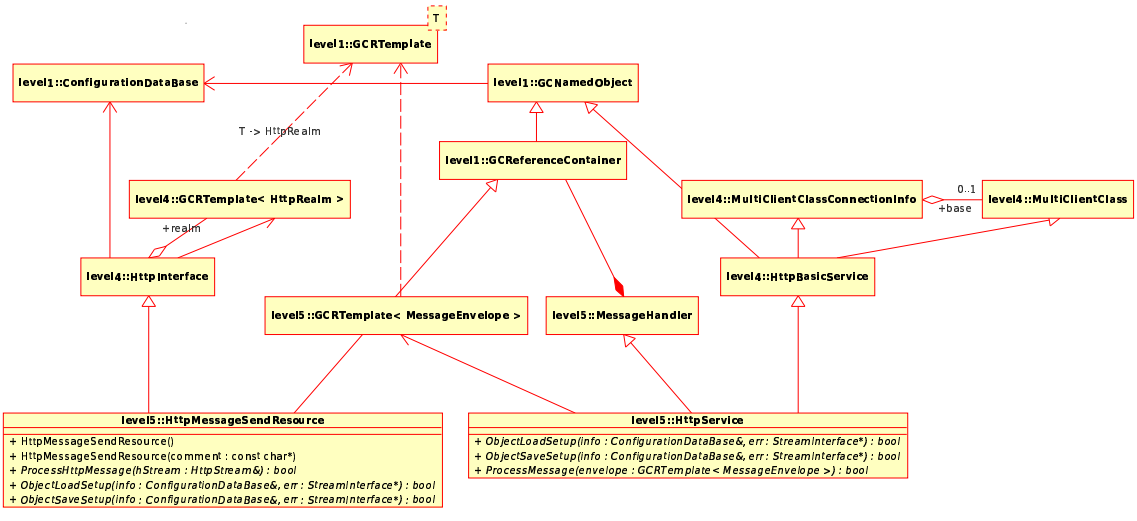

Figure 6.5: BaseLib Level 5 HTTP Messages

HttpMessageSendResource

`[HttpMessageSendResource.h, HttpMessageSendResource.cpp]`
TODO

HttpService

`[HttpService.h, HttpService.cpp]`
TODO

6.3 State Machine

The state machine implemented in this section is an extended one that also deal with errors and system failures, it's similar to the MATLAB© Stateflow© behaviour.

Basically a state machine is build up on two sets of elements: firstly a **set of states** and a **set of archs** that connect states and identify the happening of an event. When an event is triggered (i.e. by an external event) the state machine is in a state, in this state it will search if there is an associated event's behaviour and if it is the state machine execute the registered actions. If an error occours there is basically an error state to reach.

Actions usually are in form of messages sended between objects, i.e. think about a windows system where each window object can receive and send messages and react on those messages. In fact the state machine is the first entity using the message infrastructure.

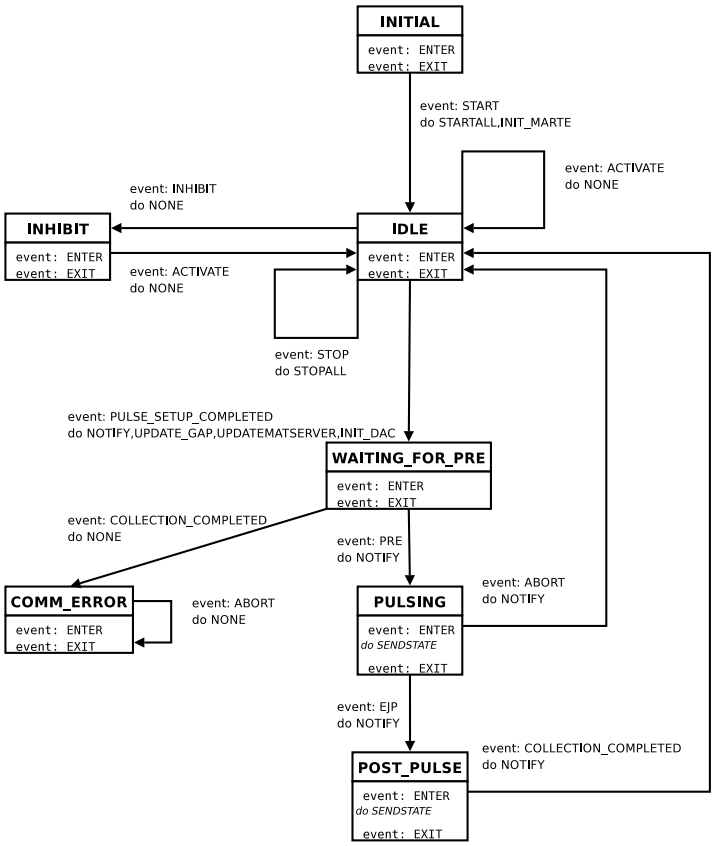

Figure 6.6: VS5 State Machine example

Figure 6.6 represent an example of the state machine required in the Vertical Stabilization control algorithm. In the following comes an excerpt of the configuration file that generate this state machine. There is no tool right now for the autogeneration of the configuration file from the design (that is actually not fully sketched: default actions, error and fault conditions are not visible).

```
+StateMachine = {
    Class = StateMachine
    VerboseLevel = 10
```

```
    +INITIAL = {
        Class = StateMachineState
        StateCode = 0x0
        +START = {
            Class = StateMachineEvent
            NextState = IDLE
            Value = START
            +STARTALL = {
                Class = MessageDeliveryRequest
                Sender = StateMachine
                Destinations = "HTTPSERVER␣CODAS␣MARTe"
                MsecTimeOut = 1000
                Flags = NoReply
                Message = {
                    Class = Message
                    Content = START
                }
            }
            +INIT_MARTe = {
...
            }
        }
    }
    +IDLE = { ...
    }
    +WAITING_FOR_PRE = { ...
    }
    +PULSING = { ...
    }
    +POST_PULSE = { ...
    }
    +INHIBIT = { ...
    }
    +COMM_ERROR = { ...
    }
}
```

In this example we saw how the configuration file works for a state machine: it really relies on `GCReferenceContainer` syntax. For each state you can register an action on entry and on exit and a default action. Each arch has a set of actions associated with it. The state machine react to events acting as preconfigured dependent on the current state.

Class involved in this section are depicted in the UML schema of Figure 6.7 and are listed below:

- StateMachineState
- StateMachineEvent
- StateMachine

State Machine State

`[StateMachineState.h, StateMachineState.cpp]`
The class `StateMachineState` represents a state within the state machine; the `StateMachineState` inherits from `GCReferenceContainer` and so it is a container of `StateMachineEvent`s or `MessageDeliveryRequest`s.

There are only two `MessageDeliveryRequest`s one named "`ENTER`" and one "`EXIT`"; these messages are delivered on entering or exiting the state.

The enumeration `SMS_ActOnResults` is first introducted as below, and is usually the result of some methods implemented in the `StateMachineState` class.

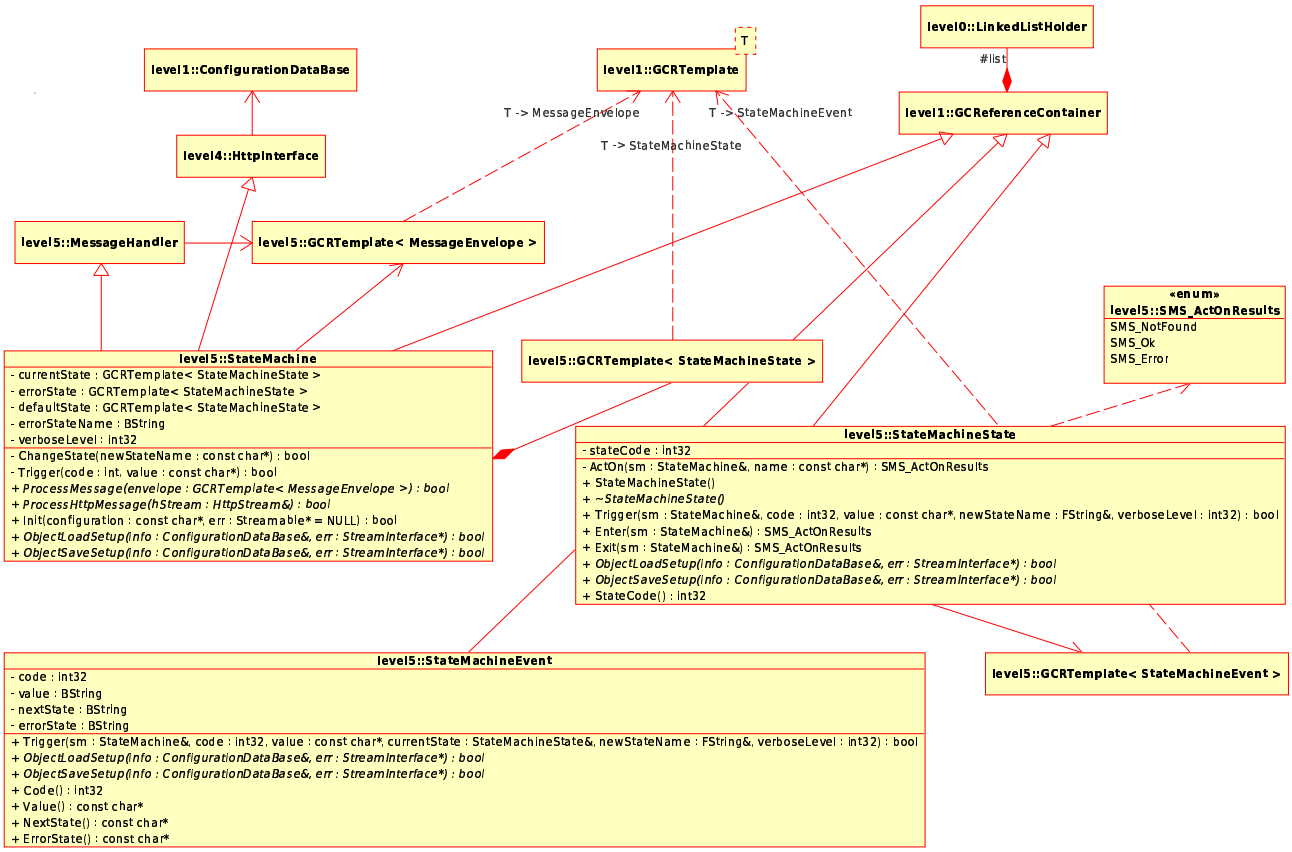

Figure 6.7: BaseLib Level 5 State Machine

```
enum SMS_ActOnResults{
   SMS_NotFound = 0,
   SMS_Ok = 1,
   SMS_Error = -1,
};
```

The `stateCode` class's attribute is a code associated with this state. The first, private method, `ActOn` perform the activity called `name` and returns `SMS_NotFound` if the action was not found or `SMS_Error` in case of errors; the method `ActOn` searches for a `name` classes throught the list holded by the `GCReferenceContainer` superclass, it tries to cast it first to a `MessageEnvelope` and then to a `MessageDeliveryRequest`. If it succeeds to cast to `MessageEnvelope` searchs again for a "`SENDSTATE`" message; otherwise compares the mesagge string with the "`SENDSTATE`" string; then send the message, the `StateMachine` passed by argument is the source object.

The constructor sets the `stateCode` attribute to zero. The destructor does anything usefull. The method `Trigger` finds this trigger and execute the associated actions returning the new state; the trigger infrastructure deals with `StateMachineEvents`. `Enter` simply calls the method `ActOn` with `name` setted to "`ENTER`", the method `Exit` with "`EXIT`".

The common method `ObjectLoadSetup` initialises an object from a set of configs, the syntax is the same as `GCReferenceContainer` one shall also define a code using `StateCode = nnn`; the main container content should be only of class `StateMachineEvent`; if an object called "`ENTER`" is added, of type `MessageDeliveryRequest` or `MessageEnvelope` than a message is sent on entry; if an object called "`EXIT`" is added of type `MessageDeliveryRequest` or `MessageEnvelope` than a message is sent on exit; if a `StateMachineEvent` called "`DEFAULT`" is encountered while matching the trigger then

a match is always automatically found; the object name is taken from the current node name if a MessageEnvelope or MessageDeliveryRequest with the first object contained being a Message called "SENDSTATE" is found than the current state is used as content for that Message. In this case the MessageEnvelope and MessageDeliveryRequest are duplicated, note that the current state not the final state are sent.

ObjectLoadSetup saves an object to a set of configs, the syntax is the same as GCReferenceContainer; the container content should be only of class StateMachineState. The object name is taken from the current node name.

```
private:
   int32 stateCode;

   SMS_ActOnResults ActOn(StateMachine& sm,const char* name);
public:
   StateMachineState();
   virtual ~StateMachineState();

   bool Trigger(StateMachine& sm,int32 code,const char* value,FString& newStateName,int32 verboseLev
   inline SMS_ActOnResults Enter(StateMachine& sm);
   inline SMS_ActOnResults Exit(StateMachine& sm);

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   inline int32 StateCode();
```

State Machine

[StateMachine.h, StateMachine.cpp]
The class StateMachine is a generic state machine that reacts to messages and acts by sending messages. The code is organized as a container of StateMachineState and a MessageHandler: when it receives a message it matches it to the StateMachineEvent contained in the currentState.

If code matches or if (code equal zero) content matches than it sends all the messages contained in StateMachineEvent. A reply to the original message is sent with the parameters (State_code,"State_label") if the original message required manual reply ErrorState is the state the machine reaches if the specified state cannot be found. By default its value is ERROR but if ERROR does not exist then the state will not change. See StateMachineState for the ENTER EXIT action objects and for the Events. If any of the ENTER or EXIT actions fail the state machine will reach ErrorState DEFAULT is a state whose Events and ENTER/EXIT actions are appended to those of the current state if the next state is specified as "SAMESTATE" this means that not change of state will occurr; if a MessageEnvelope or MessageDeliveryRequest with the first object contained being a Message called "SENDSTATE" is found than the current state is used as content for that Message. In this case the MessageEnvelope and MessageDeliveryRequest are duplicated. Note that the current state not the final state are sent

class StateMachine: public GCReferenceContainer, public MessageHandler, public HttpInterface

The first three attributes are all of type GCRTemplate<StateMachineState> the first one is a refrence to the currently active state: initialised with the first state; refrence to the default error state: last state or the one called "ERROR" and the last is another reference to a state called "DEFAULT". "ENTER" and "EXIT" messages are executed before and after the state specific ones at every state change. The attribute errorStateName is the name of the error state, by default "ERROR"; if verboseLevel is zero means no diagnostics, 1 shows all the state changes.

```
   GCRTemplate<StateMachineState> currentState;
   GCRTemplate<StateMachineState> errorState;
   GCRTemplate<StateMachineState> defaultState;
   BString errorStateName;
   int32 verboseLevel;
```

The method `ChangeState` implement a change of state if `newStateName == SAMESTATE` will not change state returns `true` if the state has changed.

`Trigger` processes the `code` or `value` arguments to determine the next state.

The `StateMachine` is basically a `MessageHandler` so it should implement the `MessageHandlerInterface` providing `ProcessMessage` or `ProcessMessage2` for incoming message handling. `ProcessHttpMessage` is the main entry point for `HttpInterface`.

`Init` creates a menu tree creating a stream first then a CDB and then using ObjectLoadSetup.

The usual `ObjectLoadSetup` and `ObjectSaveSetup` come; for both the syntax is the same as `GCReferenceContainer`, the container content should be only of class StateMachinState optional parameter is ErrorStateName which allows selecting the ErrorState; if NextState is SAMESTATE no change will occur VerboseLevel integer if set >0 will cause diagnostic messages to appear >=1 shows status changes >=2 shows actions performed >=10 shows message processing details; the object name is taken from the current node name.

```
   bool ChangeState(const char* newStateName);
   bool Trigger(int code,const char* value);

public:
   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
   virtual bool ProcessHttpMessage(HttpStream& hStream);

   bool Init(const char* configuration,Streamable* err=NULL);

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

StateMachineEvent

`[StateMachineEvent.h, StateMachineEvent.cpp]`
The class `StateMachineEvent` is an event contained(referred to) within a state. This is a container of `MessageDeliveryRequest` it also can contains a set of `StateMachineState`. The `StateMachineEvent` class is also a `GCReferenceContainer`.

The first attribute `code` is a code associated with this event, zero means use the `value` attribute; such attribute is a string value associated with this event. When this event occurs the next state has name `nextState` and if the action fails we go to `errorState`.

```
   int32 code;
   BString value;
   BString nextState;
   BString errorState;
```

The method `Trigger` performs the action associate with a trigger.

`ObjectLoadSetup` initialises an object from a set of configs; the syntax is the same as GCReferenceContainer; the container content should be only of class `MessageEnvelope` or `MessageDeliveryRequest` important parameters are:

`Code`	(an integer) or
`UserCode`	(an integer)
`Value`	(a string) that identify the trigger
`NextState`	(a string) is the next state and
`ErrorState`	(a string) is the next state on error

The object name is taken from the current node name. If the object is called "`DEFAULT`" than it matches any message code, if a `MessageEnvelope` or `MessageDeliveryRequest` called "`SENDSTATE`" is founded than the current state is added as content. In this case the `MessageEnvelope` and `MessageDeliveryRequest` are duplicated. Note that the current state not the final state are sent. `ObjectSaveSetup` saves an

object to a set of configs.

Then follows a set of getter methods that returns the attributes content.

```
public:
   bool Trigger(StateMachine& sm,int32 code,const char* value,StateMachineState& currentState,FStrin

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   int32 Code();
   const char* Value();
   const char* NextState();
   const char* ErrorState();
```

6.4 Signals

Class involved in this section are depicted in the UML schema of Figure 6.8 and are listed below:

- SignalInterface
- Signal

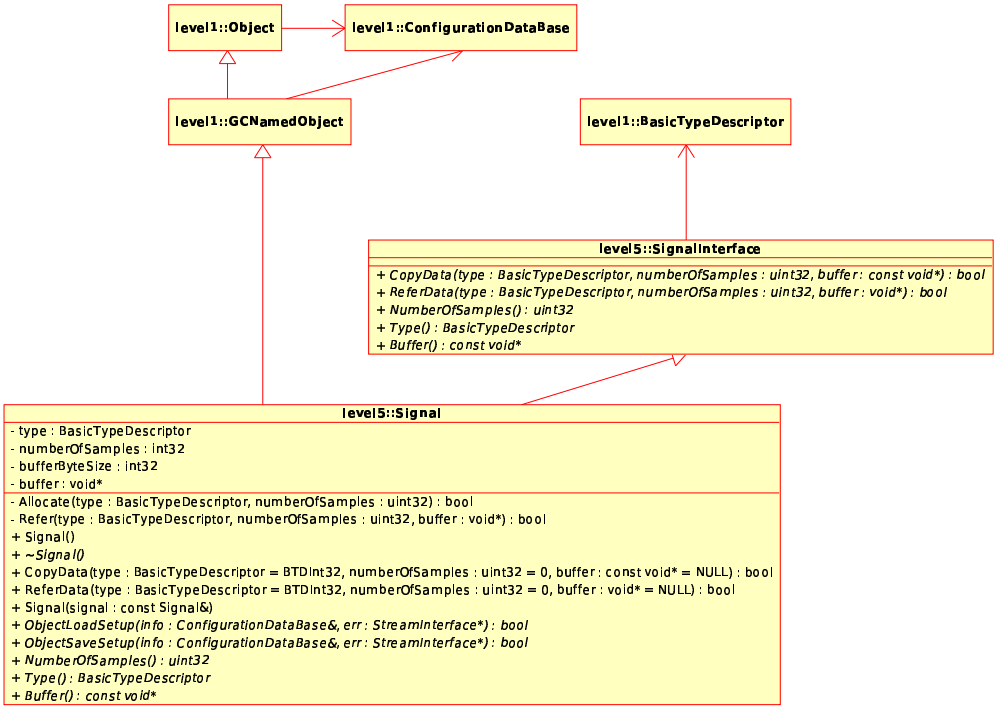

Figure 6.8: BaseLib Level 5 Signal interface classes

SingalInterface

[SignalInterface.h]

Signal

`[Signal.h, Signal.cpp]`

6.5 Dynamic Data Buffer

The Dynamic Data Buffer (DDB) is a memory data bus. Different entity could connect to such bus and as happens in an hardware bus each entity could write and read from other entity connected to the bus, the DDB infrastructure let such entity simply communicate providing a sort of shared memory buffers. This is not implemented like a shared memory in an Operating System way. It is not the same concept of the Memory Data Bus developed in the QNX© microkernel architecture. Refer to the following Figure 6.9 to visualize what a DDB is an looks like.

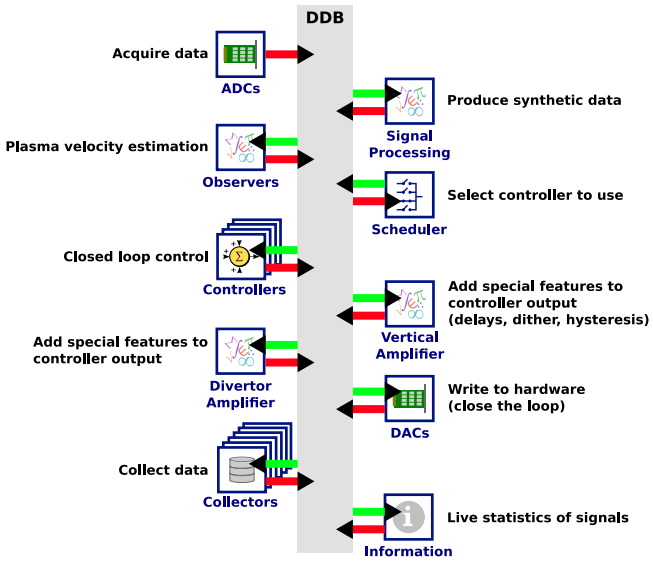

Figure 6.9: BaseLib Level 5 DDB setup for the Vertical Stabilization ver 5.0

Each entity connected to the described memory bus is called Generic Application Module (GAM), in Figure 6.9 there are different GAMs: ADC, Signal Processing, Observer, Scheduler, Controller, Vertical Amplifier, Divertor Amplifier, DAC, Collector, Information; each GAM can be associated with a physical piece of hardware, like an ADC, or not. Data is transferred between GAMs using the Dynamic Data Buffer. The first role of such tool is to ensure coherency across the system, verifying if all the signals requested by each of the GAMs is produced by one module and in case of any inconsistency issue an error. Although it is not the default behavior, a GAM may write over a signal already produced by another module. This process, named patching, must be explicitly requested otherwise it is assumed as an error. Each GAMs can produce and/or consume data and so will be an output, input or IO GAM. Figure 6.10 illustrate a path of data between e producer GAM and an IO GAM.

The DDB is a really powerfull tool but the internal architecture is far away to be simple and use many underlying data structures that can be integrated in the class itself.

The Dynamic Data Buffer is basically a buffer, i.e. a piece of memory area, this memory is divided in a way that each signal can be written and readed in a smart way. If we think about a control system, this system is usually represented by blocks (like in Matlab© Simulink©) and each block has many connection points that are named *interfaces*. Each interface loop togheter many *signals*. A Generic

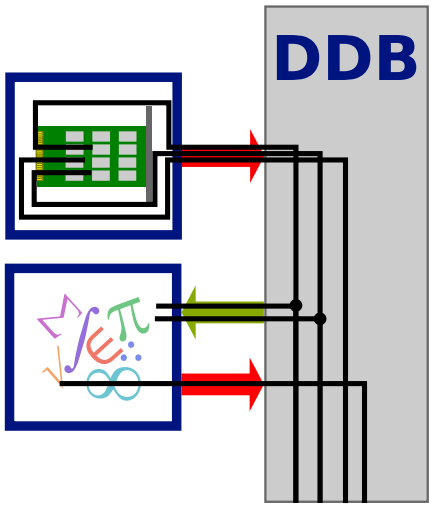

Figure 6.10: BaseLib Level 5 DDB setup for the Vertical Stabilization ver 5.0 (detail view)

Application Module (GAM) is a block.

A GAM can be an IO device as well as an algorithm, anyway a GAM is a set of interfaces (class `DDBInterface`) and an interface is a set of signals (or signal descriptors, class `DDBSignalDescriptor`). Each signal is distinguished by a name, a basic type and a set of storing properties, thanks to the DDBInterface to each signal a memory buffer can be associated. In Figure 6.11 the UML/logic diagram of the interfaces is depicted, in Figure 6.12 the GAM internals are depicted.

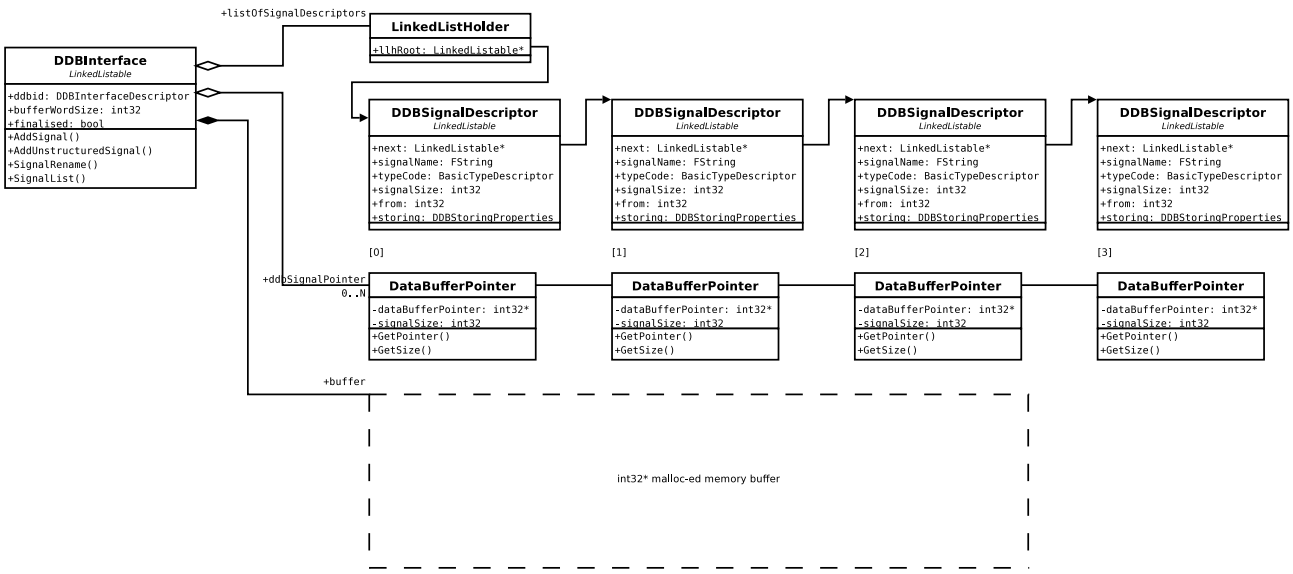

Figure 6.11: BaseLib Level 5 DDB, DDBInterface UML/logic scheme

A DDB take as inputs some GAMs, then analizes its connection and creates the buffers assigning to each signal a predeterminated space (that an interface has asked for), when the DDB is created the DDB can be used. GAMs are tightly bounded with the DDB, infact they were designed togheter. Steps to make a DDB from a set of GAMs, i.e. connecting GAMs togheter, are:

1. create the `GAM`s (i.e. via the CDB)
2. for each `DDBInterface` in each `GAM` call `DDB::AddInterface`

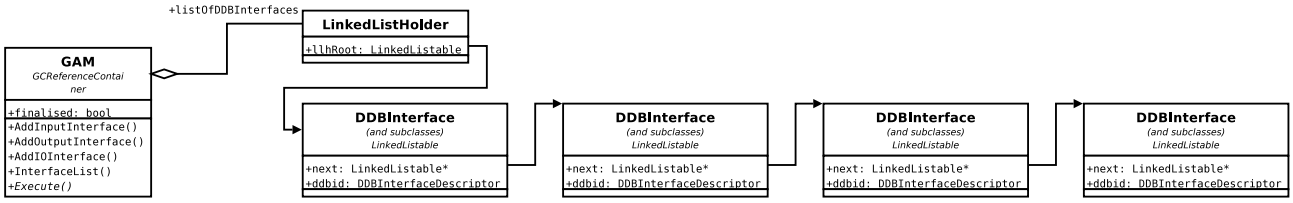

Figure 6.12: BaseLib Level 5 DDB, GAM UML/logic scheme

3. a `DDBItem` will be created for each `DDBSignalDescriptor`

4. then call `DDB::CheckAndAllocate`, this method compute each signal offset in the DDB buffer and check properties of different memory areas

5. for each GAM call `DDB::CreateLink`, every `DataBufferPointer` (in each `DDBInterface`) will be filled with the correct pointers.

6. the DDB is ready to be used.

To summarize the insights of the DDB we can make the following reasonment: each GAM represent a block in a control scheme, in each block there are some or any signals as input and some or any signals as output, some input signals can be control signal as well as the output ones. Each signal is grouped togheter to any other following some properties rule and others, i.e. each group of signal has a logical meaning. Each interface of a block can be connected to one or other interfaces of other blocks, and then the signals can flow between interfaces.

When the DDB is finalised the basic piece of information that flows between blocks (GAMs) is the signal data. Basically what the DDB does is to create a map between each signal i.e. it creates a list of all signals involved at runtime and gives to each producer/consumer the same address of that signal allocating once the space for it. So first the users work with GAMs that are block centric and uses many interface that contains signals then when it want that those blocks communicate creates a DDB that is signal centric and let blocks transfer the data throught it.

Everytime a new GAMs is added to the DDB each signal not just added to the DDB is added as a `DDBItem`, on `DDB::CheckAndAllocate` every connection is checked about rights and the summ of the total memory space to allocate is done.

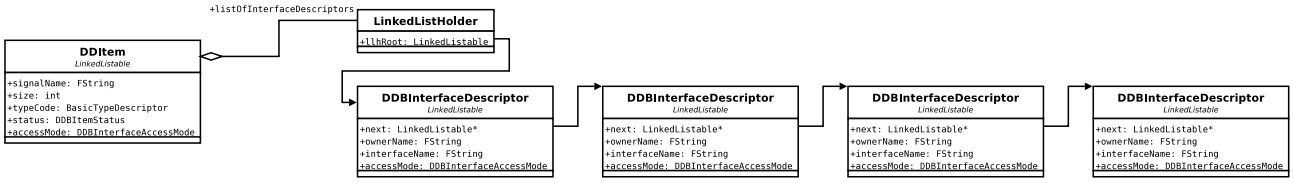

Figure 6.13: BaseLib Level 5 DDB, DDBItem UML/logic scheme

In Figure 6.14 it is possible to see that each `DDBItem` is associated to a signal and a DDB is a collection of `DDBItem` (that are collection of `DDBInterfaceDescriptor`). During the `DDB::CheckAndAllocate` links between each `DDBItem` and the memory area buffer are created (i.e. in Figure 6.14 the bezierline's arrows between a `DDBItem` and the `buffer`. Using `DDB::CreateLink` each one of this information is copied from the correct `DDBItem` to the `DataBufferPointer` of a GAM.

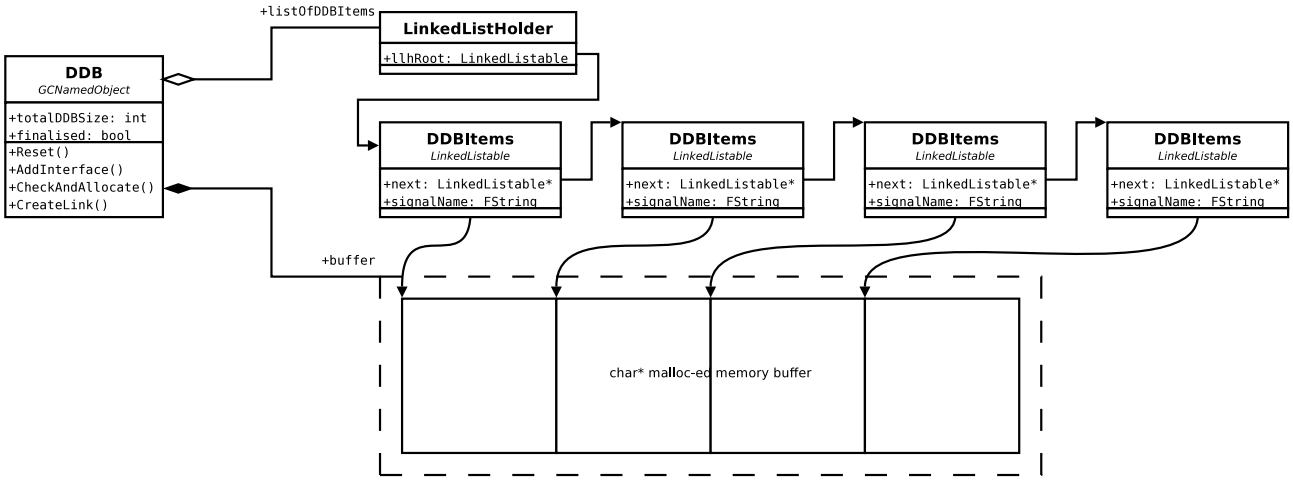

Figure 6.14: BaseLib Level 5 DDB, Dynamic Data Buffer UML/logic scheme

Once the DDB is ready to use, i.e. each data buffer is correctly assigned, the DDB class is used only to deallocate all the memory space at the end of the program. All the data passing is done by the GAM itself. The DDB bus is used only to allocate and check the memory.

In Figure 6.15 a whole representation of all the connection between classes in this section is depicted, the UML schema is not exhaustive.

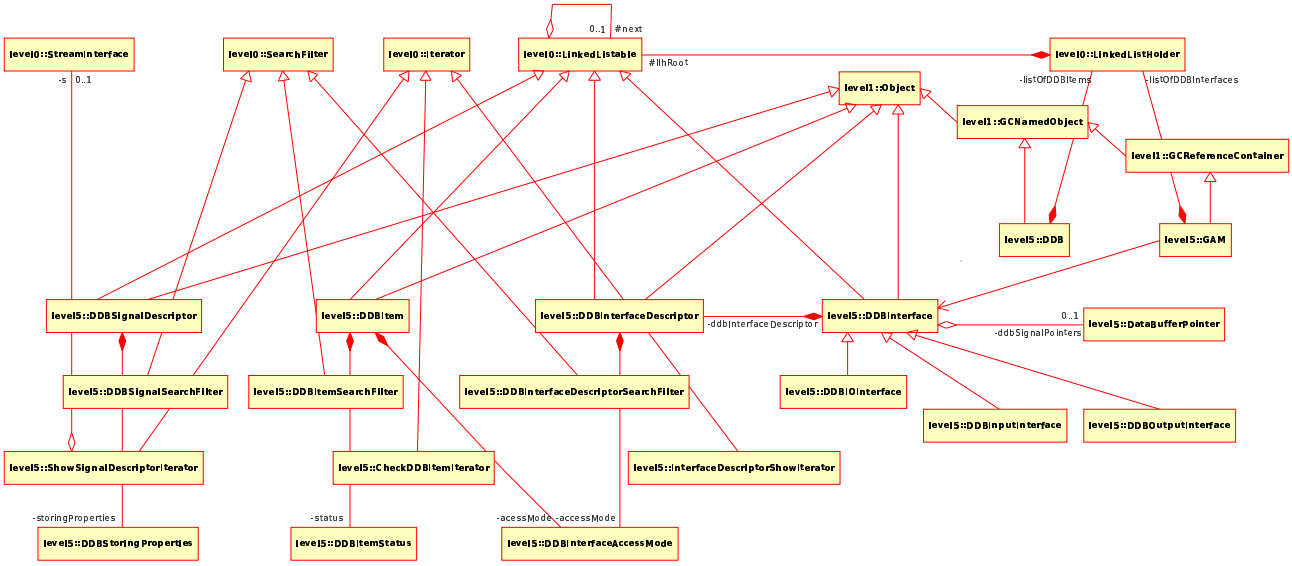

Figure 6.15: BaseLib Level 5 DDB

In the following all classes are grouped in two, the first set are classes involved with `GAM`s and `DDBInterface`s and the second group examines the DDB internals i.e. the `DDBItem` and the `DDB` itself.

6.5.1 GAM, DDBInterface

A GAM is a block of code implementing a specific interface specified in the BaseLib2 library. Each GAM contains three communication points: one for configuration and two for data input and output. The core of a typical GAM, processes the input accordingly to how it was configured and outputs the

modified information. The only entry point which is mandatory is the configuration and some GAMs, for instance the ones which interface hardware, are read or write only.

During initialization the modules declare what data they expect to receive and what information is going to be produced in output. Each type of input or output is declared as a configurable named signal with a data type associated. This is the only available way to chain GAMs and provides a clear boundary in the system: GAMs are not aware of other modules. Although GAMs are unrestricted in functionality, the set of GAMs which interface with hardware can be conceptually extracted. These kind of modules are named IOGAMs and provide a unique high level interface to any kind of hardware. The connection between the IOGAM and the specific low level code responsible for driving the hardware is performed through the specialization of a high level class named generic acquisition module. This interface requires the number of hardware inputs and outputs to be specified and forces the existence of a reading and write function. These functions must be implemented for each kind of devices, although it is common that devices belonging to the same family are able to share a common acquisition module. These high level interfaces to the hardware can usually be configured to return the latest acquired value or to wait for a new sample to arrive (at the cost of delaying the execution), depending on the requirements of the application. The most common type of GAMs allow to interface with hardware, store acquired data, execute algorithms, take decisions based on the current state of the system, debug internal states and provide live information about the system.

Classes involved in this section are depicted in Figure 6.16 and listed below:

- DDBStoringProperties
- DDBSignalDescriptor, ShowSignalDescriptorIterator, DDBSignalSearchFilter
- DDBInterfaceAccessMode
- DDBInterfaceDescriptor, InterfaceDescriptorShowIterator, DDBInterfaceDescriptorSearchFilter
- DataBufferPointer
- DDBInterface
- DDBIOInterface, DDBInputInterface, DDBOutputInterface
- GAM

DDBStoringProperties

`[DDBDefinitions.h, DDBDefinitions.cpp]`
The class `DDBStoringProperties` describes the way data are stored in the DDB. All the possible properties and attributes are stored in the single class's attribute `properties`, this value codes the property mask.

Constructors lets user make a `DDBStoringProperties` from scratch, from an integer and from another `DDBStoringProperties`. There are some assignement operators and operators ridefinitions.

The method `CheckMask` check the object with the passed by object; `Print` prints on the `StreamInterface` passed by argument the `properties` attribute in a human readable form.

```
private:
   int32 properties;

public:
   DDBStoringProperties();
   DDBStoringProperties(const int intProperties);
   DDBStoringProperties(const DDBStoringProperties& reference);

   DDBStoringProperties& operator=(const DDBStoringProperties right);
```

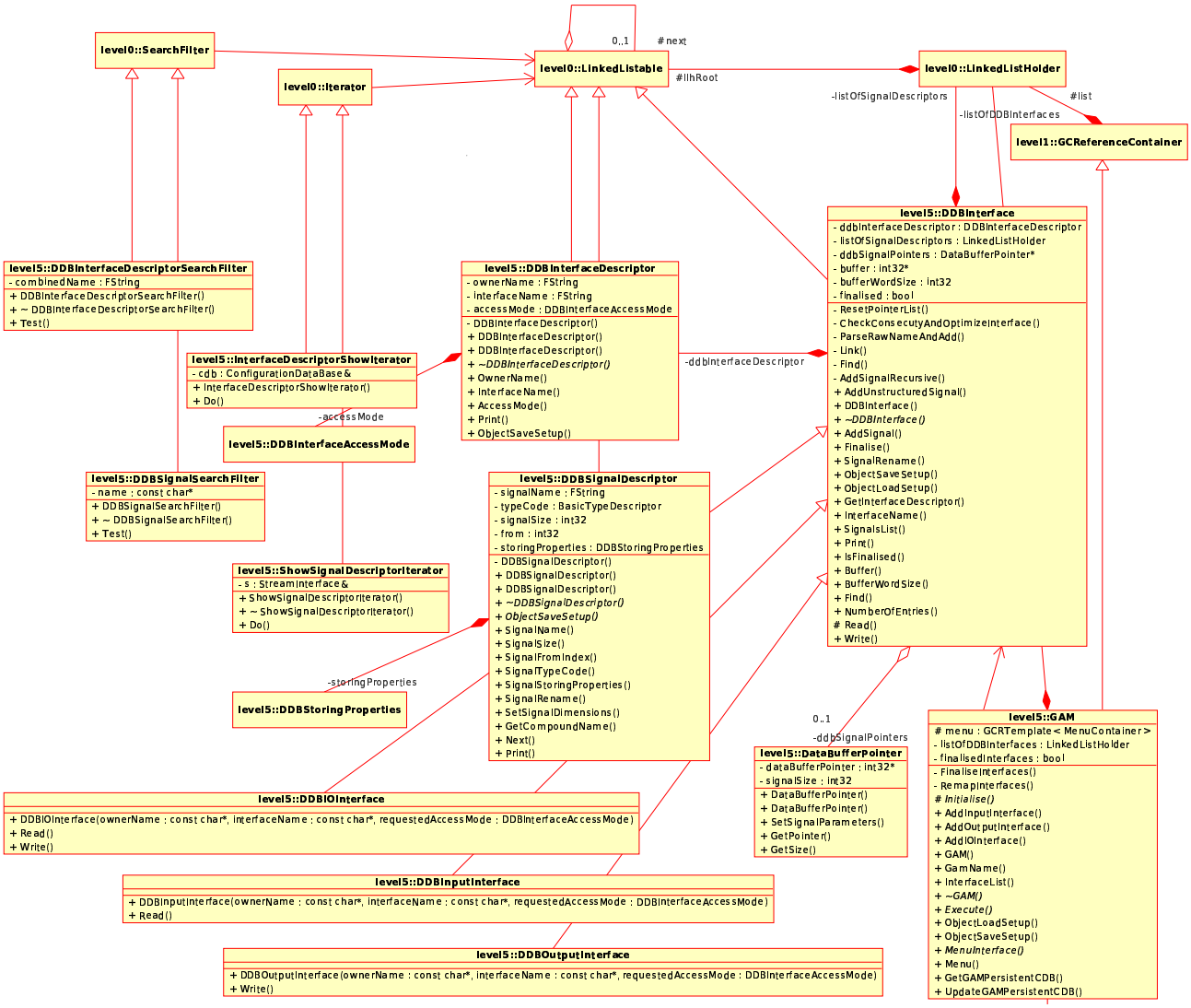

Figure 6.16: BaseLib Level 5 DDB Interface

```
DDBStoringProperties& operator|=(const DDBStoringProperties right);
DDBStoringProperties& operator&=(const DDBStoringProperties right);
DDBStoringProperties operator~() const;
bool operator==(const DDBStoringProperties& reference) const;

bool CheckMask(const DDBStoringProperties& mask) const;
void Print(StreamInterface& s) const;
```

Declared `DDBStoringProperties` in BaseLib are the following (also declared in *DDBDefinitions.h* as `const`):

type	value	description
`DDB_Default`	0x00	The signal structure is inserted without using fully qualified names.
`DDB_FlatNamed`	0x01	The signal structure is inserted without using fully qualified names.
`DDB_Consecutive`	0x02	The DDB tries to allocate the signal just next to the previous.
`DDB_Unsized`	0x04	The dimensions of signal are not fully specified.

DDBSignalDescriptor

[DDBSignalDescriptor.h, DDBSignalDescriptor.cpp]
This class is used to represent a signal. It inherits from `Object` and `LinkedListable`. The most important attribute is the name of the signal (`signalName` attribute) of type `FString`; `typeCode` codes the type and size of the elementary type used to build this signal.

`signalSize` is the actual signal size if the `from` attribute is -1; if it is different this attribute specifies the number of consecutive elements to be read. The attribute `from` if its different from -1 specifies the starting element from which to start reading. `storingProperties` specifies the storing properties of the signal, i.e. `DDB_Default`, `DDB_FlatNamed`, `DDB_Consecutive` and `DDB_Unsized`.

```
private:
   FString signalName;
   BasicTypeDescriptor typeCode;
   int32 signalSize;
   int32 from;
   DDBStoringProperties storingProperties;
```

The standard contructor has been overridden to avoid silent use; the second constructor is for partially sized signals; this constructor sets the `DDB_Unsized` bit overriding the setting made by the user. Then follows the copy contructor and the destructor that just frees the memory allocated during construction.

The method `ObjectSaveSetup` is the `DDBSignalDescriptor` save function; it uses a CDB to pass the initialisation parameters; the CDB information is read from the subtree that is currently addressed.

Next five methods get signal attributes, the first one returns the name of the signal (`SignalName`); the method `SignalSize` gets the number of elements in the signal. `SignalFromIndex` gets the starting index of the signal. `SignalTypeCode` returns the type code of the signal and `SignalStoringProperties` gets the storing properties of the signal. To get the size in byte of the signal the user has to code:

```
sizeof_theSignal = theSignal.SignalSize()*TypeSize(theSignal.SignalTypeCode());
```

There are also some setter methods: `SignalRename` that sets the name of the signal and `SetSignalDimensions` that sets the number of `maxDimensions` of the signal.

The class ends with some utility functions; the method `GetCompoundName` returns the signal name with added informations, if the signal is unsized it returns the element/s to be used withing round brackets, if the signal is sized it returns the signal name with the dimension within square brackets. `Next` calls `LinkedListable::Next` and `Print` simply prints the content.

```
   DDBSignalDescriptor();
public:
   inline DDBSignalDescriptor(const char* signalName,const int signalSize,
       const int from,const BasicTypeDescriptor typeCode,const DDBStoringProperties& ddbss);
   inline DDBSignalDescriptor(const DDBSignalDescriptor& descriptor);
   virtual ~DDBSignalDescriptor();

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err) const;

public:
   const char* SignalName() const;
   int32 SignalSize() const;
   int32 SignalFromIndex() const;
   BasicTypeDescriptor SignalTypeCode() const;
   DDBStoringProperties SignalStoringProperties() const;

   void SignalRename(const char* newSignalName);
   void SetSignalDimensions(const int signalSize);

   bool GetCompoundName(FString& compoundName);
   DDBSignalDescriptor* Next() const;
```

```
   void Print(StreamInterface& s)const;
```

Remember from the introduction of this section that a `DDBSignalDescription` is the description of a signal and is not directly bounded to signal data itself, data and pointers to signal data are accounted by an associated `DataBufferPointer` structure.

ShowSignalDescriptorIterator

`[DDBSignalDescriptor.h]`
This is the iterator for a list of `DDBSignalDescriptor`s. It inherits from `Iterator` and simple iterate through `DDBSignalDescriptor`s and prints its content on the `StreamInterface` attribute.

This function uses the C++'s dynamic cast operator.

```
private:
   StreamInterface& s;

public:
   ShowSignalDescriptorIterator(StreamInterface& si);
   virtual ~ShowSignalDescriptorIterator();

   void Do(LinkedListable* ll);
```

DDBSignalSearchFilter

`[DDBSignalDescriptor.h]`
This class is a search filter used to find a `DDBSignalDescriptor` by name in a list of `DDBSignalDescriptor`s (for example `DDBInterface::listOfSignalDescriptors`). It inherits from `SearchFilter`.

The single attribute that must be initialized by the constructor is the signal name to be searched.

```
private:
   const char* name;

public:
   DDBSignalSearchFilter(const char* name);
   virtual ~DDBSignalSearchFilter();

   bool Test(LinkedListable* ll);
```

DDBInterfaceAccessMode

`[DDBDefinitions.h, DDBDefinitions.cpp]`
The class `DDBInterfaceAccessMode` defines the the DDB access mode. The structure actually contains two conceptually different variables: the first is a code for the type of `DDBInterface` (as we will see there are input, output and IO DDB interfaces) used by its owner (and it really is an access mode to the DDB); the second represent the state of the DDB while adding interfaces and is used to detect errors. Users can access the DDB using five different interfaces or modalities:

- a placeholder.
- As a reader.
- As a writer.
- As a patcher.
- As an exclusive writer.

TODO
TODO
TODO
spiegare meglio le differenze

These access modes are encoded by a number of bits in a `DDBAccessMode` variable as shown in the following table.

Access Type	R	W	P	E
Placeholder	0	0	0	0
Reader	1	0	0	0
Writer	0	1	0	0
Patcher	0	0	1	0
Exclusive Writer	0	1	0	1

Different users however (and also a single user), can access the same signal using different interfaces provided that they are compatible. Below it is shown the table of the compatible interfaces.

Access Type	R	W	P	E
Reader and Writer	1	1	0	0
Reader and Exclusive Writer	1	0	0	1
Reader and Patcher	1	0	1	0
Writer and Patcher	0	1	1	0
Reader, Writer and Patcher	1	1	1	0

All of the other bits combinations are not valid. In particular it is worth noticing that if `E` is set also `W` is set. In a `DDBItem` object, where a list of users of the same signal is managed, the following access rules apply:

- Either there must be a user having the Writer access to the signal or all the users being Placeholders.
- No more than one user can have the Writer or Exclusive Writer access to that signal.
- If the writer has has the Exclusive Writer access the other users can't get the Patcher access.

The class `DDBInterfaceAccessMode` live around a single attribute `mode` that codes the access mode. There are three constructors: the standard constructor initialises to `DDB_PlaceholderMode`, a constructor from integer and the copy constructor. The assignment operator, the bitwise or, and, not and the equal operator are defined to make operations easier.

The method `CheckMask` compare the object itself with the object passed by. `ObjectSaveSetup` saves parameters to CDB.

```
   int mode;

public:
   DDBInterfaceAccessMode();
   DDBInterfaceAccessMode(const int intMode);
   DDBInterfaceAccessMode(const DDBInterfaceAccessMode& reference);

   DDBInterfaceAccessMode& operator=(const DDBInterfaceAccessMode& right);
   DDBInterfaceAccessMode& operator|=(const DDBInterfaceAccessMode& right);
   DDBInterfaceAccessMode& operator&=(const DDBInterfaceAccessMode& right);
   DDBInterfaceAccessMode operator~() const;
   bool operator==(const DDBInterfaceAccessMode& reference) const;
```

```
bool CheckMask(const DDBInterfaceAccessMode& mask) const;

void Print(StreamInterface& s) const;

bool ObjectSaveSetup(ConfigurationDataBase& info, StreamInterface* err)const;
```

In BaseLib are defined the following `DDBInterfaceAccessModes`:

type	value	description
`DDB_PlaceholderMode`	0x00	The value of the Placeholder.
`DDB_ReadMode`	0x01	The value of the ReadMode.
`DDB_WriteMode`	0x02	The value of the WriteMode.
`DDB_PatchMode`	0x04	The value of the PatchMode.
`DDB_ExclusiveWriteMode`	0x0a	The value of the WriteExclusive.
`DDB_ExclusiveWriteBit`	0x08	The mask for the value of the bit WriteExclusive.

DDBInterfaceDescriptor

`[DDBInterfaceDescriptor.h, DDBInterfaceDescriptor.cpp]`
The class `DDBInterfaceDescriptor` inherits from `LinkedListable` and `Object`. Implements a user of the DDB.

The attribute `ownerName` is the name of the user (GAM). In this context user means both GAMs for which the signal is an input and those for which the signal is an output. The attribute `interfaceName` is the name of the `DDBInterface` holding the class. `accessMode` specifies the `DDBInterface` access mode to the signals.

```
private:
   FString ownerName;
   FString interfaceName;
   DDBInterfaceAccessMode accessMode;
```

The standard constructor is disabled, i.e. its private. The next constructor is the only one that can be safely called in the constructor of a derived class to initialise the parent, then the copy constructor follows. With this trick it is possible to create a new DDBInterfaceDescriptor only copying it from a derived class, or creating it from scratch with all the parameters.

Some getter methods follows that let you know the name of the owner of the interface, the name of the interface itself and the access mode.

```
   DDBInterfaceDescriptor();

public:
   DDBInterfaceDescriptor(const char* ownerName,const char* interfaceName,
      const DDBInterfaceAccessMode accessMode);
   DDBInterfaceDescriptor(const DDBInterfaceDescriptor& descriptor);
   virtual ~DDBInterfaceDescriptor();

   const char* OwnerName() const;
   const char* InterfaceName() const;
   const DDBInterfaceAccessMode& AccessMode() const;

   void Print(StreamInterface& s) const;

   bool ObjectSaveSetup(ConfigurationDataBase& info, StreamInterface* err);
```

InterfaceDescriptorShowIterator

[DDBInterfaceDescriptor.h]
The class InterfaceDescriptorShowIterator implements the ObjectSaveSetup method for all members of a DDBInterfaceDescriptor's list. It inherits from Iterator. It works on a CDB, infact the only attribute is a reference to a ConfigurationDataBase.

```
private:
   ConfigurationDataBase& cdb;

public:
   InterfaceDescriptorShowIterator(ConfigurationDataBase& cdbi);
   void Do(LinkedListable* ll);
```

DDBInterfaceDescriptorSearchFilter

[DDBInterfaceDescriptor.h]
This is an ancillary class used to serch a DDBInterfaceDescriptor's list. Class DDBInterfaceDescriptorSearchFi
inherits from SearchFilter. The only attribute combinedName is the identifier to look for.

```
private:
   FString combinedName;

public:
   DDBInterfaceDescriptorSearchFilter(const char* gamName,const char* interfaceName);
   virtual ~DDBInterfaceDescriptorSearchFilter();

   bool Test(LinkedListable* ll);
```

DataBufferPointer

[DDBInterface.h, DDBInterface.cpp]
The class DataBufferPointer containes the int32 pointer to the memory where the signal is stored. It contains also the signalSize, where the size is the number of int32 elements that are to be read. A DataBufferPointer is associated to a single DDBSignalDescriptor in a DDBInterface.

A DataBufferPointer can be set via the constructor (the second) or via the SetSignalParameters method. The last two methods return the attributes.

```
private:
   int32* dataBufferPointer;
   int32 signalSize;

public:
   DataBufferPointer();
   DataBufferPointer(int32* dataBufferPointer,int32 signalSize);
   void SetSignalParameters(int32* dataBufferPointer,int32 signalSize);

   int32* GetPointer() const;
   int32 GetSize() const;
```

DDBInterface

[DDBInterface.h, DDBInterface.cpp]
Many DDBInterfaces build up a single GAM. A DDBInterface is build around the access method is being used to collect data. Signals with the same access mode should be grouped and accessed in a unique operation for efficiency. It inherits from Object and LinkedListable.

The attribute ddbInterfaceDescriptor describes the properties of the interface, i.e. the interface name, the GAM owner and the access mode to the signals in the DDB. The attribute listOfSignalDescriptors is a list of DDBSignalDescriptors, this list should be made by all the signals accessed by the user with the specified access mode. The attribute ddbSignalPointers is an array of pointers to specific DDB signals. buffer is a pointer to the working buffer of this interface. buffeWordSize is the size in int32 of buffer. The last attribute, finalised, is true when the DDBInterface can be linked to the DDB.

```
private:
   DDBInterfaceDescriptor ddbInterfaceDescriptor;
   LinkedListHolder listOfSignalDescriptors;
   DataBufferPointer* ddbSignalPointers;
   int32* buffer;
   int32 bufferWordSize;
   bool finalised;
```

The method ResetPointerList resets the pointer list. It should only be called by a DDB class to allow reinitialization. The method CheckConsecutyAndOptimizeInterface should be called by the DDB after having added the interface; it looks into the ddbSignalPointers array for consecutive elements; if consecutive elements are found, only one pointer with an increased number of elements is used; return True if everything goes well False if the interface has not been finalized. The method ParseRawNameAndAdd checks if the signal name has the interval extension as in the rawName parameter; if the signals name is specified with square brackets thenction assumes that the number specified within brackets is the signals size (i.e. A[5] is a signal of name A of size 5). If the signal is specified with round brackets the function assumes that the numbers specify the elements that are to be used; valid sintax are A(4), A(1,3,4,6), A(4:6) and A(2:4,7:9). If the user specifies the signal with round brackets the storing properties of the signal are set to DDB_Unsized. Only interface with read access can use the range specifications.

The method Link is used to initialise the pointers to the DDB buffer. This function has to be called from the DDB who is the only one entity that can assign meaningful pointers and sizes, the parameter signalName is the name of the signal to link, ddbSignalPointer is the pointer of the signal in the DDB, it returns True if it has been successfull, False otherwise. The method Find finds by name a signal in the listOfSignalDescriptors starting from index element in the list, returns the descriptor which matches the signalName. The method AddSignalRecursive adds structured signals to the DDBInterface. The method AddUnstructuredSignal adds basic type signals to the DDBInterface.

```
   bool ResetPointerList();
   bool CheckConsecutyAndOptimizeInterface();
   bool ParseRawNameAndAdd(const char* rawName,const char* signalType,
      DDBStoringProperties& properties);

   bool Link(const char* signalName,int32* ddbSignalPointer);
   DDBSignalDescriptor* Find(const char* signalName,uint32& index);
   bool AddSignalRecursive(const char* signalName,const char* signalType,
      int signalSize,int from,DDBStoringProperties& properties);
   bool AddUnstructuredSignal(const char* signalName,const int signalSize,
      const int from,const BasicTypeDescriptor typeCode,DDBStoringProperties& properties);
```

Then comes a constructor, parameter ownerName is the name of the GAM that owns the DDBInterface, interfaceName is the name of the interface and requestedAccessMode specifies the access mode of the interface to the DDB. A destructor follows.

The public method AddSignal adds a structured type signal, param signalName is the name of the signal, signalType is the type of the signal and param properties specifies the storing properties of the signal. The method Finalise allocates memory for the DDB read and/or write operations and put the interface in a status which stops accepting new signals.

The method `SignalRename` renames all occurences of signal "signalName" in the `listOfSignalDescriptors` list with the name "newSignalName". `ObjectSaveSetup` and `ObjectLoadSetup` saves the `DDBInterface` to a DDB and initialize the `DDBInterface`.

Then follows a great number of getter methods. The method `GetInterfaceDescriptor` returns the `DDBInterfaceDescriptor`; `InterfaceName` returns the interface name; `SignalsList` returns the first element in the `listOfSignalDescriptors` attribute; `Print` is the print method. The method `IsFinalised` checks if the DDB is finalised, `True` if its finalised, `False` otherwise; `Buffer` returns the `buffer` attribute; the method `BufferWordSize` returns the number of float/int32 signals that are stored in the buffer; if this method is called before the interface has been finalised it returns the buffer size of the added signal list. Otherwise it returns the buffer size, the returned value is greater or equal of the value returned from the `NumberOfEntries` method.

The method `Find` returns a pointer to a `DDBSignalDescriptor` of a signal identified by name "signalName" in the `DDBInterface`. The method `NumberOfEntries` returns the number of entries in the `listOfSignalDescriptors` list. This can be different from the BufferWordSize if any of the signal in the List has a size larger than one.

The last two protected methods are fast method to read and write to the data buffer.

```
public:
   DDBInterface(const char* ownerName,const char* interfaceName,
      DDBInterfaceAccessMode requestedAccessMode);
   virtual ~DDBInterface();

   bool AddSignal(const char* signalName,const char* signalType,
      DDBStoringProperties properties=DDB_Default);
   bool Finalise();
   bool SignalRename(const char* signalName,const char* newSignalName);

   bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);

   const DDBInterfaceDescriptor& GetInterfaceDescriptor();
   const char* InterfaceName() const;
   const DDBSignalDescriptor* SignalsList() const;

   void Print(StreamInterface& s) const;
   bool IsFinalised() const;

   int32* Buffer();
   int32 BufferWordSize() const;

   const DDBSignalDescriptor* Find(const char* signalName) const;
   int32 NumberOfEntries() const;

protected:
   inline void Read();
   inline void  Write();
```

DDBInputInterface

[DDBInputInterface.h]
The `DDBInputInterface` is one of the three class that inherits from `DDBInterface`; it represents an input interface, i.e. a data producer, infact the only implemented method is the `Read` method. This class must be extended from a user.

```
public:
   DDBInputInterface(
```

```
      const char* ownerName, const char* interfaceName,
      DDBInterfaceAccessMode requestedAccessMode) :
         DDBInterface(ownerName, interfaceName, requestedAccessMode){};

    inline void Read();
```

DDBOutputInterface

[DDBOutputInterface.h]
The DDBOutputInterface is one of the three class that inherits from DDBInterface; it represents an output interface, i.e. a data consumer, infact the only implemented method is the Write method. This class must be extended from a user.

```
public:
   DDBOutputInterface(
      const char* ownerName, const char* interfaceName,
      DDBInterfaceAccessMode requestedAccessMode) :
         DDBInterface(ownerName, interfaceName, requestedAccessMode){};

   inline void Write();
```

DDBIOInterface

[DDBIOInterface.h]
The class DDBIOInterface is the last class that inherits from DDBInterface. It implements the input and the output interface at the same time.

```
public:
   DDBIOInterface(
      const char* ownerName, const char* interfaceName,
      DDBInterfaceAccessMode requestedAccessMode) :
         DDBInterface(ownerName, interfaceName, requestedAccessMode);

   inline void Read();
   inline void Write();
```

GAM

[GAM.h, GAM.cpp]
This class is the interface of a Generic Acquisition Module (GAM) in the conceptual sense (i.e. not an interface in the OOP meaning). A GAM inherits from GCReferenceContainer. The first attribute is of type GCRTemplate<MenuContainer> and is the Menu Interface for the GAM, it is initialized after the FinaliseInterface method is called. The attribute listOfDDBInterfaces is a container of interfaces within the DDB. The attribute finalisedInterfaces accounts if the GAM is finalised or not.

```
protected:
   GCRTemplate<MenuContainer> menu;

private:
   LinkedListHolder listOfDDBInterfaces;
   bool finalisedInterfaces;
```

The method FinaliseInterfaces finalises all DDBInterfaces in the attribute listOfDDBInterfaces; it parses all interfaces, sets each interface in a state that does not allow adding or modifying signals, and allocates the memory for storing the signals, returns True if everything was fine; False otherwise. The method RemapInterfaces renames some or all signals in the DDBInterfaces to the value specified in the CDB; the code checks for the presence of a field "Remappings". If this field is found, the code checks in this subtree for an entry with the same name of one of the interfaces; if the interface name

is found, the code parses all the fields in the subtree. The name of the entry has to be the name of a signal contained in the interface, the value associated to the entry name is the new name that is used for the signal. Then follow an example of configuration file:

```
Remappings = {
  InputInterface = {
    OldSignalName = NewSignalName
    ...
  }
  OutputInterface = {
    OldSignalName = NewSignalName
    ...
  }
}
```

The method `Initialise` is called by the executor during the `ObjectLoadSetup` process, it initialises the GAM parameters and add interfaces to the list. The method `AddInputInterface` allows the user to add a `DDBInputInterface` in the list, the interface created will have only the `DDB_ReadMode` access and this cannot be changed. The method `AddOutputInterface` allows the user to add a `DDBOutputInterface` in the list, by default the interface has only the `DDB_WriteMode` access, anyway other access modes can be used with the exception of the read mode, the `DDB_ReadMode` bit is always cleared before construction. The method `AddIOInterface` allows the user to add a `DDBIOInterface` in the list, by default the interface has the `DDB_ReadMode` and `DDB_WriteMode` access, all the other access modes can freely be used.

```
   bool FinaliseInterfaces();
   bool RemapInterfaces(ConfigurationDataBase& cdbData);

protected:
   virtual bool Initialise(ConfigurationDataBase& cdbData)=0;

public:
   inline bool AddInputInterface(DDBInputInterface*& ddbi,const char* interfaceName);
   inline bool AddOutputInterface(DDBOutputInterface*& ddbo, const char* interfaceName,
      DDBInterfaceAccessMode accessMode=DDB_WriteMode);
   inline bool AddIOInterface(DDBIOInterface*& ddbio,const char* interfaceName,
      DDBInterfaceAccessMode accessMode=DDB_ReadMode|DDB_WriteMode);
```

The method `GamName` returns the name of the GAM, `InterfacesList` gets the list of interfaces used by the GAM. The method `Execute` is called by the executor (i.e. MARTe) when appropriate. The user can (has to) take different actions depending on the value of the parameter passed, values supperted are:

name	value	description
`GAMPrepulse`	0x00000001	Called once before going online.
`GAMOffline`	0x00000002	Called continuously while offline.
`GAMSafety`	0x00000003	Called continuously after a fault has been detected.
`GAMCheck`	0x00000004	Called once to verify if the parameter configuration is acceptable.
`GAMPostpulse`	0x00000005	Called once before going online.
`GAMStartUp`	0x00000006	Called once before going online.
`GAMOnline`	0x00010000	Called continuously when online after data has been read.
`GAMOnline2`	0x00010001	Called continuously when online after data has been written (but before writing the JPF). User defined functions starts from this offset.

User defined functions are GAMOnline3, GAMOnline4, etc.. . `ObjectLoadSetup` is the method actually called by the executor during the initialisation phase. `ObjectSaveSetup` saves the GAM parameters into the specified CDB.

The method `MenuInterface` implements the user menu interface, in this implementation simply returns `True`. `Menu` returns the `MenuContainer` class.

The method `GetGAMPersistentCDB` gets a CDB which can be used to store any values that will be maintained also when the GAM is reconfigured. If the CDB doesn't exist its create it; the CDB will be named *GAMPersistentCDB*. `UpdateGAMPersistentCDB` updates the *GAMPersistentCDB* with the requested values.

```
public:
   GAM();
   virtual ~GAM(){}

   const char* GamName() const;
   LinkedListable* InterfacesList()const;

   virtual bool Execute(GAM_FunctionNumbers functionNumber)=0;

   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   virtual bool MenuInterface(StreamInterface& in,StreamInterface& out,void* userData);
   GCRTemplate<MenuContainer> Menu();

   bool GetGAMPersistentCDB(ConfigurationDataBase& cdb);
   bool UpdateGAMPersistentCDB(ConfigurationDataBase& cdb);
```

6.5.2 DDB, DDBItem

The DDB is the main topic of this section and is a memory buffer pool that simply allocates memory and then is trasparent to each other mechanisms (i.e. the data transfer between buffers).

Classes involved in this section are depicted in Figure 6.17 and listed below:

- DDBItemStatus
- DDBItem, CheckDDBItemIterator, DDBItemSearchFilter
- DDB

DDBItemStatus

`[DDBDefinitions.h, DDBDefinitions.cpp]`
The class `DDBItemStatus` is basically a variable (attribute `status`) coding the status of a `DDBItem`. There are three constructors; the first one is the standard constructor that will be initialized the status to `OK` (or `DDB_NoError`), the second is a constructor from integer, unfortunately is not possible to avoid usign it to make the constants. The third is a copy constructor.

Assignment operator, bitwise operators and the equal operator are redefined. The method `Check` check the object with the mask. `Print` method prints the mask.

```
   int status;
public:
   DDBItemStatus();
   DDBItemStatus(const int intStatus);
   DDBItemStatus(const DDBItemStatus& reference);

   DDBItemStatus& operator=(const DDBItemStatus right);
   DDBItemStatus& operator|=(const DDBItemStatus right);
   DDBItemStatus& operator&=(const DDBItemStatus right);
   DDBItemStatus operator~() const;
   bool operator==(const DDBItemStatus& reference) const;

   bool CheckMask(const DDBItemStatus& mask) const;
   void Print(StreamInterface& s) const;
```

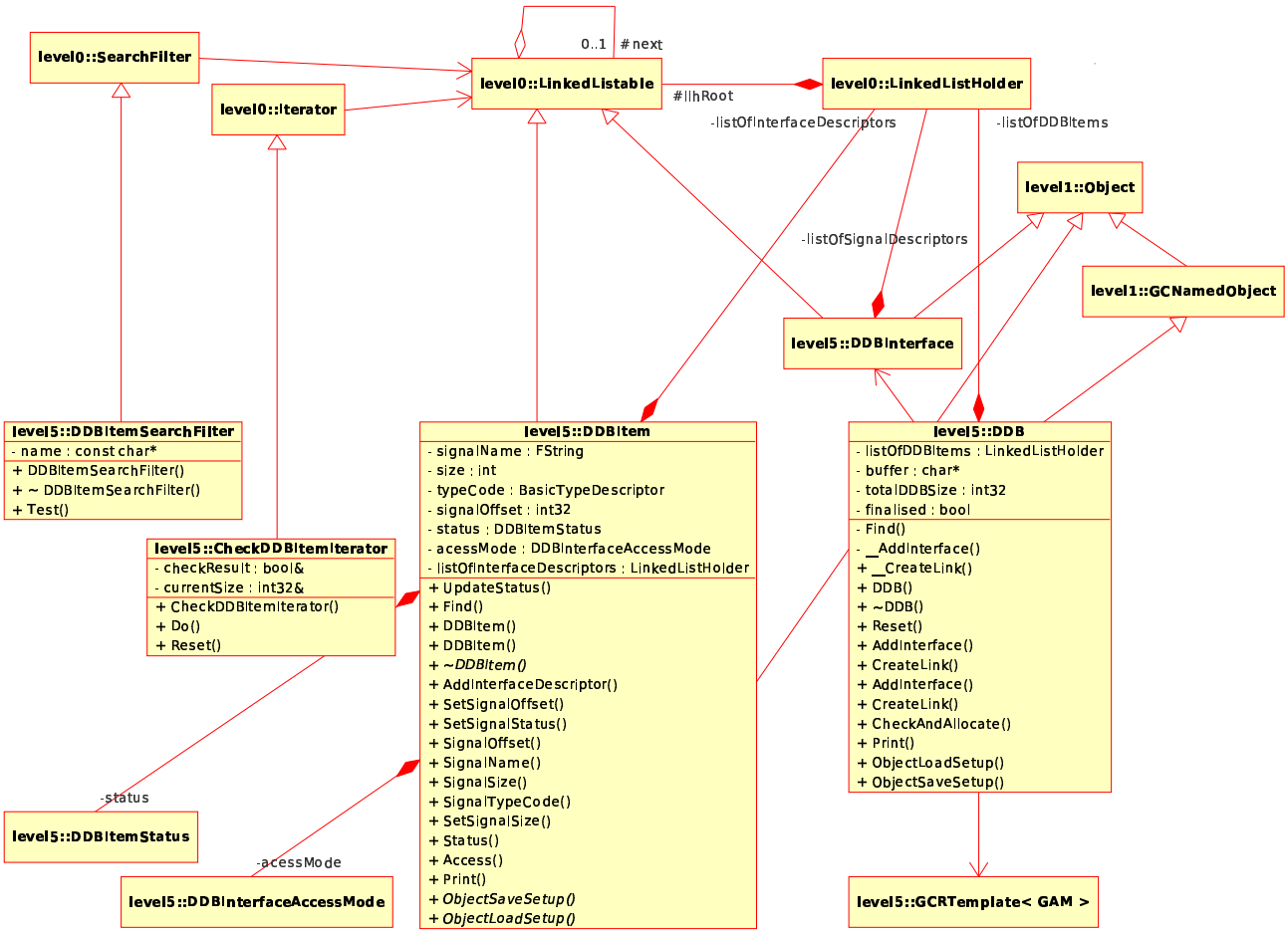

Figure 6.17: BaseLib Level 5 DDBItem

In BaseLib are defined the following `DDBItemStatus`:

type	value	description
`DDB_NoError`	0x00	The DDBItem status is OK.
`DDB_WriteConflictError`	0x01	More than one user set as Writer on the signal.
`DDB_WriteExclusiveViolation`	0x02	One or more user set as Patcher of a signal which is exclusively writte
`DDB_MissingSourceError`	0x04	The signal is read by a user but it is not written by any user.
`DDB_ConsecutivityViolation`	0x08	It has not been possible to arrange the items in the DDB as requested
`DDB_SizeMismatch`	0x10	Set when a reader of a signal try to access it beyond the limit specifie
`DDB_UndefinedSize`	0x20	Set when a signal is usized.
`DDB_TypeMismatch`	0x40	Set when a mismatch between type is found.
`DDB_UndefinedType`	0x80	Set when a mismatch between type is found.

DDBItem

`[DDBItem.h, DDBItem.cpp]`
This class imlements the abstraction of the signal descriptor from the DDB point of view. The name of the `DDBItem` is the same as the name of the signal. It inherits from `LinkedListable` and `Object`. The attribute `signalName` is obviously the name of the signal. `size` is the size of the signal, `typeCode` is a field coding the type and size of the elementary type used to build this signal; `signalOffset` is the

offset in bytes of the signal structure in the DBB. It will be set to 0 at construction time and updated by the proper function of the DDB class. The attribute `status` contains the status of the signal (i.e. if there are some errors..). The attribute `accessMode` contains the access mode of the signal (should be the or of the access modes of the DDB interfaces) and `listOfInterfaceDescriptors` is a list of users of the signal accounted by the `DDBItem`.

```
private:
   FString signalName;
   int size;
   BasicTypeDescriptor typeCode;
   int32 signalOffset;
   DDBItemStatus status;
   DDBInterfaceAccessMode accessMode;
   LinkedListHolder listOfInterfaceDescriptors;
```

The method `UpdateStatus` updates the status of the `DDBItem` after insertion of a new interface; the function checks whether there are some access violation and set the status accordingly. The method `Find` finds by name a user (i.e. a `DDBInterfaceDescriptor`) in the `listOfInterfaceDescriptors`.

The method `AddInterfaceDescriptor` if it is the first user sets the variables name and type otherwise it checks for compatibility, it may set the `WriteConflictError` bit. The method `SetSignalOffset` is basically used by the `AddInterface` method in DDB; `SetSignalStatus` sets the signal status to the desired value; previous values stored in the status are lost.

The method `SignalOffset` reads the `signalOffset` attribute; `SignalName` returns the `signalName`; `SignalSize` returns the signal size; `SignalTypeCode` returns the signal type code. The method `SetSignalSize` sets the signal size to the desired value. `Status` returns a copy of the `DDBItemStatus` attribute. `Access` returns a copy of the `DDBItemAccessMode` attribute. `Print` prints the object data.

The method `ObjectSaveSetup` uses a CDB to write the parameters, the CDB information is written to the subtree that is currently addressed. `ObjectLoadSetup` uses a CDB to pass the initialisation parameters, the CDB information is read from the subtree that is currently addressed.

```
private:
   void UpdateStatus(const DDBInterface& ddbInterface);
   DDBInterfaceDescriptor* Find(const DDBInterface& ddbInterface);

public:
   DDBItem(const char* signalName,int size,BasicTypeDescriptor typeCode);
   DDBItem(const DDBSignalDescriptor& descriptor);
   virtual ~DDBItem();

   bool AddInterfaceDescriptor(const DDBInterface& ddbInterface);
   void SetSignalOffset(int32 offset);

   void SetSignalStatus(DDBItemStatus status);
   int32 SignalOffset() const;
   const char* SignalName()const;

   int32 SignalSize();
   BasicTypeDescriptor SignalTypeCode() const;

   void SetSignalSize(int32 size);
   DDBItemStatus Status() const;
   DDBInterfaceAccessMode Access() const;
   void Print(StreamInterface& s, bool showUsers);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& cdb,StreamInterface* err);
   virtual bool ObjectLoadSetup(ConfigurationDataBase& cdb,StreamInterface* err);
```

CheckDDBItemIterator

[DDBItem.h, DDBItem.cpp]
The class CheckDDBItemIterator is the iterator for a list of DDBItem; after the iterator has been used, a call to the Reset method need to be made to use it again.

The iterator has two attibutes, the first, checkResult is the result of of the check on the DDBItem list, the second, currentSize is the size of the required DDB space during the iteration of the list.

```
private:
   bool& checkResult;
   int32& currentSize;

public:
   CheckDDBItemIterator(bool& checkResult, int32& currentSize);

   void Do(LinkedListable* ll);
   void Reset();
```

DDBItemSearchFilter

[DDBItem.h, DDBItem.cpp]
This class implements the search of a DDBItem by name. It inherits as usal from the SearchFilter class. The attribute name is the string to search for.

```
private:
   const char* name;

public:
   DDBItemSearchFilter(const char* name);
   virtual ~DDBItemSearchFilter();

   bool Test(LinkedListable* ll);
```

DDB

[DDB.h, DDB.cpp]
The class DDB is Dynamic Data Buffer class definition, the DDB provides a way to store the signals in memory and made data exchange in Real Time. It inherits from GCNamedObject.

The attribute listOfDDBItems is a list of signal descriptions (DDBItem objects). The attribute buffer is a pointer to the buffer memory, totalDDBSize is the size in byte of the DDB buffer. The attribute finalised is a flag specifying if the DDB has been finalised, when the DDB is finalised it cannot accept new DDBItems, the CheckAndAllocate method sets this flag if the DDB is coherent.

```
private:
   LinkedListHolder listOfDDBItems;

   char* buffer;
   int32 totalDDBSize;
   bool finalised;
```

The method Find finds a DDBItem by name in the listOfDDBItems list; the name of the DDBItem is the name of the signal has passed to the DDBInterface. The method __AddInterface is a private method that implements the public AddInterface method; the reason of this member is to allow access to private members of the DDBInterface from the exported functions. The method __CreateLink is a private member that implements the public CreateLink method, the reason of this member is to allow access to private members of the DDBInterface from the exported functions.

The method AddInterface is used to add an interface to the DDB. The method CreateLink can be used only if the DDB is finalised, a call to this method will initialise pointers (in the DataBufferPointer

object) in the specified interface. The second method `AddInterface` is used to add all the interfaces that are used by a GAM to the DDB. The second declared method `CreateLink`, used after the DDB is finalised, will initialise pointers to all interfaces in the specified GAM.

The method `CheckAndAllocate` checks the list of `DDBItem` for consistency and allocates the memory. The method `Print` prints the DDB information on the specified stream. The method `ObjectLoadSetup` loads the DDB parameters from the CDB and `ObjectSaveSetup` saves the DDB Parameters from the CDB.

```
   DDBItem* Find(const char* name);
   bool __AddInterface(const DDBInterface& gamInterface);
   bool __CreateLink(DDBInterface& gamInterface);

public:
   DDB();
   ~DDB();

   void Reset();
   bool AddInterface(const DDBInterface& gamInterface);
   bool CreateLink(DDBInterface& gamInterface);
   bool AddInterface(GCRTemplate<GAM> gam);
   bool CreateLink(GCRTemplate<GAM> gam);

   bool CheckAndAllocate();
   void Print(Streamable& s);

   bool ObjectLoadSetup(ConfigurationDataBase& cdb, StreamInterface* err);
   bool ObjectSaveSetup(ConfigurationDataBase& cdb, StreamInterface* err);
```

6.5.3 Design Notes

`ShowSignalDescriptorIterator` and `DDBSignalSearchFilter` (but also `InterfaceDescriptorShowIterator` `DDBInterfaceDescriptorSearchFilteral`) use `dynamic_cast` C++ operator that can throws exceptions, there is no code that catch those exceptions but only an if that checks against the zero value of this cast.

The following note is really important: using the GNU GCC compilers catching a `std::bad_cast` exception is possible only if the conversion is done by reference and not by pointers.

The DDB doesn't provide access control to the differents memory area it provides. The concurrent access control to the areas must be done at higher level (MARTe).

Looking at figure 6.18 you can see that GAM is a GCReferenceContainer, well, a GCReferenceContainer holds a LinkedListHolder as an attribute, GAM has another linked list holder as an attribute: GAM has two LinkedListHolder but it uses only one! On this LinkedListHolder it has DDBInterface elements that can be of input output or either.

Deepening in an implementation of a GAM when you use AddInputInterface (and others...) you are adding your Interface to the list and to another attribute you have to pass to the method. this attribute must be implemented in the new GAM.

Quindi crea un Linkedlist holder che tiene tutte le DDBinterface e ogni interfaccia deve pero essere puntata dall'interno della tua GAM di modo che ti obbliga cmq ad averne un controllo (per poterla anche usare nel codice). (penso che il puntatore all'interfaccia che hai nella classe non venga mai usato per avere array di interfaccie).

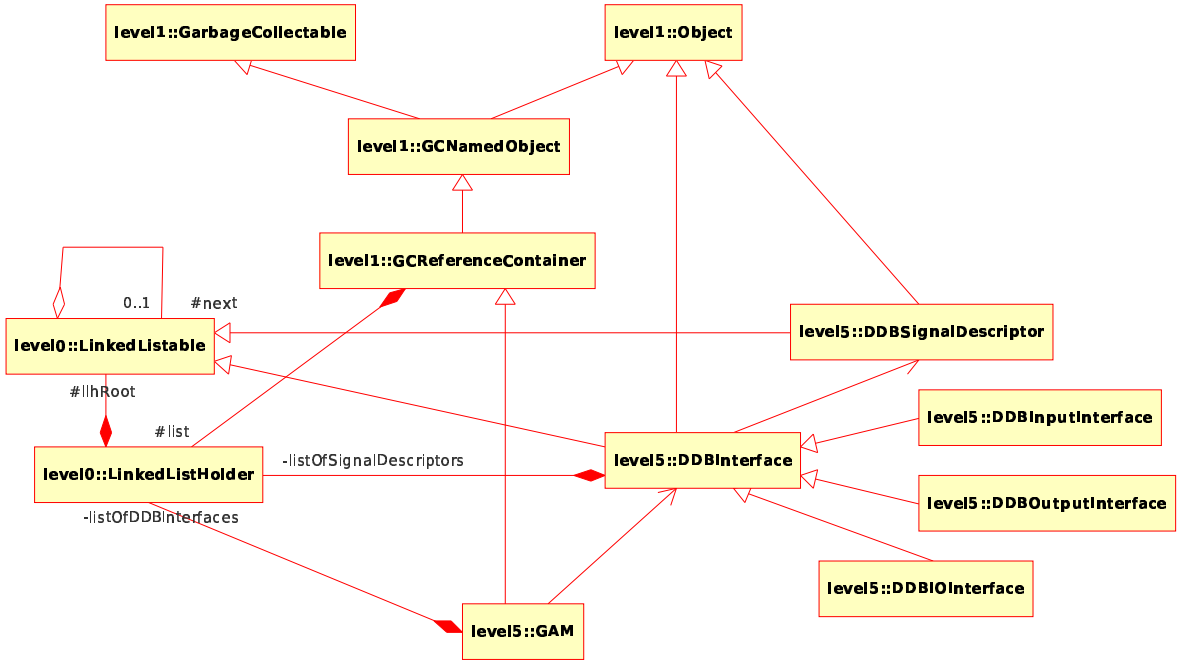

Figure 6.18: BaseLib Level 5 GAM hierical in the DDB

6.6 Menu

The menu infrastructure lets the user creating a menu structure with which is possible to navigate between menu entry and select a menu entry to execute. A menu structure can be created in C source code or via the wide used CDB. An excerpt of a menu created via the CDB follows. The example create a menu with two entry: one labelled "Abort" and another labelled "Inhibit". Each of those when selected sends a message.

```
+MARTeMenu = {
    Class = MARTeMenu
    Title = "MARTe␣Menu"
    +MenuA = {
        Class = MenuContainer
        Title = "CODAS␣Interface"
        +ABORT = {
            Class = SendMessageMenuEntry
            Title = Abort
            Envelope = {
               ...
            }
        }
        +INHIBIT = {
            Class = SendMessageMenuEntry
            Title = Inhibit
            Envelope = {
                Class = MessageEnvelope
                Sender = MARTeMenu
                Destination = StateMachine
                +Message = {
                    Class = Message
                    Code = 0x704
                    Content = Inhibit
                }
            }
        }
```

```
        ...
    }
    ...
}
```

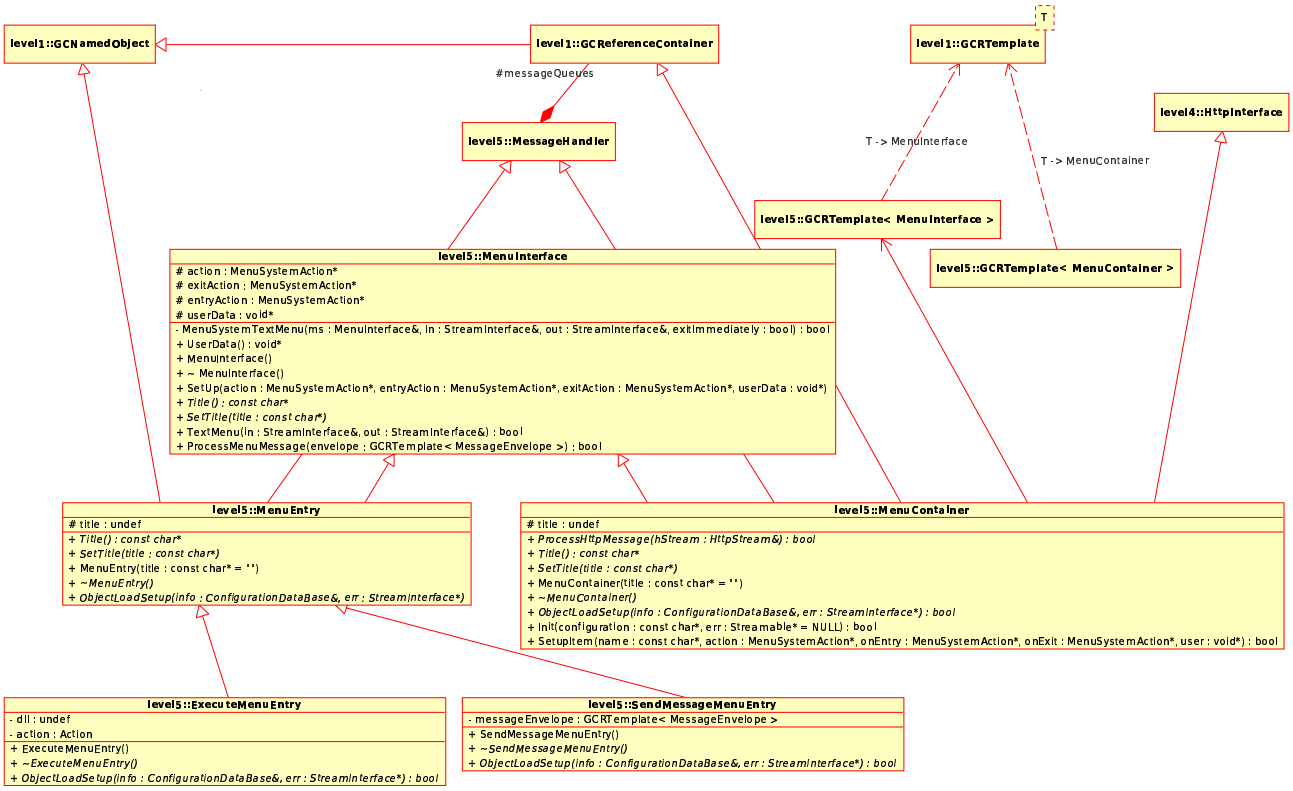

Figure 6.19: BaseLib Level 5 Menu

Classes involved in this section are depicted in Figure 6.19 and listed below:

- MenuInterface
- MenuEntry
- ExecuteMenuEntry
- SendMessageMenuEntry
- MenuContainer

MenuInterface

`[MenuInterface.h, MenuInterface.cpp]`
The class `MenuInterface` starts with a prototype for a function to associate to a menu page, it follows.

```
typedef bool (MenuSystemAction)(StreamInterface& in,StreamInterface& out,void* userData);
```

Such function takes as parameters an input straem, an output stream and a pointer to user data. The class `MenuInterface` is used to build a tree or menu pages that operate via the `StreamInterface`. The first `MenuSystemAction*` attribute define an action, if this is not NULL this is a menu container. The second attribute define an action to do when exiting and the last `MenuSystemAction*` let you set what to do before entering the menu item. The attribute `userData` contains user specific information,

that can be retrieved using the `UserData` method.

The method `SetUp` sets the action associated with this menu item, the pure virtual method `Title` is defined to be able to choose the labelling policy and returns the label of the menu that can be set with `SetTitle`. The method `TextMenu` calls the menu and `PRocessMenuMessage` is called by a class if it is also a `MessageHandler`.

```
protected:
   MenuSystemAction* action;
   MenuSystemAction* exitAction;
   MenuSystemAction* entryAction;

   void* userData;
public:
   void* UserData();

public:
   MenuInterface();
   virtual ~MenuInterface();

   void SetUp(MenuSystemAction* action,MenuSystemAction* entryAction,
      MenuSystemAction* exitAction,void* userData);
   virtual const char* Title()=0;
   virtual void SetTitle(const char* title)=0;

   virtual bool TextMenu(StreamInterface& in,StreamInterface& out);
   bool ProcessMenuMessage(GCRTemplate<MessageEnvelope> envelope);
```

MenuEntry

`[MenuEntry.h, MenuEntry.cpp]`
The class `MenuEntry` is used to build a tree o menu pages that operate via Streamable. It inherits from `MenuInterface`, `MessageHandler` and `GCNamedObject` so it is a menu part that can handle messages and it is garbage collectable and addressable by name.

It implements the pure virtual methods of `MenuInterface` infact it has one `BString` attribute as a `title`, distinct from the object name.

The method `ObjectLoadSetup` create a tree of `MenuContainer` and `MenuEntry` from a CDB, it search the CDB for `Title` entry.

```
protected:
   BString title;

public:
   virtual const char* Title()
   virtual void SetTitle(const char* title)

   MenuEntry(const char* title= "")
   virtual ~MenuEntry()

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
```

ExecuteMenuEntry

`[ExecuteMenuEntry.h, ExecuteMenuEntry.cpp]`
The class `ExecuteMenutEntry` starts defining the following type:

```
typedef bool (*Action)(StreamInterface& in,StreamInterface& out,void* userData);
```

Such class inherits from `MenuEntry` so has all the same capability and adds an action. The `LoadableLibrary` attribute is a user specified DLL containing the function to be executed and `action` is a pointer to the user specified action.

```
private:
   LoadableLibrary dll;
   Action action;

public:
   ExecuteMenuEntry();
   virtual ~ExecuteMenuEntry();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
```

SendMessageMenuEntry

[SendMessageMenuEntry.h, SendMessageMenuEntry.cpp]
The class SendMessageMenuEntry is a MenuEntry object that contains a MessageEnvelope used for sending a message. This MenuEntry class is able to react when selected sending a message.

The method ObjectLoadSetup calls the MenuEntry's ObjectLoadSetup and then loads the user specified MessageEnvelope according to the MessageEnvelope specifications. The MessageEnvelope must be specified within a member named Envelope.

```
private:
   GCRTemplate<MessageEnvelope> messageEnvelope;

public:
   SendMessageMenuEntry();
   virtual ~SendMessageMenuEntry(){}

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
```

MenuContainer

[MenuContainer.h, MenuContainer.cpp]
A MenuContainer class is a container of MenuEntry objects, it inherits infact from GCReferenceContainer; it inherits also from MenuInterface, MessageHandler and HttpInterface.

The only attribute, like MenuEntry, is title, i.e. a specific title, distinct from the object name; title can be returned using Title and setted by SetTitle method. The method ProcessHttpMessage is the main entry point for HttpInterface.

The method ObjectLoadSetup use the initialization syntax of GCReferenceContainer, additionally set the title; using the above mentioned syntax it is possible to create a tree of MenuContainer and MenuEntry that individually access the MenuEntry elements to set the user functions. The method Init creates a menu tree creating a stream first then a CDB and then using ObjectLoadSetup. SetupItem allows setting up the user functions and data for a specific sub menu, the menu is identified by the path to reach the specific menu item.

```
protected:
   BString title;

public:
   virtual const char* Title();
   virtual void SetTitle(const char* title);
   virtual bool ProcessHttpMessage(HttpStream &hStream);

   MenuContainer(const char* title="");
   virtual ~MenuContainer();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);

   bool Init(const char* configuration,Streamable* err=NULL);
```

```
bool SetupItem(const char* name,MenuSystemAction* action,MenuSystemAction* entryAction,
   MenuSystemAction* exitAction,void* userData);
```

6.6.1 Design Notes

A menu infrastructure is basically a tree where each leaf is an executable menu entry, i.e. if selected has associated a menu action to it. All this structure can be simplified implemented at lower level a tree data structure instead using list and sublists. This can lead to faster data structures.

6.7 Other

HRT Synchronizer

`[HRTSynchronised.h]`
This class depends only on *level0/HRT.h* and `level0/HRT.cpp` nothing else is required to compile it, the class can be easily moved to *level0*.

The first attribute, `oldTime` is a reference time of the previous call to the `Syncronise` method. The attribute `oldHrt` is the HRT value at the time of the previous call to the `Synchronise` method.
`hrtPeriod` is the current estimate of the HRT clock period, `timeOffset` is the offset used to calculate the time estimation and `lastTimeOffset` is another offset used to calculate the time estimation.

The method `Synchronise` must be called periodically to synchronise the current thread, this routine needs to be called at least 3 times before the class can be used (this is because it must aligns its offsets). The method `Period` estimates the HRT period, the estimation is done using the external source. The method `CurrentTime` returns the estimated time in seconds reading from the HRT timer.

```
private:
   double oldTime;
   int64 oldHrt;
   double hrtPeriod;
   double timeOffset;
   double lastTimeOffset;

public:
   HRTSynchronised();

   void Synchronise(int64 usecTime);
   double Period();
   double CurrentTime();
```

6.7.1 Design Note

The class `HRTSynchronised` depends only on *level0/HRT.h* and `level0/HRT.cpp` nothing else is required to compile it, the class can be easily moved to *level0*.

Chapter 7

BaseLib Level 6

INTRO TODO

- Memory Mapped CDB
- Mathematic support library
- HTTP Browsing
- System Support

7.1 Memory Mapped CDB

TODO INTRO
TODO logical schema

- MMCDBItem, MMCDBISearchFilter, MMCDBIPositionIterator
- MMCDB

MMCDBItem

`[MMCDBItem.h, MMCDBItem.cpp]`
An `MMCDBItem` is used by an MMCDB object as elements of a path. It is a contianer of elements of the same type. It inherits from `LinkedListable` and `Object`.

Attributes are not setted by the constructor by the `Init` method, only the attribute `elements` is setted up by the `Populate` method. The attribute `address` is the address of the variable, `className` stores the class name of the variable and the `variableName` the name of the variable itself; `arrayDimensions` is a NULL terminated list of dimensions, `classStructure` is a class structure associated to the class. The last attribute is a `LinkedListHolder` that is the one loaded by the `Populate` method.

```
private:
   void* address;
   FString className;
   FString variableName;
   int* arrayDimensions;
   const ClassStructure* classStructure;

private:
   LinkedListHolder elements;
```

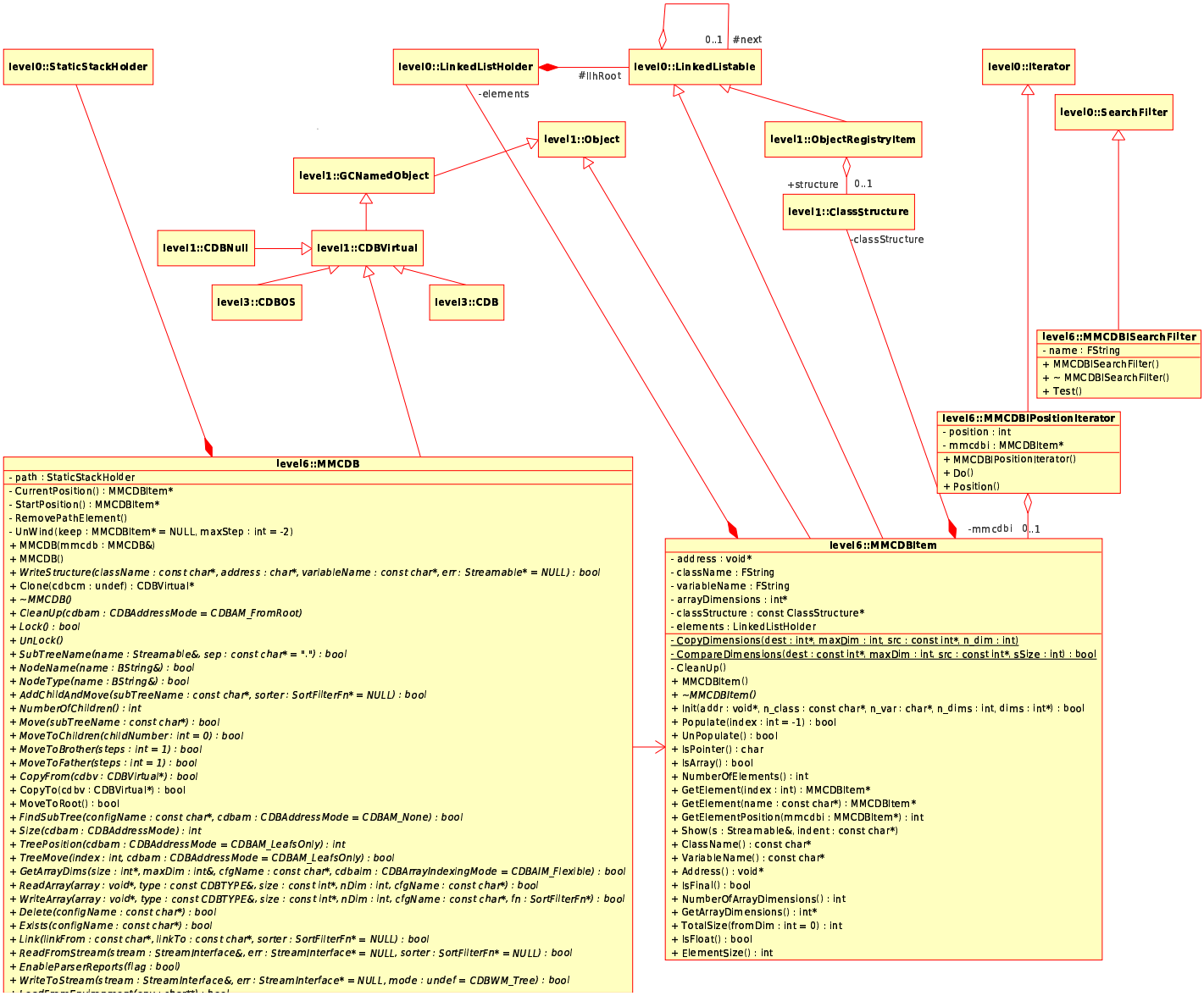

Figure 7.1: BaseLib Level 6 Memory Mapped CDB

The list of methods starts with two static methods; `CopyDimensions` copies all the dimensions that fit and collapses the last, `CompareDimensions` simply compare the `arrayDimensions`, but the arrays must be the same or comaptible. `CleanUp` deallocates memory.

The method `Init` initializes attributes, the last three parameters are only necessary to handle packed structures. The method `Populate` set up the linked list, in case of a structure creates virtaul view of the structure, in case of an arry does nothing and if (`index` its greater then zero) then it will populate one array row; `UnPopulate` basically de-populate elements.

The method `IsPointer` return `True` whether this is a pointer or not; the same can be said for `IsArray`. `IsFloat` is used for elementary types to know whether this is a `float` or `double`.

```
static void CopyDimensions(int* destination,int maxDim,const int* source,
   int numberOfDimensions);
static bool CompareDimensions(const int* destination,int maxDim,const int* source,
```

```
      int sSize);

   void CleanUp();
public:
   MMCDBItem();
   virtual ~MMCDBItem();

   bool Init(void* address,const char* className,const char* variableName,
      int numberOfDimensions=0,int* dimensions=NULL);

   bool Populate(int index=-1);
   bool UnPopulate();

   bool IsPointer();
   bool IsArray();
   bool IsFloat();

   int NumberOfElements();

   MMCDBItem* GetElement(int index);
   MMCDBItem* GetElement(const char* name);
   int GetElementPosition(MMCDBItem* mmcdbi);

   void Show(Streamable& s,const char* indent);
   const char* ClassName() const;
   const char* VariableName() const;
   void* Address() const;

   bool IsFinal();
   int NumberOfArrayDimensions();
   int* GetArrayDimensions()

   int TotalSize(int fromDim=0);
   int ElementSize();
```

MMCDBISearchFilter

[MMCDBItem.h]
The class `MMCDBISearchFilter` serches members for a given name, it inherits as usual from `SearchFilter`. The single attribute `name` accounts for the name to search for. The method `Test` is that one that performs the search on a set of data.

```
   FString name;
public:
   MMCDBISearchFilter(const char* name);
   virtual ~MMCDBISearchFilter();

   virtual bool Test (LinkedListable* data);
```

MMCDBIPositionIterator

[MMCDBItem.h, MMCDBItem.cpp]
The class `MMCDBIPositionIterator` finds the node position, it inherits from `Iterator`; the attribute `mmcdbi` is the object to search for, it will be NULL after found.

```
   int position;
   MMCDBItem* mmcdbi;

public:
   MMCDBIPositionIterator(MMCDBItem* mmcdbi);
```

```
   virtual void Do (LinkedListable* data);
   int Position();
```

MMCDB

[MMCDB.h, MMCDB.cpp
The class MMCDB is a tool to access a block of memory as a CDB, the block of memory is described by a class or structure name, the name must be registered. An MMCDB is another kind of CDB that inherits from CDBVirtual. It defines only one attribute, path, of type StaticStackHolder (a dynamic vector that implements the standard stack push and pop operations).

The method CurrentPosition gets an MMCDBItem object pointer of the last path position; StartPosition gets the root path position; RemovePathElement removes the last path position, deallocation is performed only if this is the last left. The method UnWind removes all the path elements until the one specified in maxStep, if maxStep argument is below or equal -2 means to remove all elements if equal to -1 means leave father.

The method WriteStructure either copies or references a memory structure at address address and of type className, note that a CDB transforms the memory but a MMCDB just references it. The method Clone creates a new reference to a database, or if that is not possible it creates a copy of it. CleanUp does nothing; Lock and UnLock doesn't block and simply return.

```
private:
   StaticStackHolder path;

   MMCDBItem* CurrentPosition();
   MMCDBItem* StartPosition();
   void RemovePathElement();
   void UnWind(MMCDBItem* keep=NULL,int maxStep=-2);

public:
   MMCDB(MMCDB &mmcdb);
   MMCDB();
   virtual ~MMCDB();

   virtual bool WriteStructure(const char* className,char* address,const char* variableName,
      Streamable* err=NULL);

   CDBVirtual* Clone(CDBCreationMode cdbcm);
   virtual void CleanUp(CDBAddressMode cdbam = CDBAM_FromRoot);

   virtual bool Lock();
   virtual void UnLock();
```

The method SubTreeName finds the overall path leading to the current node; the method NodeName returns in the BString argument the name of the current node, NodeType gets the type of the current node. The method NumberOfChildren returns how many branches from this node, negative numbers implies that the location is a leaf.

The method AddChildAndMove work as the following Move method; the method Move move to the specified location, the movement is relative to the current location. MoveToChildren move to a childrens, 0 is the first of the childrens. MoveToBrother is similar and 0 means remain where you are, greater then zero brothers on the right and below zero berothers on the left. MoveToFather lets you navigate to father and also to the root node; MoveToRoot moves back to the root calling MoveToFather with argument -1.

The method `CopyFrom` copies the pointed subtree in the `cdbv` into the current subtree. `CopyTo` copies the current subtree in the pointed `cdbv`.

The method `FindSubTree` searches on the right of the tree for the subtree identified by the string name, on success `nodes` attribute points to the node containing the subtree or leaf, it will not follow links. The method `Size` returns the total number of nodes; `TreePosition` returns from left to right bottom to top order the absolute location of a node; `TreeMove` moves to a location within the whole (sub)tree; the nodes are numbered from left to right and from subnode to supernode; if the node does not exist returns `False` and remains in the start position.

The method `GetArrayDims` returns how many dimensions has the `MMCDBItem` returned by the `CurrentPosition` method. `ReadArray` and `WriteArray` read and write an array data structure in the `MMCDBItem` in the current position.

The method `Delete` will remove an entry or subtree (position is relative); to delete a link use the `linkTo` as the leaf name, to delete a subtree simply specify the group node; in this implementation `Delete` returns `False`. `Exists` return `True` if a certain entry exists. `Link` simply returns `False` and does nothing.

```
virtual bool SubTreeName(Streamable& name,const char* sep = ".");
virtual bool NodeName(BString& name);
virtual bool NodeType(BString& name);
virtual int NumberOfChildren();

virtual bool AddChildAndMove(const char* subTreeName,SortFilterFn* sorter=NULL);
virtual bool Move(const char* subTreeName);
virtual bool MoveToChildren(int childNumber=0);
virtual bool MoveToBrother(int steps=1);
virtual bool MoveToFather(int steps=1);
inline bool MoveToRoot();

virtual bool CopyFrom(CDBVirtual* cdbv);
inline bool CopyTo(CDBVirtual* cdbv);

virtual bool FindSubTree(const char* configName,CDBAddressMode cdbam=CDBAM_None);
virtual int Size(CDBAddressMode  cdbam);
virtual int TreePosition(CDBAddressMode cdbam=CDBAM_LeafsOnly);
virtual bool TreeMove(int index,CDBAddressMode cdbam=CDBAM_LeafsOnly);

virtual bool GetArrayDims(int* size,int& maxDim,const char* configName,
   CDBArrayIndexingMode cdbaim=CDBAIM_Flexible);
virtual bool ReadArray(void* array,const CDBTYPE& valueType,const int* size,
   int nDim,const char* configName);
virtual bool WriteArray(const void* array,const CDBTYPE& valueType,const int* size,
   int nDim,const char* configName,SortFilterFn* sorter=NULL);

virtual bool Delete(const char* configName);
virtual bool Exists(const char* configName);
virtual bool Link(const char* linkFrom,const char* linkTo,SortFilterFn* sorter=NULL);
```

Finally follow a set of complex load/save methods that are not implemented in this class and if they are functions return `False`.

Basically `ReadFromStream` should read a database from a stream; `EnableParserReports` enables reports of parser during `ReadFromStream` into `err` argument; `WriteToStream` writes the database to stream without any ordering; `LoadFromEnvironment` loads from environment or from any NULL terminated list of chars. `ReadStructure` will read a data structure from CDB to memory and `WriteStructure` will write a data structure from memory to CDB.

```
virtual bool ReadFromStream(StreamInterface& stream,StreamInterface* err=NULL,
```

```
   SortFilterFn* sorter=NULL);
virtual void EnableParserReports(bool flag);
virtual bool WriteToStream(StreamInterface& stream,StreamInterface* err=NULL,
   CDBWriteMode mode=CDBWM_Tree);

virtual bool LoadFromEnvironment(char** env);

virtual bool ReadStructure(const char* className,char* address,
   Streamable* err=NULL);
virtual bool WriteStructure(const char* className,char* address,
   Streamable* err=NULL);
```

7.2 Mathematic Support Library

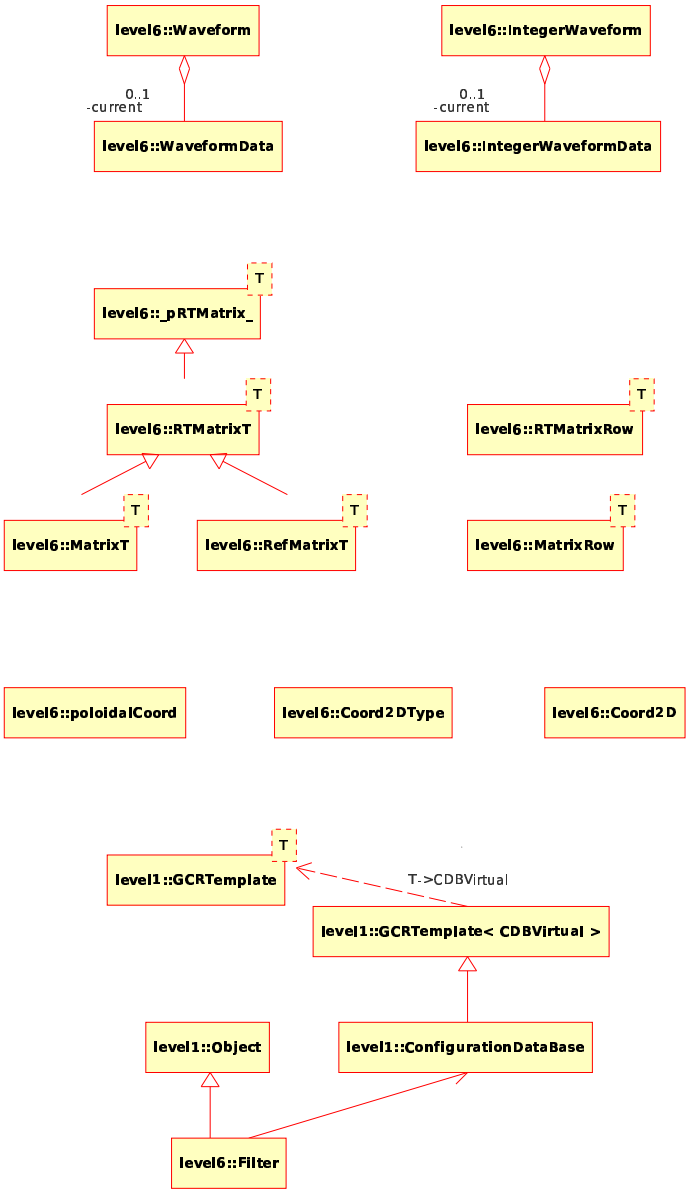

Figure 7.2: BaseLib Level 6 Mathematic Support Library

7.2.1 Matrix

This section deal about matrix and matrix's operations. There are two main implementations: one with stringent bounds checks and another without boundary checks. The first is implemented with classes: `MatrixRow` and `MatrixT` and the other one with `RTMatrixRow` and `RTMatrixT`. The whole diagram is depicted in Figure 7.3.

Classes with the name starting with `RT` are stripped from many methods and checks in boundary functions and so are more suitable to use it in real time (test it before using in Real Time).

Classes `MatrixRow` and `RTMatrixRow` implement a row matrix, `MatrixT` and `RTMatrixT` implement bidimensional matrix, no multidimensional matrix are currently implemented. `RefMatrixT` lets the user using a common C matrix.

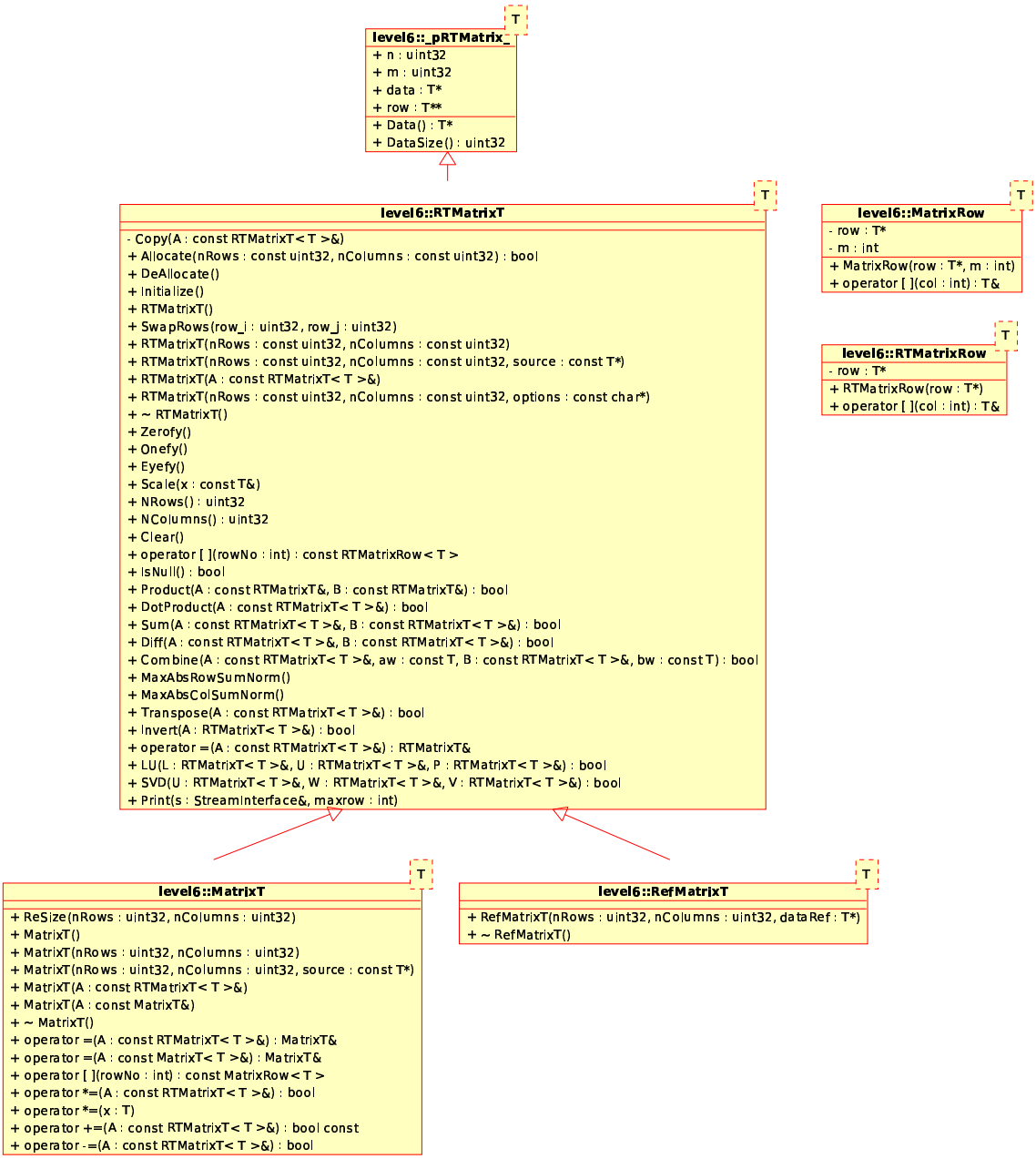

Figure 7.3: BaseLib Level 6 Mathematic Support Library

RTMatrixRow

[RTMatrix.h]
The class `RTMatrixRow` is a class that holds a row of a generic matrix (in the source code its written that a `RTMatrixRow` is a orw of a `RTMatrix` but those classes have no links). As all classes in this section such class is templatized on the `T` parameter. The main attribute is a `T*` that holds a row of data, no other attributes are present.

To access the data in a simpler way the `&operator[]` is redefined.

```
template <class T>
class RTMatrixRow{
   T* row;
public:
   inline RTMatrixRow(T *row);

   inline T &operator[](int col)const;
};
```

MatrixRow

[Matrix.h]
The class `MatrixRow` is anther implementation of a row of a matrix, this implementation is more rich. This class reimplement a row from scratch without inheriting from `RTMatrixRow`. It has not only the `T*` attribute but also an attribute `m` that count the number of elements in the row.

```
template <class T>
class MatrixRow{
   T* row;
   int m;
public:
   inline MatrixRow(T* row,int m);

   inline T &operator[](int col)const;
};
```

pRTMatrix

[RTMatrix.h]
The class `_pRTMatrix_` define the basic structure of a bidimensional matrix. It holds the main attributes to define a 2 dimensional matrix structure: the number of rows (`n` attribute) and the number of columns (`m` attribute); names of those attributes are not really explicit.

The attribute `data` is a pointer to the datas and `row` to the row information. The `data` attribute is returned by the `Data` method. The method `DataSize` returns the number of elements in the matrix (row*columns).

```
template <class T>
class _pRTMatrix_ {
public:
   uint32 n;
   uint32 m;

   T *data;
   T **row;

   inline T *Data()const;
   inline uint32 DataSize()const;
};
```

RTMatrixT

[RTMatrix.h, RTMatrix.cpp]

The template class RTMatrixT is build upond the template class _pRTMatrix_ and adds some important operations to matrix (bidimensional).

The method Copy copies or initialize an RTMatrix, it checks if the data pointer is NULL and in that case do the allocation. The method Initialize initializes all the attributes to zero.

The first contructor simply calls Initialize, the second contructor calls Allocate; the third does the same but also load the datas. The fourth constructor calls Copy, the fifth constructor will be written for allocating a zeros, ones, eye matrix. The deconstructor simply call Deallocate.

```
template <class T>
class RTMatrixT:public _pRTMatrix_<T> {
private:
   void Copy(const RTMatrixT<T> &A);

public:
   bool Allocate(const uint32 nRows,const uint32 nColumns);
   void DeAllocate();

   void Initialize();

   inline void SwapRows(uint32 row_i, uint32 row_j);

   RTMatrixT();
   inline RTMatrixT(const uint32 nRows,const uint32 nColumns);
   inline RTMatrixT(const uint32 nRows,const uint32 nColumns,const T *source);
   inline RTMatrixT(const RTMatrixT<T> &A);
   inline RTMatrixT(const uint32 nRows,const uint32 nColumns,const char* options);
   inline ~RTMatrixT();
```

The method Zerofy sets elements of the matrix to zero; Onefy sets elements of the matrix to one; Eyefy sets the diagonal entries of the matrix to one, the others to zero. The method Scale scales all the values by x.

The method NRows returns the number of rows of the matrix, NColumns returns the number of columns, Clear sets elements of the matrix to zero. The method IsNull returns True if the matrix is the null one.

```
   inline void Zerofy();
   inline void Onefy();
   inline void Eyefy();
   inline void Scale(const T &x);

   inline uint32 NRows()const;
   inline uint32 NColumns()const;
   inline void Clear();

   inline const RTMatrixRow<T> operator[](int rowNo);

   inline bool IsNull();
```

Then follows a set of mathematical functions implemented in the class. The first method Product loads the product of matrix A and B. The method DotProduct is a product element by element. Sum loads the sum of matrix A and B. Diff loads the difference of matrix A and B.

The method Combine loads the linear combination of matrix A and B. Methods MaxAbsRowSumNorm and MaxAbsColSumNorm return the absolute maximum value of the rows or of the columns.

The method Transpose loads the transposed of A, Invert inverts the matrix A passed by reference. The assignement operator is then redefined to copy a matrix to another.

At the end follow the implementation of some really useful matrix operation the LUP decomposition (also called LU decomposition) and the SVD decomposition. The method `Print` just dumps on stream the matrix.

```
   inline bool Product(const RTMatrixT& A,const RTMatrixT& B);
   inline bool DotProduct(const RTMatrixT<T>& A)

   inline bool Sum(const RTMatrixT<T>& A,const RTMatrixT<T>& B);
   inline bool Diff(const RTMatrixT<T>& A,const RTMatrixT<T>& B);

   inline bool Combine(const RTMatrixT<T>& A,const T aw,const RTMatrixT<T>& B,const T bw);

   inline T MaxAbsRowSumNorm();
   inline T MaxAbsColSumNorm();

   inline bool Transpose(const RTMatrixT<T>& A);
   inline bool Invert(RTMatrixT<T>& A);

   inline RTMatrixT &operator=(const RTMatrixT<T>& A);

   bool LU(RTMatrixT<T>& L,RTMatrixT<T>& U,RTMatrixT<T>& P);
   bool SVD(RTMatrixT<T>& U,RTMatrixT<T>& W,RTMatrixT<T>& V);

   void Print(StreamInterface& s,int maxrow=10);
};
```

MatrixT

[Matrix.h, Matrix.cpp]
The class `MatrixT` is an extension of the class `RTMatrixT` with algebraic capabilities. It redefines the assignment operator, the access operator, the product, the sum and the difference operators.

```
template <class T>
class MatrixT: public RTMatrixT<T>{
public:
   void ReSize(uint32 nRows,uint32 nColumns);

   inline MatrixT();
   inline MatrixT(uint32 nRows,uint32 nColumns);
   inline MatrixT(uint32 nRows,uint32 nColumns,const T* source);
   inline MatrixT(const RTMatrixT<T>& A);
   inline MatrixT(const MatrixT& A);
   inline ~MatrixT();

   inline MatrixT &operator=(const RTMatrixT<T>& A);
   inline MatrixT &operator=(const MatrixT<T>& A);

   inline const MatrixRow<T> operator[](int rowNo);

   inline bool operator*=(const RTMatrixT<T>& A);
   inline void operator*=(T x);
   inline bool const operator+=(const RTMatrixT<T>& A);
   inline bool operator-=(const RTMatrixT<T>& A);
};
```

RefMatrixT

[RefMatrix.h]
The class template `RefMatrixT` create a matrix built on top of a C matrix, i.e. once created a C matrix you can manipulate it using a `RefMatrixT` and so all methods in `RTMatrixT`. Creating a new

`RefMatrixT` from a C matrix you are being creating all the row pointers that speed up operations in the next phase of compuation.

```
template <class T>
class RefMatrixT: public RTMatrixT<T>{
public:
   RefMatrixT(uint32 nRows,uint32 nColumns,T* dataRef);
   ~RefMatrixT()
};
```

7.2.2 Waveforms

This section group togheter basically two class that let the user defining manually a waveform by defining touples of amplitude and time.

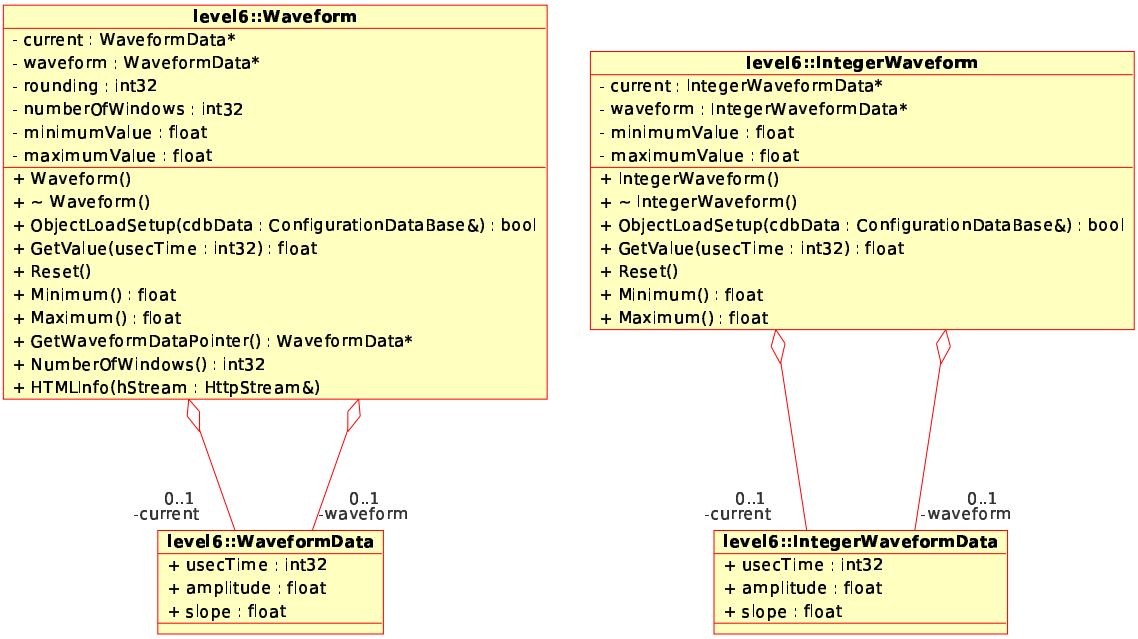

Figure 7.4: BaseLib Level 6 Mathematic Support Library

Classes involved in this section are depicted in Figure 7.4 and listed below:

- WaveformData, Waveform
- IntegerWaveformData, IntegerWaveform

WaveformData, Waveform

`[Waveform.h]`
The class `Waveform` lets the user define a specified waveform of data. The basic data structure of a waveform is the `WaveformData` structure that holds each amplitude and time of the defined signal.

```
struct WaveformData {
   int32 usecTime;
   float amplitude;
   float slope;
};
```

The class `Waveform` keeps its internal time in microseconds but loads data from CDB in seconds. Note that all `slopes` are calculated at the moment of creation (during methods `ObjectLoadSetup`),

and that the waveform has to be resetted (function `Reset`) between a pulse and the other.

The first attribute, `current` is a pointer to the current element in the `Waveform`, `waveform` is a pointer to the first element in the `Waveform` array.

The attribute `rounding` is a rounding factor for conversion between seconds and microseconds, `numberOfWindows` is the number of time windows. The attribute `minimumValue` and `maximumValue` are respectively the minimum and the maximium value of the waveform.

```
class Waveform {
private:
   WaveformData* current;
   WaveformData* waveform;
   int32 rounding;
   int32 numberOfWindows;

   float minimumValue;
   float maximumValue;
```

Using method `ObjectLoadSetup` its possible to load a `Waveform` from a CDB; in this case you have to define in the configuration file two arrays: `Times` and `Amplitudes`. The method `GetValue` returns the value of the waveform at a certain time. The method `Reset` resets the internal states and waveforms; to be called in the "PREPULSE" phase. `Minimum` returns waveform's minimum value; `Maximum` returns waveform's maximum value. `GetWaveformDataPointer` returns `waveform` attribute value. `NumberOfWindows` gets number of windows in the current waveform. `HTMLInfo` print the information about the waveform with HTML format.

```
public:
   Waveform();
   virtual ~Waveform();

   bool ObjectLoadSetup(ConfigurationDataBase& cdbData);

   inline float GetValue(int32 usecTime);

   inline void Reset();

   float Minimum();
   float Maximum();
   inline WaveformData* GetWaveformDataPointer();
   inline int32 NumberOfWindows();

   void HTMLInfo(HttpStream& hStream);
};
```

IntegerWaveformData, IntegerWaveform

`[IntegerWaveform.h]`
The class `IntegerWaveform` works thanks to an `IntegerWaveformData` structure. Such structure holds data for a single element in an `IntegerWaveform`. The class is the same as the class `WaveformData`.

```
struct IntegerWaveformData {
   int32 usecTime;
   float amplitude;
   float slope;
};
```

An `IntegerWaveform` is similar to a `Waveform` and keeps its internal time in microseconds; in the same way all slopes are calculated at the moment of creation (during function `ObjectLoadSetup`). Attributes have also the same meaning.

```
class IntegerWaveform {
```

```
private:
   IntegerWaveformData* current;
   IntegerWaveformData* waveform;

   float minimumValue;
   float maximumValue;

public:
   IntegerWaveform();
   virtual ~IntegerWaveform();

   bool ObjectLoadSetup(ConfigurationDataBase& cdbData);
   inline float GetValue(int32 usecTime);

   inline void Reset();

   float Minimum();
   float Maximum();
};
```

7.2.3 Filters

TODO

Filter

[Filter.h, Filter.cpp]
The class `Filter` implements a single input single output filter. Initialisation is performed via CDB, i.e. it must be written in the configuration file. The class descend only from `Object`.

A filter in the configuration file can be defined in different ways as the table below explains:

Numerator = {a0 a1 a2...} Denominator= {b0 b1 .... }	
0 = {a0 a1 a2...} 1 = {-b1 -b2.... }	b0 assumed to be 1
Poles = {p0 p1 ... } Zeros = {z0 z1 ... } SamplingTime = 1e-3 Gain = g	$\frac{p0}{S+p0} * \frac{p1}{S+p1}$ $\frac{S+z0}{z0} * \frac{S+z1}{z1}$ which multiplies the whole expression

The first two filter configuration are already expressed in z domain, the last one is expressed in s domain and the Filter class discretizes the transfer function using Tustin bilinear transformation.

```
private:
   int inputSize;
   int outputSize;

   float* inputStatus;
   float* outputStatus;
   double* inputCoefficients;
   double* outputCoefficients;

private:
   bool InitP(ConfigurationDataBase &cdb);
   bool Resize(int inputSize,int outputSize);

public:
   ~Filter();
```

```
Filter();

bool Init(ConfigurationDataBase &cdb);
inline void Reset(float input=0.0,float output=0.0);

inline float Process(float input);
```

7.2.4 Coords

TODO

7.3 HTTP Browsing

TODO

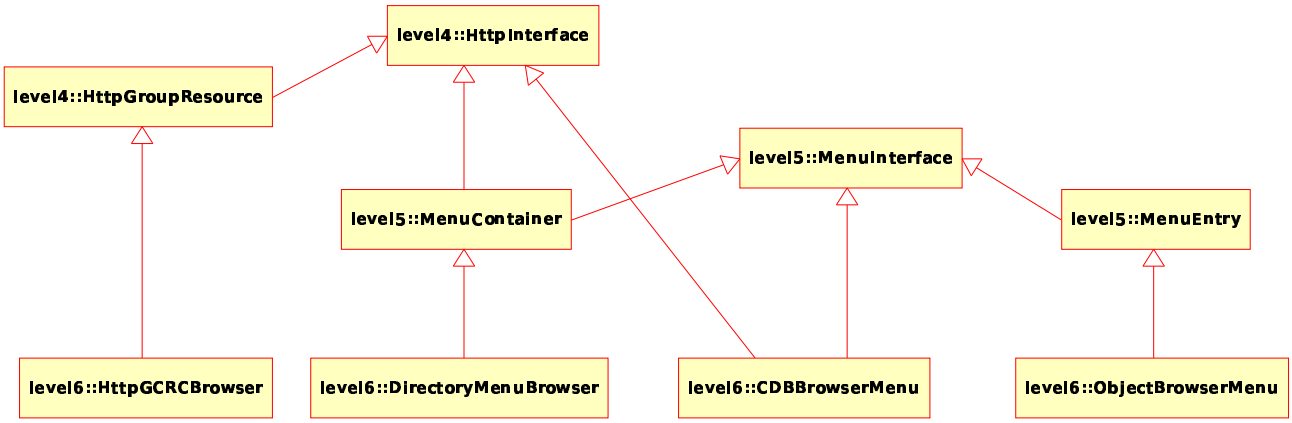

Figure 7.5: BaseLib Level 6 HTTP Browsing

7.4 System Support

TODO

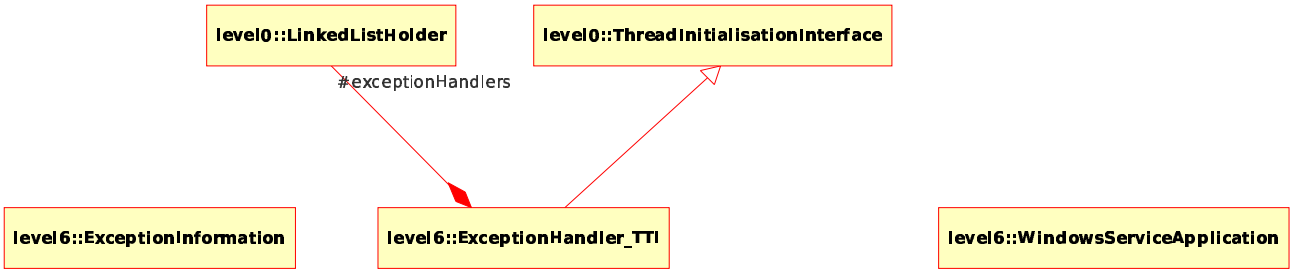

Figure 7.6: BaseLib Level 6 System Support Library

7.4.1 Design Notes

System support must be moved to level0.

Part III

MARTe

Chapter 8

MARTe Support Library

This chapter is about the core of the MARTe framework, i.e. the source code that let's GAMs work one with the other, reading data and write data from and to acquisition boards and take the control system in synch with all the plant system. This chapter address the dispatching of the different GAM's codes, the dispatching is done by the master thread that is bounded to the `RealTimeThread` object. No scheduling issues are addressed because there is no scheduling entity, the schedule of GAMs execution is done serially using a dependency analisys done a priori.

In this chapter we strongly analise sources in the *MarteSupportLib* directory of the MARTe distribution package and then the MARTe executables coming with MARTe: `MARTe` and `MARTeService` (both in the root directory of the source package).

The chapter starts introducing a new type of component in the framework: the device driver (high level and low level device drivers). A real time framework to be deterministic must control each aspect of the control system, let's imagine that we have a real time algorithm that let's the feedback control waiting for input data from an acquisition board, if the device driver respond in microseconds but sometimes in milliseconds the responsiveness could not be sufficient for application we want, some delays can be caused from the Operating System so the developers of BaseLib/MARTe try to optimize and sometimes rewrite the device driver we use.

8.1 IO Drivers

This section addresses how device drivers are to be written to comply with the MARTe interface. First of all a device driver in MARTe is called a **Generic Acquisition Module** (*GenericAcqModule* or *GACQM*). A GACQM can be of different types regarding also the operating system on which is designed to work with. A common properties of all **Generic Acquisition Module**s is that once they start on a system they are never removed until MARTe has finished its execution and are permenently running (they have no "run" or "start" method); this characteristic make GACQM differ from GAM that are used within a precise slice of time calling the implemented control code once in a cycle.

Figure 8.1 is the UML diagram of the involved classes in this section, classes are just two: `TriggerTimeService` (quite usefull) and `GenericAcqModule` the most important class, every device driver in MARTe is a class that implements the `GenericAcqModule` interface.

GenericAcqModule, TriggerTimeService

`[GenericAcqModule.h, GenericAcqModule.cpp]`
The class `GenericAcqModule` is an abstract class (an interface with some methods implemented and

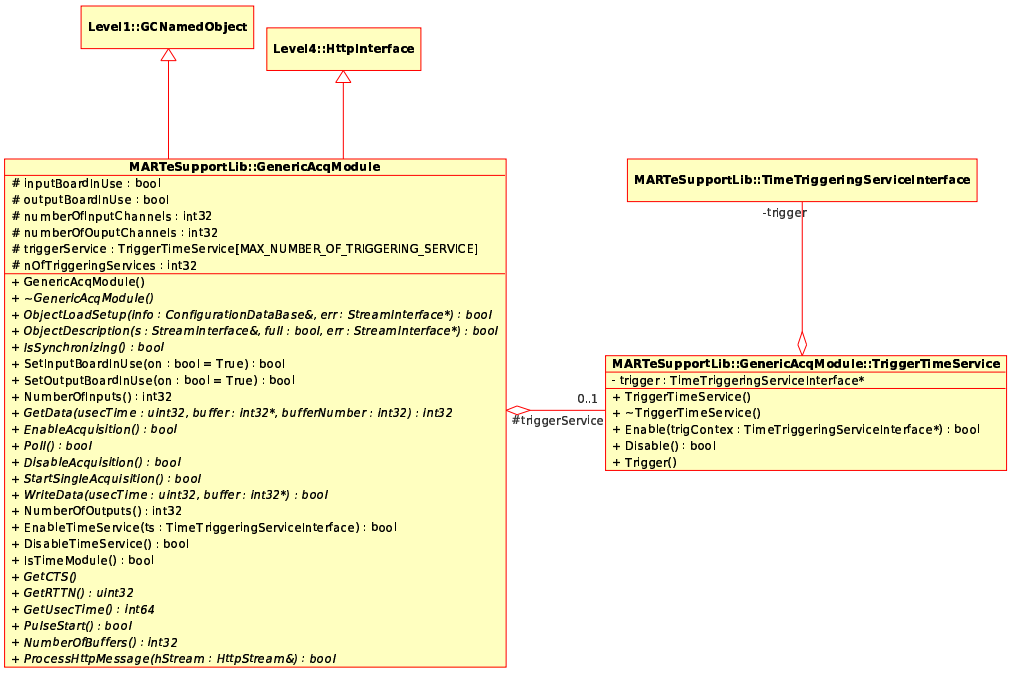

Figure 8.1: MARTe Generic Acquisitio Module

with attributes) that abstracts a generic driver for an acquisition board. An acquisition board can generate data (synchronously or asynchcronously) and output data to the environment. But those are not the only tasks an acquisition board can do, a timing module, for example, can simply wake up an interrupt every predefined period or time instant. A `GenericAcqModule` abstracts all kind of external device that collect data, output data and generate events, i.e. also timing modules are `GenericAcqModule`. The end of an AD or DA conversion is also considered as an event.

A `GenericAcqModule` inherits from `GCNamedObject`, i.e. is garbage collectable and from `HttpInterface` (it can be ispected using an internet browser). The class relies on an internal defined class called `TriggerTimeService`. The class's interface follows. Such class is not necessary because it holds just one attribute `trigger` that points to a `TimeTriggeringServiceInterface` object. The class is only a sandbox that lets the user call in a safer way the `trigger` method of the holded class (via the `Trigger` method).

```
class TriggerTimeService{
private:
   TimeTriggeringServiceInterface* trigger;

public:
   TriggerTimeService();
   ~TriggerTimeService();

   bool Enable(TimeTriggeringServiceInterface* trigContext);
   bool Disable();
   void Trigger();
}
```

We switch now to analyse attributes of the `GenericaAcqModule` class. The first attribute (`inputBoardInUse`) is a boolean, `True` if one *GAM* is using this `GenericAcqModule` (*GACQM*) as input, such decision restrict the usage of a GACQM from only one GAM at a time; `outputBoardInUse` is `True` if one GAM is using this module as output.

The attribute numberOfInputChannels is the umber of input channels in the module it's equal to the size of the data buffer (in int32); numberOfOutputChannels holds the number of output channels in the module.

Last group of attributes store informations regarding timing devices (or event generating boards); the array attribute triggerService lets the user register up to MAX_NUMBER_OF_TRIGGERING_SERVICE classes that will be informed (calling Trigger method) of the happening of an event; nOfTriggeringServices simply counts the number of registered TriggerTimeService objects.

```
protected:
   bool inputBoardInUse;
   bool outputBoardInUse;

   int32 numberOfInputChannels;
   int32 numberOfOutputChannels;

   TriggerTimeService triggerService[MAX_NUMBER_OF_TRIGGERING_SERVICE];
   int32 nOfTriggeringServices;
```

First two methods in the class are the constructor and the destructor. Then comes ObjectLoadSetup, ObjectSaveSetup and ObjectDescription that loads the setup from a CDB, save it on a CDB and output a description of the object on a stream.

The method IsSynchronizing returns if the module is synchronizing, i.e. if it produces periodical events. the attribute SetInputBoardInUse set the input board in use, each module should use these method appropriately, for instance an ATM Module can only be used either as input module or output module; in this case both flags must be set by the SetInputBoardInUse/SetOutputBordInUse (the last follows) methods, for modules that can perform both input and output activities each flag should be set accordingly.

Input Boards The method NumberOfInputs returns the number of inputs to be read, GetData copies local input buffer in destination buffer, returns 0 if data is not ready, below than 0 if error, greater than 0 if it all done. The method EnableAcquisition enable the acquisition and is disabled by DisableAcquisition (is not really clear if it also disables the board or not). The method StartSingleAcqusition command the start of a single acquisition activity.

Output Boards The method NumberOfOutputs returns the number of outputs to be read, WriteData copies local data buffer in the board's buffer.

Timing Boards The method EnableTimeService enables and register a TimeTriggeringServiceInterface object (passed by argument to the method). DisableTimeService disables and deletes all registered TimeTriggeringServiceInterface objects. Poll is to be used by timing modules which support polling, no more explanations at this level are needed. IsTimeModule return True if the board is a Timing Device and it has the triggering service enabled. GetCTS, GetRTTN and GetUsecTime are methods that returns the time in different forms.

For simulation purposes the method PulseStart was added. The method NumberOfBuffers returns the number of buffers used for acquisition; it is only used by boards with circular buffers and multiple acquisitions methods. ProcessHttpMessage is required by the HttpInterface and outputs an html page with information about the acquisition driver.

```
public:
   GenericAcqModule();
   virtual ~GenericAcqModule();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectDescription(StreamInterface& s,bool full=False,
      StreamInterface* err=NULL)=0;

   virtual bool IsSynchronizing();
   virtual bool SetInputBoardInUse(bool on=True)=0;
   virtual bool SetOutputBoardInUse(bool on=True)=0;

   int32 NumberOfInputs() const;
   virtual int32 GetData(uint32 usecTime,int32* buffer,int32 bufferNumber=0)=0;

   virtual bool EnableAcquisition();
   virtual bool Poll();
   virtual bool DisableAcquisition();
   virtual bool StartSingleAcquisition();

   int32 NumberOfOutputs() const;
   virtual bool WriteData(uint32 usecTime,const int32* buffer)=0;

   bool EnableTimeService(TimeTriggeringServiceInterface* ts);
   bool DisableTimeService();
   bool IsTimeModule() const;

   virtual uint16 GetCTS();
   virtual uint32 GetRTTN();
   virtual int64 GetUsecTime();

   virtual bool PulseStart();

   virtual int32 NumberOfBuffers();
   virtual bool ProcessHttpMessage(HttpStream& hStream);
```

8.1.1 Design Notes

The class `TriggerTimeService` has only one attribute, by the way it can be deleted. Methods are not strictly necessary.

8.2 Time Triggering Services

This second section in this chapter will address time triggering services, such services associate temporal events with service routines; one temporal event can have many service routines associated with it. Temporal events can be the start or the end of an acquisition activity, a periodic clock, a timing sinchronization signal.. Temporal events can also generate other temporal events (think about a frequency divider).

In the past versions of MARTe there was only one type of `TimeTriggeringServiceInterface` and there is no interface defined, the class we are speaking about is the `ExternalTimeTriggeringService` class. Such functionalities are now played by the `InterruptDrivenTTS` class and the triggering activity on data polling is performed by the `DataPollingDrivenTTS` class (note that the synchronisation of the RealTimeThread is now done within the execution of the `TimeInputGAM`).

In Figure 8.2 there is the UML schema depicting classes involved in this section. Such classes are:

- TimeServiceActivity

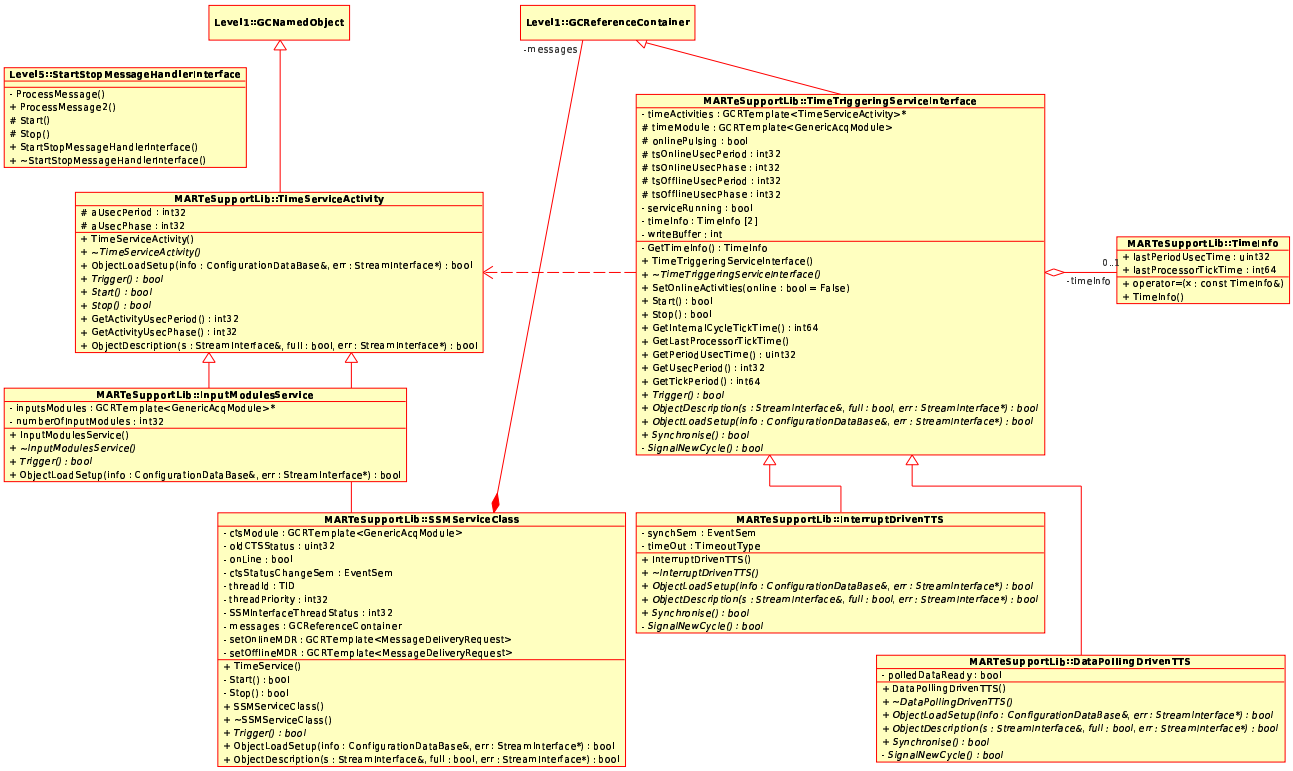

Figure 8.2: MARTe Timing infrastructure

- InputModulesService
- SSMServiceClass
- TimeTriggeringServiceInterface, TimeInfo
- InterruptDrivenTTS
- DataPollingDrivenTTS

In Figure 8.3 is shown how classes interact and are linked one each other.

A GACQM can also be used to synchronise, in this case you need a `GAM` associated with the GACQM that calls its `Synchronize` method, the `InterruptDrivenTTS` offer a synchronization via ISR and semaphores but the `PollingDrivenTTS` offer a synchronization by polling a register (calling the Poll method of a synchronizing GACQM).

TimeServiceActivity

`[TimeServiceActivity.h, TimeServiceActivity.cpp]`
The class `TimeServiceActivity` abstract a periodic activity to do every `aUsecPeriod` with an `aUsecPhase` phase respect to a defined time event. Such class inherits only from `GCNamedObject`. There are only two attributes and we have just spend some words about them, the unit of time is the micro second.

```
protected:
   int32 aUsecPeriod;
   int32 aUsecPhase;
```

The constructor intializes the attributes to invalid values (-1), the destructor simply does nothing, `Trigger` isa pure virtual methotd, `Start` and `Stop` methods are the interface methods with the `StartStopMessageHandlerInterface` class and simply return true. The method `GetActivityUsecPeriod`

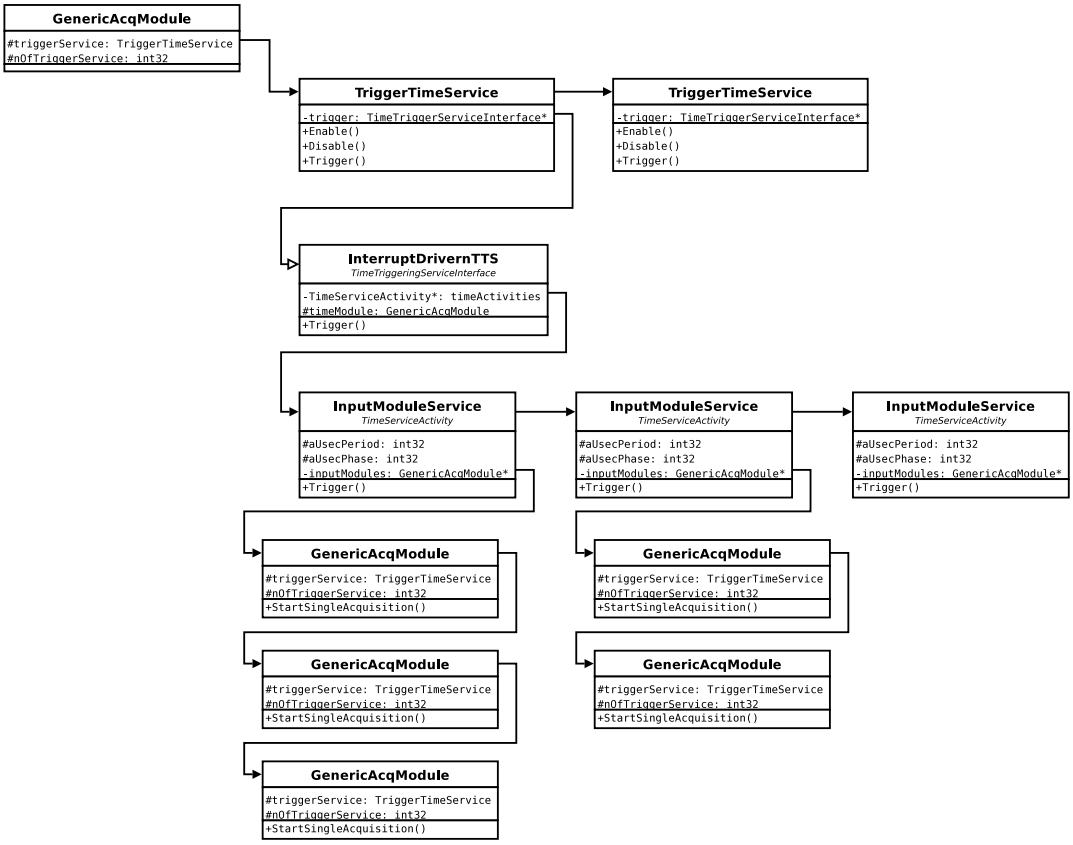

Figure 8.3: MARTe Timing Classes logical linkage

returns the attribute `aUsecPeriod` and `GetActivityUsecPhase` returns `aUsecPhase`.

The method `ObjectLoadSetup` loads time service activity parameters from a CDB. `ObjectDescription` writes on a stream all module's parameters.

```
public:
   TimeServiceActivity();
   virtual ~TimeServiceActivity();

   virtual bool Trigger()=0;

   virtual bool Start();
   virtual bool Stop();

   inline int32 GetActivityUsecPeriod();
   inline int32 GetActivityUsecPhase();

   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool ObjectDescription(StreamInterface& s,bool full=False,StreamInterface* err=NULL);
```

InputModulesService

`[InputModulesService.h, InputModulesService.cpp]`
The class `InputModulesService` is the first implementation of the class `TimeServiceActivity`. It holds a list of GACQM, on *trigger*, at the correct period and phase, all GACQM's `StartSingleAcquisition` methods will be called. The attribute `inputsModules` is a pointer to an array of `GenericAcqModule`, `numberOfInputModules` count how many elements are in the array.

The constructor simply initialize the array that is populated via the `ObjectLoadSetup` and the distructor delete all elements in the array. The Trigger method call the `StartSingleAcquisition` method of each element in the array.

```
private:
   GCRTemplate<GenericAcqModule>* inputsModules;
   int32 numberOfInputModules;

public:
   InputModulesService();
   virtual ~InputModulesService();

   virtual bool Trigger();

   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
```

SSMServiceClass

`[SSMServiceClass.h, SSMServiceClass.cpp]`
The second implementation of the class `TimeServiceActivity` is the `SSMServiceClass`, i.e. the *Supervisor State Machine Service Class*.

This is a dismissed class and can be taken only as an example, it will mimic the JET system supervisor state machine for MARTe receiving directly messages from the JET control (and timing) system; i.e. it was a replacement of part of the *CODASLib*.

The first attribute, `ctsModule`, is a reference to the device driver module providing information about the pulse state, at JET the CTS is the *Central Timing System*. The attribute `oldCTSStatus` is the previous state of the pulse; `onLine` tell us whether the pulse is in progress or not; `ctsStatusChangeSem` is a semaphore used to signal the change in the state of the pulse; `threadId` is a the identifier of the messaging handling task whos priority is setted with `threadPriority`; `SSMInterfaceThreadStatus` is the status of the task handling the messaging during pulse, related events: *ACTIVE* or *STOPPED*.

The attribute `messages` is a container of the messages for the *State Machine* and `Real Time Threads`; `setOnlineMDR` and `setOfflineMDR` are two message sender for the *ONLINE* and *OFFLINE* states.

```
private:
   GCRTemplate<GenericAcqModule> ctsModule;

   uint32 oldCTSStatus;

   bool onLine;

   EventSem ctsStatusChangeSem;
   TID threadId;
   int32 threadPriority;
   int32 SSMInterfaceThreadStatus;

   GCReferenceContainer messages;
   GCRTemplate <MessageDeliveryRequest> setOnlineMDR;
   GCRTemplate <MessageDeliveryRequest> setOfflineMDR;
```

The method `TimeService` is the messaging handling method; `Start` and `Stop` implements the `StartStopMessageHandlingInterface` starting and stopping the task handling pulse status change. `Trigger` is the method called by `TimeTriggeringServiceInterface`, note that the code in this method is executed only if $((ActualTime\%Period) == Phase)$.

```
   void TimeService();
```

```
   bool Start();
   bool Stop();

public:
   SSMServiceClass();
   virtual ~SSMServiceClass();

   virtual bool Trigger();

   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool ObjectDescription(StreamInterface& s,bool full=False,StreamInterface* err=NULL);
```

TimeTriggeringServiceInterface, TimeInfo

[TimeTriggeringServiceInterface.h, TimeTriggeringServiceInterface.cpp]
The `TimeTriggeringServiceInterface` class is a container of `TimeServiceActivity` objects. The first attribute is infact a pointer to an array of `TimeServiceActivity` object (via the classic `GCRTemplate`).

The `TimeTriggeringServiceInterface` use the following `TimeInfo` structure. The structure holds a first attribute `lastPeriodUsecTime` that registered the last time obtained from the time module board (in μsec); next attribute, `lastProcessorTickTime`, is the last sample from the cpu internal clock as ticks. Then the `operator=` is overridden and also the constructor is redefined.

```
struct TimeInfo{
   uint32 lastPeriodUsecTime;
   int64 lastProcessorTickTime;

   inline void operator=(const TimeInfo &x);
   TimeInfo();
};
```

Now we switch speaking about the `TimeTriggeringServiceInterface` class that is a `GCReferenceContainer`. The first attribute is a pointer to an array of `TimeServiceActivity` objects; `timeModule` is a reference to a *TimeModule* (a timing board GACQM).

The attribute `onlinePulsing` is a flag to monitor online pulsing activities, to be controlled by the owning object; `tsOnlineUsecPeriod` is the time service period for online operations in μsec; `tsOnlineUsecPhase` is the time service phase for online operations in μsec; `tsOfflineUsecPeriod` is the time service period for offline operations in μsec; finally `tsOfflineUsecPhase` is the time service phase for offline operations in μsec. This design probably is thought not generally but designed for the JET control system.

The boolean attribute `serviceRunning` is a flag monitoring the status of the activities (`TimeServiceActivityes`) `timeInfo` is a double buffer containing the time information, there is always one write-only `TimeInfo` structure while the other is read-only; the `Trigger` method updates the index of the write-only buffer; such index is the attribute `writeBuffer`.

```
private:
   GCRTemplate<TimeServiceActivity>* timeActivities;

protected:
   GCRTemplate<GenericAcqModule> timeModule;

   bool onlinePulsing;
   int32 tsOnlineUsecPeriod;
   int32 tsOnlineUsecPhase;
   int32 tsOfflineUsecPeriod;
   int32 tsOfflineUsecPhase;
```

```
private:
   bool serviceRunning;
   TimeInfo timeInfo[2];
   int writeBuffer;
```

The method `GetTimeInfo` returns the read-only buffer (the last written timestamp buffer). The constructor initialize to zero all the attributes and the destructor simply calls all destructors. The method `SetOnlineActivities` set the attribute `onlinePulsing` to `online`; `Start` starts the `TimeTriggeringServiceInter` and `Stop` stops it.

The method `GetInternalCycleTickTime` gets internal cycle time in ticks, asks to the `HRT` static class the current time in tick and subtract from that the `timeInfo.lastProcessorTickTime`; `GetLastProcessorTic` gets the last sample from the cpu internal clock as ticks; `GetPeriodUsecTime` gets time in μsec but as multiple of cycle time; `GetUsecPeriod` returns the period in μsec; `GetTickPeriod` returns the period in cpu ticks.

The method `Trigger` is to be called by a synchronising `GenericAcqModule` on data arrival; for data arrival that is signaled by interrupt, this method is to be called within the ISR; for data arrival that based on polling a register, this method is to be called within the `GetData` method of the synchronising `GenericAcqModule`. If $((ActualTime\%Period) == Phase)$ this method updates the `timeInfo` structure and calls the `SignalNewCycle` method, finally it checks if some activity is to be executed and if is the case calls its `Trigger` method.

The metohd `Synchronise` synchronizes the system to the cycle time, for triggering methods posted by interrupts the `Synchronise` waits on a semaphore to be posted by the ISR, namely the `SignalNewCycle` method; for triggering methods based on data arrival without interrupts, the `Synchronise` returns immediately and the synchronisation is performed by the `GetData` method of the synchronising module.

The method `SignalNewCycle` is used within the `Trigger` method. It performs a set of activities marking the start of the new real-time cycle; for interrupt driven synchronising methods, the `SignalNewCycle` method posts the semaphore the `Synchronise` method is waiting on. For triggering methods based on data arrival, the SignalNewCycle returns without performing any activity.

```
private:
   inline TimeInfo GetTimeInfo() const;

public:
   TimeTriggeringServiceInterface();
   virtual ~TimeTriggeringServiceInterface();

   void SetOnlineActivities(bool online=False);
   bool Start();
   bool Stop();

   inline int64 GetInternalCycleTickTime() const;
   inline uint64 GetLastProcessorTickTime() const;
   inline uint32 GetPeriodUsecTime() const;
   inline int32 GetUsecPeriod() const;
   inline int64 GetTickPeriod() const;

   virtual bool Trigger();

   virtual bool ObjectDescription(StreamInterface& s,bool full=False,
      StreamInterface* err=NULL);
   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);

   virtual bool Synchronise()=0;
```

```
private:
   virtual bool SignalNewCycle()=0;
```

InterruptDrivenTTS

[InterruptDrivenTTS.h, InterruptDrivenTTS.cpp]
This is the first implementation of the TimeTriggeringServiceInterface, this implementation associate ad each interrupt a triggering event (i.e. an acquisition cycle). Interrupt can be generated from any type of board, acquisition board, network cards, timing modules etc..

The attribute synchSem is an EventSem that will inform a task waiting on the Synchronise method (for example a RealTimeThred) that something is happening (the Trigger method is called); timeOut holds the semaphore timeout time.

The method Synchronise only waits on the synchSem and the method SignalNewCycle that is called by the Trigger method post the same semaphore.

```
private:
   EventSem synchSem;
   TimeoutType timeOut;

public:
   InterruptDrivenTTS();
   virtual ~InterruptDrivenTTS();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectDescription(StreamInterface& s,bool full=False,
      StreamInterface* err=NULL);

   virtual bool Synchronise();

private:
   virtual bool SignalNewCycle();
```

DataPollingDrivenTTS

[DataPollingDrivenTTS.h, DataPollingDrivenTTS.cpp]
The class DataPollingDrivenTTS implements the TimeTriggeringServiceInterface in cases where we want to poll a registry for data arrival (for performance reasons) or we do not have a source of interrupts.

The DataPollingDrivenTTS has only one boolean attribute polledDataReady that is true when data is ready. The method Synchronise synchronizes the system to the cycle time as expected, such method is implemented has a C loop testing the GenericAcqModule::Poll method return value, the synchronization is done by the GetData method in a GenericAcqModule. The method SignalNewCycle set polledDataReady to True, remember that is called by the Trigger method that is called by the GetData in the GACQM.

```
private:
   bool polledDataReady;

public:
   DataPollingDrivenTTS();
   ~DataPollingDrivenTTS()

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectDescription(StreamInterface& s,bool full=False,
      StreamInterface* err=NULL);
```

```
    virtual bool Synchronise();

private:
    virtual bool SignalNewCycle();
```

8.2.1 Design Notes

The first thing that is really noticing is that most objects in this section implements the `StartStopMessageHandlerI` but no one inherits from it; who has programmed those classes has forgot to declaring the inheritance relation probably.

Som words about the general purpose utilizability of the che class `TimeTriggeringServiceInterface` is generic enough? i.e. creating only two work cases: *online* and *offline* is enough for any existent plant system? Can exists any other states? States can be generally defined?

What happens if, using `DataPollingDrivenTTS`, no one call the `GetData` method? Probably the system will hang.. Differently if a polling module doesn't implement the `Poll` method all code continue to work without problems.

A modification that must be done is that when we call `GenericAcqModule::Trigger()` we have to pass by argument the associated event's time. In this way if, as usual in the code, one want to ask the event's time to the `TimeTriggeringServiceInterface` can do it but is not necessary so the code can be a bit more faster and simpler to read and understand. Now a `TimeTriggeringServiceInterface` must know how was calling it and asks to such GACQM the time.

8.3 Execution Support

Basically this section address all about execution of the real time control code. MARTe defines four realms or stages of code execution: the initialization, offline, online and safety. In each of this stages you can define your scheduling table, the schedulability entities are the GAMs (calling the `Execute` method). Who is in charge to dispatch the execution and choose (following a state machine) in which realm we are now executing is the `RealTimeThread`. To achieve bounded time responsiveness a single `RealTimeThread` should be executed to a single processor core without other threads, the scheduling of GAM's execution is done by a data dependency analisys offline. If a system can take advantage of many cores a control algorithm can also be parallelized (this feature is not supported right now).

In this section the `RealTimeThread` is presented and some other classes to register code performances are analised. Such performance data can be used to make schedulability analisys and also to try to understand which part of the code lack in performance. In Figure 8.4 an UML class diagram showing the class relation between classes involved in this section is depicted, a list of classes follows.

- RTCodeStatsStruct
- ExecutionModule
- RealTimeThread, Status
- MARTeMenu
- MARTeContainer

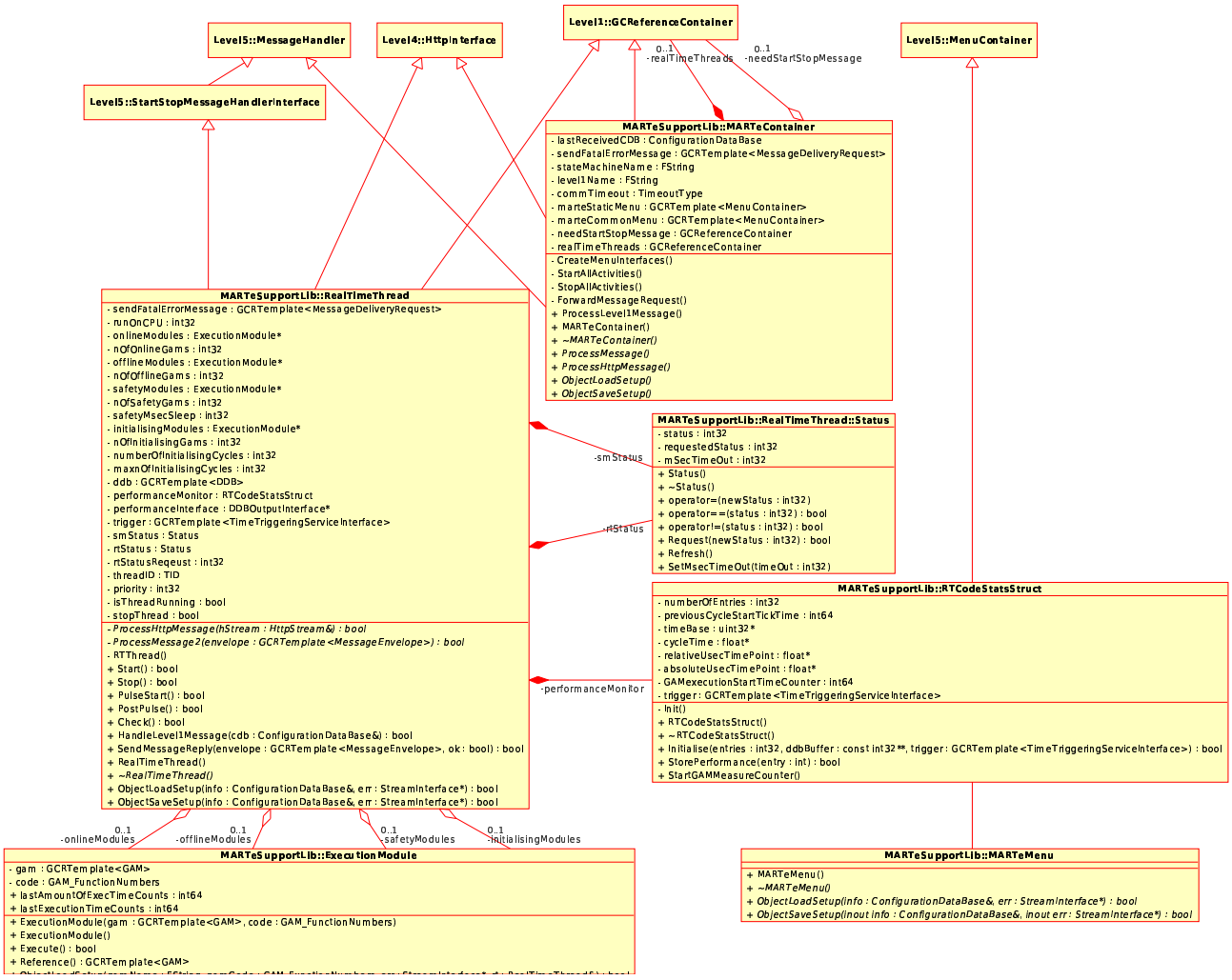

Figure 8.4: MARTe Executor infrastructure (GAM's dispatcher)

Let's spend some words to the last class `MARTeContainer`: it is MARTe. Well MARTe infact is a collection of a `ConfigurationDataBase`, a fatal error message dispatcher/receiver, an associated state machine, a level1 message receiver, a menu based interface and a set of `RealTimeThread`s. All this list is well depicted in Figure 8.5.

Note that if a `MARTeContainer` has an associated state machine such machine is outside the `MARTeContainer` but for example `RealTimeThread`s are part of it. The border between what is inside and what is not isn't well defined and require a deep understanding of the whole project.

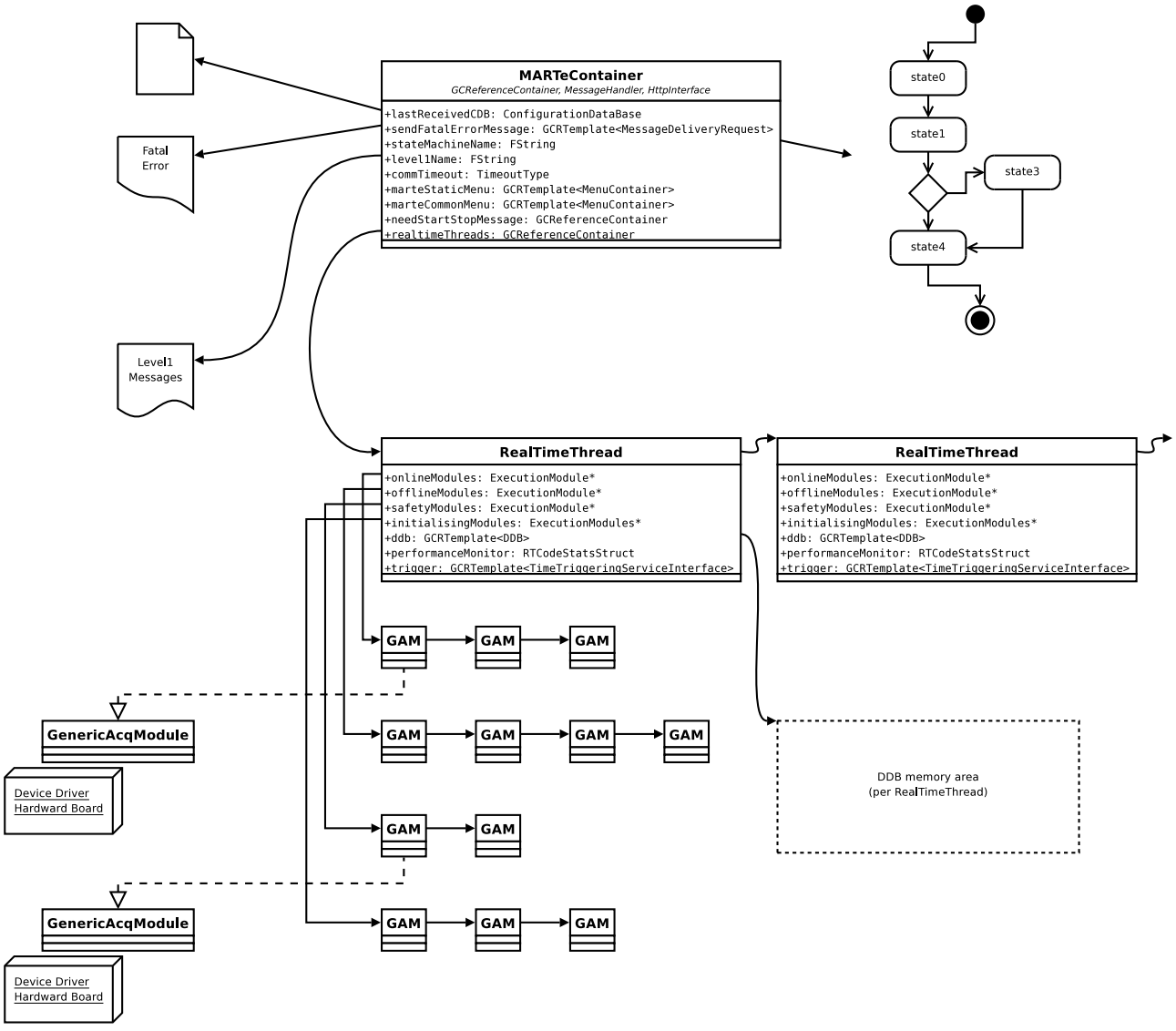

Figure 8.5: MARTe Container UML/logical scheme

8.3.1 RTCodeStatsStruct

[RTCodeStatsStruct.h]
The class RTCodeStatsStruct is the Real-Time code performance monitor, i.e. it monitors the execution time of the code. Then we will see how it is used inside the RealTimeThread entity.

The time information collected are stored in a DDB memory area, it is not compulsory to store them in a DDB memory area, the RTCodeStatsStruct class require an int32* as the address where to store the statistics. In Figure 8.6 is illustrated the format of the time data that will be stored (the number of the saved entry must be precalcolated).

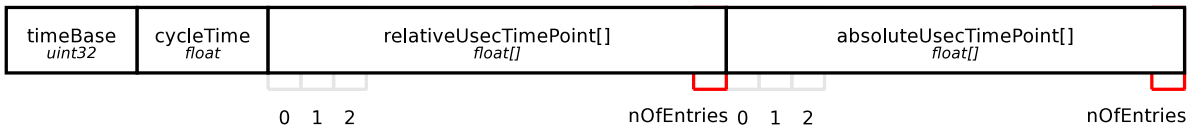

Figure 8.6: Format of the StatsStruct in the DDB memory area

The number of entries to monitor are saved in the numberOfEntries attribute. The pointers to the DDB memory area are timeBase, cycleTime, relativeUsecTimePoint and absoluteUsecTimePoint (those attributes are just depicted in Figure 8.6).

The attribute previousCycleStartTickTime holds the previous start of cycle time, GAMexecutionStartTimeCou saves the execution start time for this module (see method StartGAMMeasureCounter). The attribute trigger is a reference to the external time or triggering service of associated to the RealTimeThread; methods GetInternalCycleTickTime, GetLastProcessorTickTime and GetPeriodUsecTime of the TimeTriggeringServiceInterface class are used.

```
private:
   int32 numberOfEntries;

   int64 previousCycleStartTickTime;
   int64 GAMexecutionStartTimeCounter;

   uint32* timeBase;
   float* cycleTime;
   float* relativeUsecTimePoint;
   float* absoluteUsecTimePoint;

   GCRTemplate<TimeTriggeringServiceInterface> trigger;
```

The method Init initialise all attributes for construction to zero or NULL. Initialise initialises the RTCodeStatsStruct attributes using the parameters passed by; such method must be called in the *PRE PULSE* state, because during the loading of the library and objects DDB buffers are not available yet. Like in Figure 8.6 the code assumes that the first entry in the DDB data buffer is the time in microseconds relative to the monitoring measurement, the cycle time is the following entry, followed by the relative time points and the absolute time points.

The method StorePerformance computes performance for entry entry, i.e. must be called at the end of the execution of the entry-n GAM. StartGAMMeasureCounter signals a new performance measurament cycle.

```
   void Init();
public:
   RTCodeStatsStruct();
   ~RTCodeStatsStruct();

   bool Initialise(int32 entries,const int32* ddbBuffer,
```

```
        GCRTemplate<TimeTriggeringServiceInterface> trigger);

   bool StorePerformance(int entry);
   void StartGAMMeasureCounter();
```

8.3.2 ExecutionModule

[RealTimeThread.h, RealTimeThread.cpp]
The `ExecutionModule` class is a wrapper class that lets you call the `Execute` method of a GAM without specifing the `GAM_FunctionNumber` because is just embedded in the `ExecutionModule`, such class contain also a the instance of the `GAM` required.

The `Execute` method also measure the time consumed by the execution of the `GAM::Execute` method each time is called. The time is measured using the `BaseLib::Level0::HRT` class and is stored in the public attribute `lastAmountOfExecTimeCounts`.

```
private:
   GCRTemplate<GAM> gam;
   GAM_FunctionNumbers code;

public:
   int64 lastAmountOfExecTimeCounts;
   int64 lastExecutionTimeCounts;

   ExecutionModule(GCRTemplate<GAM> gam,GAM_FunctionNumbers code);
   ExecutionModule();

   bool Execute();

   GCRTemplate<GAM> Reference();
   bool ObjectLoadSetup(FString gamName, GAM_FunctionNumbers gamCode,
      StreamInterface* err, RealTimeThread& rt);
```

8.3.3 RealTimeThread, Status

[RealTimeThread.h, RealTimeThread.cpp]
Before introducing the most important class of MARTe we need to spend some words on its internal class `Status` whos interface follow in the text. Such internal class lets the `RealTimeThread` have a personal/private developed state machine that has no link to the one you can find in BaseLib (this could be a limitation for the general utilizability/applicability of MARTe).

The class `Status` has the attribute `status` that holds one of differents values that represent the state itself.

The attribute `requestedStatus` holds one of the previous values and is the required next state to move; `mSecTimeOut` is the timeout for the request to be completed. Some operator are redefined to ease the copy and compare of many `Status` classes.

The method `Request` requests a change in the status and waits by sleeping one millisecond by one millisecond for the accomplishment. `Refresh` copies the requested status on the `status` attribute. `SetMsecTimeOut` sets the timeout value.

```
class Status{
private:
   int32 status;
   int32 requestedStatus;
   int32 mSecTimeOut;
```

```
public:
   Status();
   ~Status();

   Status& operator=(int32 newStatus);
   bool operator == (int32 status)
   bool operator != (int32 status);

   bool Request(int32 newStatus);
   void Refresh();
   void SetMsecTimeOut(int32 timeOut);
};
```

We go over analising the `RealTimeThread` class that's the class representing a real time thread of execution, it inherits from `GCReferenceContainer`, and implements the `StartStopMessageHandlerInterface` and `HttpInterface` interfaces.

The first attribute is `sendFatalErrorMessage` a `MessageDeliveryRequest` object that enables the real-time thread send fatal error messages. The attribute `trigger` is a `TimeTriggeringServiceInterface` that lets the real-time thread synchronizing on some external events (one `RealTimeThread` has only one `TimeTriggeringServiceInterface` associated with it).

The attribute `smStatus` is the *State Machine Status* a state machine changing its state by message processing (i.e. it receives messages from the plant state machine and then convert it in internal BaseLib messages). Possible values of this state are (declared at the top of the cpp):

```
static const int32 SM_IDLE         = 101;
static const int32 SM_WAITING_PRE  = 102;
static const int32 SM_PREPULSE     = 103;
static const int32 SM_PULSING      = 104;
static const int32 SM_POSTPULSE    = 105;
static const int32 SM_INITIALISING = 106;
```

In the `SM_IDLE` it performs offline activities, in `SM_WAITING_PRE` it performs the `PulseStart` activities (method) before switching to `SM_PULSING`, in `SM_PULSING` it performs online activities, in `SM_POSTPULSE` it performs the `PostPulse` activities before switching to `SM_IDLE`, as we will see in Figure 8.7 is not really like that, there are some problems in the design of those combined state machines. The other state machine is implemented as the attribute `rtStatus` and is the status of the `RealTimeThread` object. Values are:

```
/// UNDEFINED STATE - RTApplicationThread has not been initialized
static const int32 RTAPP_UNDEFINED = 0xFFFFFFFF;
/// READY STATE - RTApplicationThread object has been initialised
static const int32 RTAPP_READY     = 0x00000000;
/// SAFETY STATE - When a major fault has been identified
static const int32 RTAPP_SAFETY    = 0x00000002;
```

The attribute `rtStatusRequest` switch status request. It is used to avoid the presence of the mutex semaphore into the real-time thread. The method that requires the change asks for the transition to the `RTAPP_UNDEFINED` and waits that `rtStatus` assumes the requested value.

Now comes **four** sets of vectors of GAMs that will be executed in differents combined states of the `RealTimeThread` and the plant state machine, Figure 8.7 is a schema with some of the messages transition between combined states.

The attribute `onlineModules` is a vector of execution modules for real time activities, the number of gams in the online activities vector is `nOfOnlineGams`.

The attribute `offlineModules` is a vector of execution modules for the offline activities, the number of gams in the onffline activities vector is `nOfOfflineGams`.

The attribute `safetyModules` is a vector of execution modules for safety activities, the number of gams in the safety activities vector is `nOfSafetyGams`; `safetyMsecSleep` is the millisecond sleep time if not module is specified in the `nOfSafetyGams` vector.

The attribute `initialisingModules` is a vector of execution modules that need an initialisation phase, the number of gams in the initialisation activities vector is `nOfInitialisingGams`; `numberOfInitialisingCy` is the number of consecutive initialising cycles and `maxnOfInitialisingCycles` is the maximum number of initialising cycles.

```
GCRTemplate <MessageDeliveryRequest> sendFatalErrorMessage;

GCRTemplate<TimeTriggeringServiceInterface> trigger;

Status smStatus;
Status rtStatus;
int32 rtStatusRequest;

ExecutionModule* onlineModules;
int32 nOfOnlineGams;

ExecutionModule* offlineModules;
int32 nOfOfflineGams;

ExecutionModule* safetyModules;
int32 nOfSafetyGams;
int32 safetyMsecSleep;

ExecutionModule* initialisingModules;
int32 nOfInitialisingGams;
int32 numberOfInitialisingCycles;
int32 maxnOfInitialisingCycles;
```

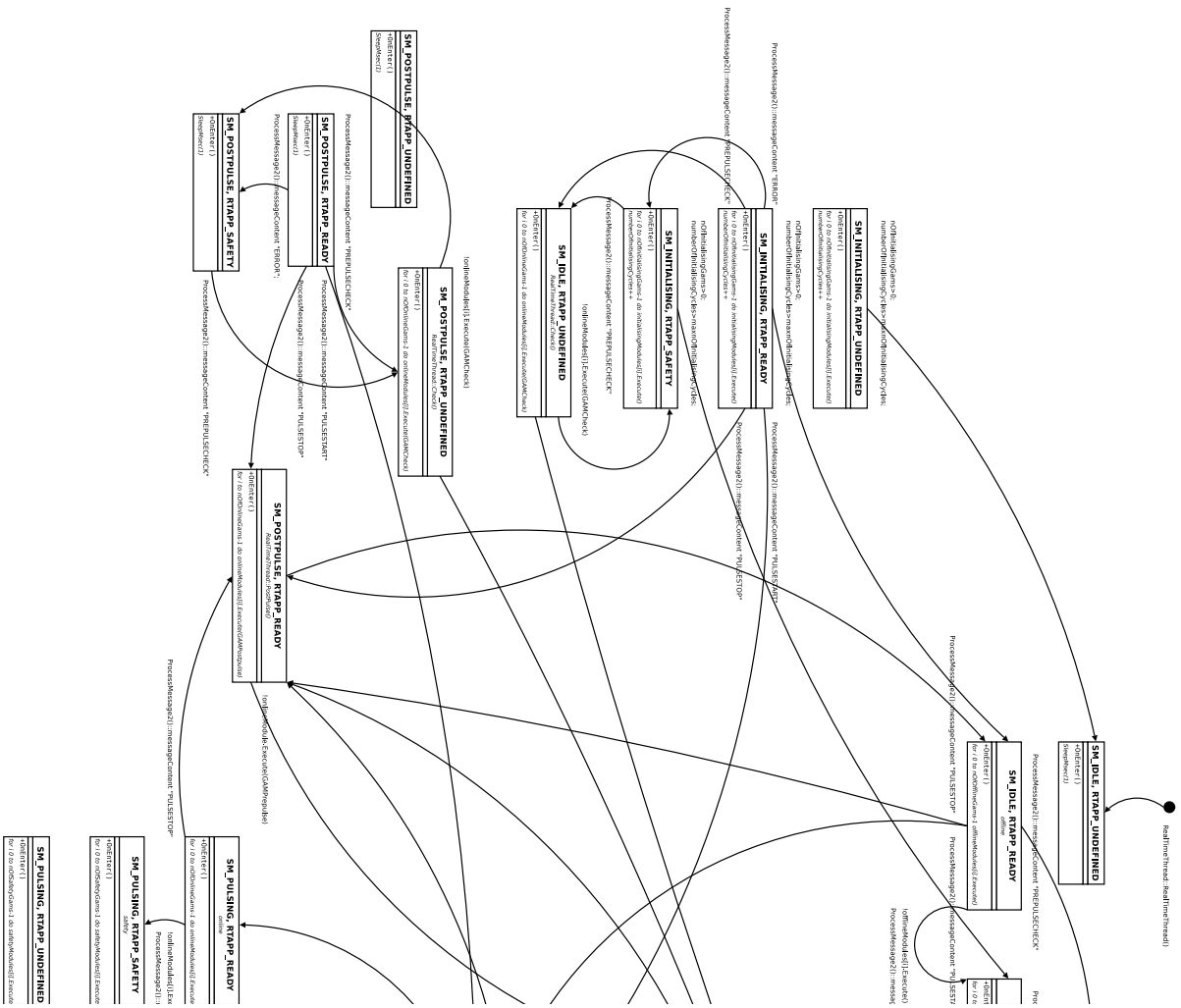

Figure 8.7: MARTe `RealTimeThread` current internal state machine

Continuing with the attributes of the `RealTimeThread` class we stop on `ddb` that is the DDB used by the real-time thread during its activities, all data between GAMs are written there.

One of the users of such DDB is the performance monitor, a performance monitor is implemented thanks to an `RTCodeSTatsStruct` object and monitors the performance of the RealTimeThread and GAMs. `performanceInterface` its a pointer to saved performance of the Real Time Thread and GAMs into the DDB.

The attribute `runOnCPU` identifies on which CPU the `RealTimeThread` is running; `threadID` is the thread identifier in the system, `priority` is the thread priority; `isThreadRunning` specifies if the thread is running or is dead, at the end `stopThread` requests the thread termination.

```
GCRTemplate<DDB> ddb;

RTCodeStatsStruct performanceMonitor;
DDBOutputInterface* performanceInterface;

int32 runOnCPU;
TID threadID;
int32 priority;
bool isThreadRunning;
bool stopThread;
```

The method `RTThread` implements the `RealTimeThread` activities; it executes a private state machine that runs the online, offline, safety or initializing GAMs depending on the combined state of the `RealTimeThread` (`smStatus`, `rtStatus`). The method `Start` starts the thread activities and the `Stop` stops the thread activities.

Then follow three methods that can be implemented inside the state machine, `PulseStart` runs the GAMs prepulse activities in response to pulse in progress message; `PostPulse` runs the GAMs postpulse activities in response to pulse in termination message; `Check` runs the GAMs check activities in response to a waiting for pre message. Note that this methods are really bounded to the plant system you are working for.

The method `ProcessHttpMessage` implements sources reflection using the HTTP/HTML format.

The method `ProcessMessage2` convert each message to the `RealTimeThread` in a state change to the combined internal state machine (`RTThread`), not all messages are valid for a state change. `SendMessageReply` prepares a `MessageReply` to the senders when processing message.

The method `HandleLevel1Message` stops the `RealTimeThread` on a semaphore and reinitializes all GAM according to the information stored in the CDB.

Methods `ObjectLoadSetup` and `ObjectSaveSetup` initializes and saves the thread parameters and the GAMs from the configuration file.

```
   void RTThread();
   bool Start();
   bool Stop();

   bool PulseStart();
   bool PostPulse();
   bool Check();

   virtual bool ProcessHttpMessage(HttpStream& hStream);

   virtual bool ProcessMessage2(GCRTemplate<MessageEnvelope> envelope);
   bool SendMessageReply(GCRTemplate<MessageEnvelope> envelope,bool ok);

   bool HandleLevel1Message(ConfigurationDataBase& cdb);
public:
   RealTimeThread();
```

```
   virtual ~RealTimeThread();

   bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

8.3.4 MARTeMenu

[MARTeMenu.h, MARTeMenu.cpp]
The class MARTeMenu follow with all its interface, it is only a specialization of the MenuContainer class to explicitly create a MARTe menu.

```
class MARTeMenu: public MenuContainer{
public:
   MARTeMenu();
   virtual ~MARTeMenu();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
};
```

8.3.5 MARTeContainer

[MARTeContainer.h, MARTeContainer.cpp]
The class MARTeContainer its a container of everything you need to make a control algorithm work in real-time, i.e. its a set of threads with application blocks (GAMs) and a memory data bus, a state machine, menues, a configuration database and some error catching mechanism. Such container class inherits from GCReferenceContainer and MessageHandler it implements the HttpInterface interface.

The first attribute, lastReceivedCDB is a reference to the last received CDB; sendFatalErrorMessage adds the cabability of sending Fatal Error messages.

stateMachineName is the name of the StateMachine object associated with the current set of instantiated object (not to be confused with the internal state of a RealTimeThread); level1Name is the name of the CODAS Level1 handling object; commTimeout is the communication timeout.

The attribute marteStaticMenu is the static part of the menu system, marteCommonMenu is a reference to the general menu system.

The attribute needStartStopMessage contains all object that need a start and stop message (objects that implement the StartStopMessageHandlerInterface) and realTimeThreads contains all real-time threads (RealTimeThread).

```
private:
   ConfigurationDataBase lastReceivedCDB;
   GCRTemplate <MessageDeliveryRequest> sendFatalErrorMessage;
   FString stateMachineName;
   FString level1Name;
   TimeoutType commTimeout;

   GCRTemplate <MenuContainer> marteStaticMenu;
   GCRTemplate <MenuContainer> marteCommonMenu;

   GCReferenceContainer needStartStopMessage;
   GCReferenceContainer realTimeThreads;
```

The method CreateMenuInterfaces creates the system menus dynamically.

The method StartAllActivities send the start message to all object in needStartStopMessage attribute; StopAllActivities send a stop message to all object in needStartStopMessage; ForwardMessageReques forwards a message to all object in needStartStopMessage.

The method `ProcessMessage` is the message handling routine, `ProcessHttpMessage` is the main entry point for `HttpInterface`.

```
   bool CreateMenuInterfaces();

   bool StartAllActivities();
   bool StopAllActivities();

   bool ForwardMessageRequest(GCRTemplate<MessageEnvelope> gcrtme);
   bool ProcessLevel1Message(ConfigurationDataBase& level1);

public:
   MARTeContainer();
   virtual ~MARTeContainer();

   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
   virtual bool ProcessHttpMessage(HttpStream& hStream);

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

8.3.6 Design Notes

A more dynamic structure to handle the performance data can be created, i.e. data can grow dynamically also if allocated statically each relative sample must be near the absolute one. As everything in this project is a bit strange that there is any generic interface to plug in a sort of generic performance code measurement tool.

The `RealTimeThread` has two state machines that interact between them; this interaction is not well defined and some spurious state are not well managed, Figure 8.7 illustrate it; the addition of methods `Check`, `PulseStart` and `PostPulse` complicate more and more the system and should be integrated in the state machine. Another notes is that if the plant system's state machine goes wrong also the `RealTimeThread`'s state machine goes wrong, no check is done.

In any case the private state machine of the `RealTimeThread` must be rewritten for generality but also for correcteness; in general the ideaa to have two state machines one that reflect the status of the plant and one that reflect the state of the `RealTimeThread` is a good idea, the final state machine will be a combination of those states.

Also the `MARTeContainer` is not so generic as it would be; infact it require a `level1Name` string that associate to them a *Level1* object that is a JET related entity, not to be founded in any other experiments.

8.4 MARTe

MARTe as we said is no more then a `MarteContainer` but not all components in MARTe are instantiated by the `MarteContainer` so we need some more code that loads for example the state machine or the level1 interface. Such code is in the following executables that are really executables and so can also be executed by the Operating System on which are loaded. The only difference between those is that `MARTe` is a normal executable running on a console and `MARTeService` is a service, it doesn't require a console to execute, both works in Microsoft Windows© and UNIX like operating systems. Both executables require only one command line paramenter that is the configuration file form wich they load all required serialized components. /** Real time manager. */

MARTe

[MARTe.cpp]
Note that MARTe.h is not included in MARTe.cpp, MARTe.h is to be considered old and must be deleted from the directory.

```
static bool initialized;
static EventSem sem;

int main(int argc, char* argv[]);
int InitGlobalContainer(const void* fileName);
int StartMARTeActivities();
int StopMARTeActivities();
int MARTeMenu();
```

MARTeService

[MARTeService.h, MARTeService.cpp]

```
class MARTeService: public WindowsServiceApplication {
   ConfigurationDataBase cdb;

private:
   virtual bool ServiceInit();
   virtual bool ServiceStart();
   virtual bool ServiceStop();

public:
   MARTeService(const char *serviceName,const char *title):
      WindowsServiceApplication(serviceName,title){
    }
};
```

8.5 Design Notes

There are some clean up to do in the main directory of MARTe and also in *MarteSupportLib* directory.

In the main directory would be likely to remove the *MARTeTrainer.cpp* and the text.cpp applications used to test the whole system. The file RealTimeThreadPool.h is also to be considered old stuff and must be deleted, the thread's pool is now integrated in the MarteContainer, also MARTe.h must deleted. For the same reasons the MObjects and the MObjectPool are not existents right now and the code of MARTe.h must be cleaned up.

In the directory *MarteSupportLib* all the **MPIC** stuff must be moved.

Chapter 9

MARTe Components

Before starting this chapter we will do the best to explain how BaseLib/MARTe is a well designed component framework were each component is a GAM. Drivers, that we saw in the previous chapter are *Generic Acquisition Module*s (GACQM) and infact are dissimle to GAMs. A driver, for example, when loaded runs until the library unloads but a Generic Application Module (GAM) runs scheduled by the `MarteSupportLib::RealTimeThread`. Keep in mind that GAM and GACQM are basically two differents entities that execute in a far different way. Remember that GAMs are control blocks in a control system algorithm but GACQMs are not control blocks also if they provide data input/output with physical hardware; to fill this gap between these two worlds some special GAMs were introduced: `InputGAM` and `OutputGAM`. These GAMs are the source and the sink blocks in a control system schema; an `InputGAM` is a provider of data (source) and has a link to a specific `GenericAcqModule`; an `OutputGAM` is a consumer of data (sink) and has a link to a specific `GenericAcqModule`. Both are components with a special characteristic: they interact with device drivers. Figure 9.1 try to highlight the different domains of which any component of BaseLib/MARTe belongs, i.e. as above, an IOGAM belongs to the *GAMs world* and so its a high level component. In the GAMs world the entity that make GAMs interacting is the `BaseLib::Level5::DDB` (Dynamic Data Buffer). Note that in Figure 9.1 GAM components are depicted at a macro level and `DDBInterface` components it uses are just hidden inside the GAMs itselfs.

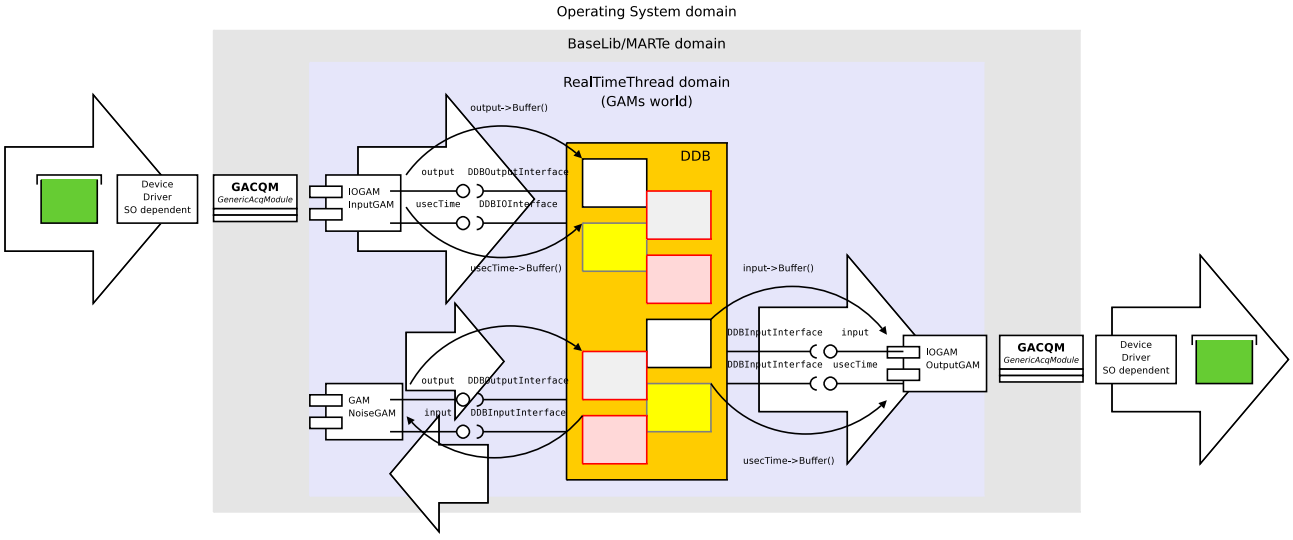

Figure 9.1: Different component's domains in MARTe. GAMs (like the `NoiseGAM` depicted) interact with the system reading and writing on the DDB, obviously they belong to the *GAMs world* like the `IOGAM`s.

Using the DDB different GAMs (the basic executing component of the framework) exchange datas. Data on the DDB must be in a predefined format, i.e. all data must be written or read in `float` or `int`. Through the chapter different GAMs are presented and not all GAMs agree on the same data format, instead some expect `float` data and some other `int` data.

This description terminate the whole walk on the components of BaseLib/MARTe validating the component design model adopted during the development of the framework. This kind of component object model design lets start a series of further development to create a simple IDE that helps the user smiply design its control system application and output a configuration that is ready to work in BaseLib/MARTe on the fly.

9.1 Input Output Generic Application Modules (IOGAMs)

This section addresses how a device driver (GACQM) can interact in the GAMs world: it needs a component interface called IOGAM, that infact is a GAM and can be an `InputGAM` or an `OutputGAM`. The UML schema in Figure 9.2 depict the inheritance relationship. Usually an input device can simply act as a counter and infact producing only timing references, some other also data; by the way `InputGAM`s are specialized in `TimeInputGAM` that are blocking GAMs that waits for the synchronization from a device driver.

Another specializations of the `InputGAM` are the `FilteredInputGAM` and the `TimeFilteredInputGAM`. Those act as median filters on the input signal, if the `GetData` of the `GACQM` is synchronized, i.e. is blocking, the `TimeFilteredInputGAM` decimate the frequency of the input signal.

Then follow a list of the involved classes in this section:

- InputGAM
- TimeInputGAM
- FilteredInputGAM
- TimeFilteredInputGAM
- OutputGAM

Figure 9.3 gives a logical understanding of the links between the different component objects (that belongs to different domains) for an acqusition device driver that offer also timing synchronization facilities (in the Figure an `InterruptDrivenTTS` driver is depicted). Note that a GACQM to produce also timing signals must have a link with a `TimeInputGAM` not only an `InputGAM`. The `TimeInputGAM` by the way must hold a link to GACQM and to the correct `InterruptDrivenTTS`, i.e. the one that belongs to the same device driver.

InputGAM

`[InputGAM.h, InputGAM.cpp]`
The class `InputGAM` extends `GAM`, `HttpInterface` and `MessageHandler`. The `InputGAM` is a special GAM that move data, when is ready, from a GACQM to a memory buffer in the DDB, then the happenning of the event is triggered down to other objects.

An `InputGAM` represent a single device driver that produce input data; the first attribute is infact a `GCRTemplate` templetized on a `GenericAcqModule`, i.e. it refers to a single class `GenericAcqModule`, in other words to one device driver. The attribute `output` is a pointer to a `DDBOutputInterface` that

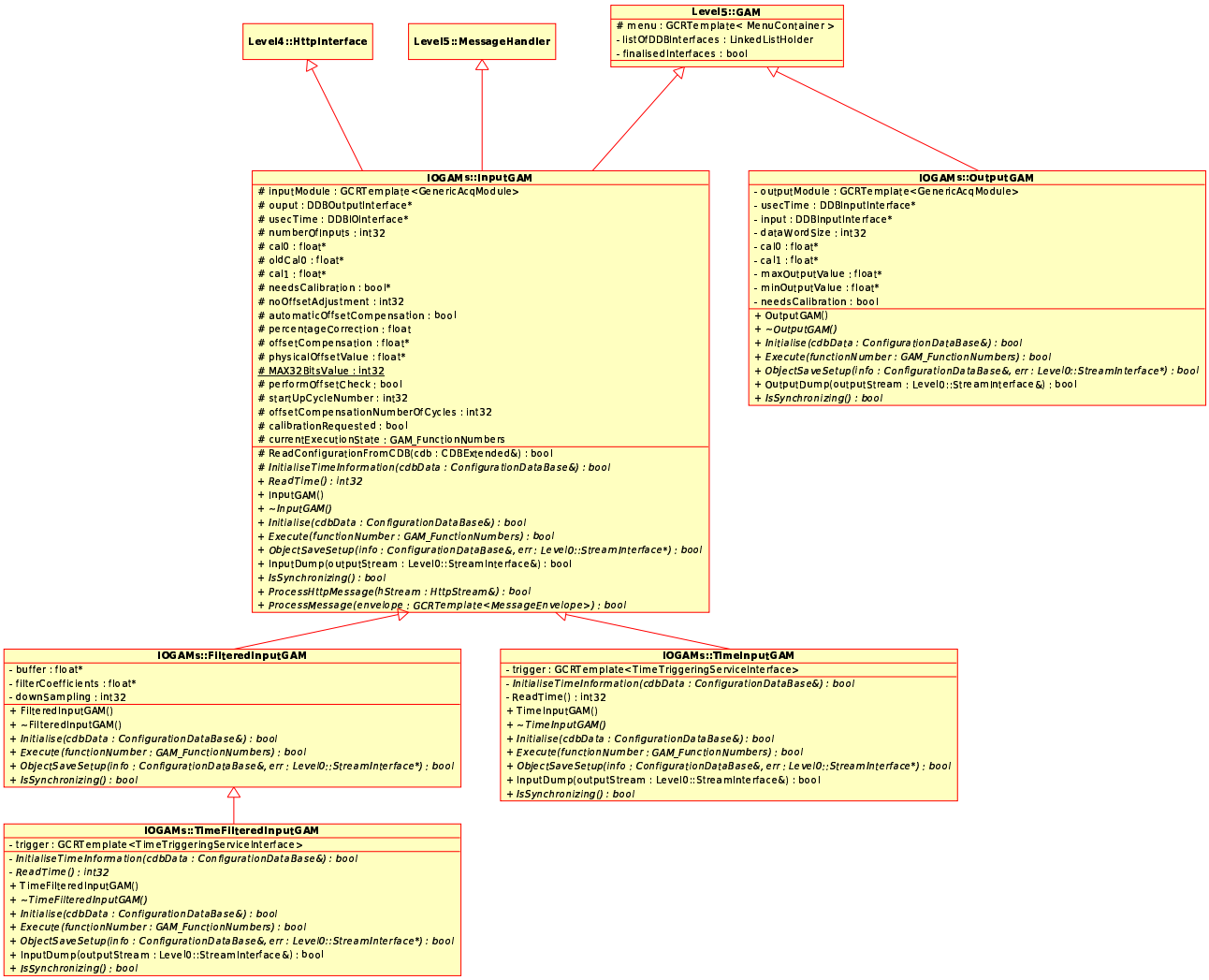

Figure 9.2: MARTe Input Output GAMs infrastructure

is a `DDBInterface` that has the `Write` method, thanks to complying to this interface the `InputGAM` has a buffer where to **write** in the DDB. This IOGAM has an `DDBOutputInterface` because looking at the logical block `InputGAM` it produces data, at the end of the data production (all data has been written to the DDB) the code calls `DDBOutputInterface::Write()`.

There is another `DDBInterface` attribute that defines an IO activity, this is `usecTime`. This attribute is define as an IO interface because a data acquisition board can also write or read a time stamp. The current implementation only read time stamps.

The attribute `numberOfInputs` is the number of input of the board or also the size of the packet to be readed on the board in words, a word in this case is an `int32` (see *Level5/DDBInterface.h:BufferSize()*).

Then it follows differents calibration factors vector attributes that lets the software convert data from the board data format, `cal0`, `oldCal0` and `cal1`. `needsCalibration` is an array used to specify if a signal needs calibration. We now have a look on how these variables are used.

The following piece of code is an example from `InputGAM::Execute(functionNumber=GAMOnline)` about how the previous software calibration variables are used to perform signal value conversion from `integer` to `float`. `cal1[]` is a multiplicative factor and `cal0[]` is a additive offset.

```
   float* floatBuffer = (float*)output->Buffer();
   int*   intBuffer   = (int*)  output->Buffer();
// ...
```

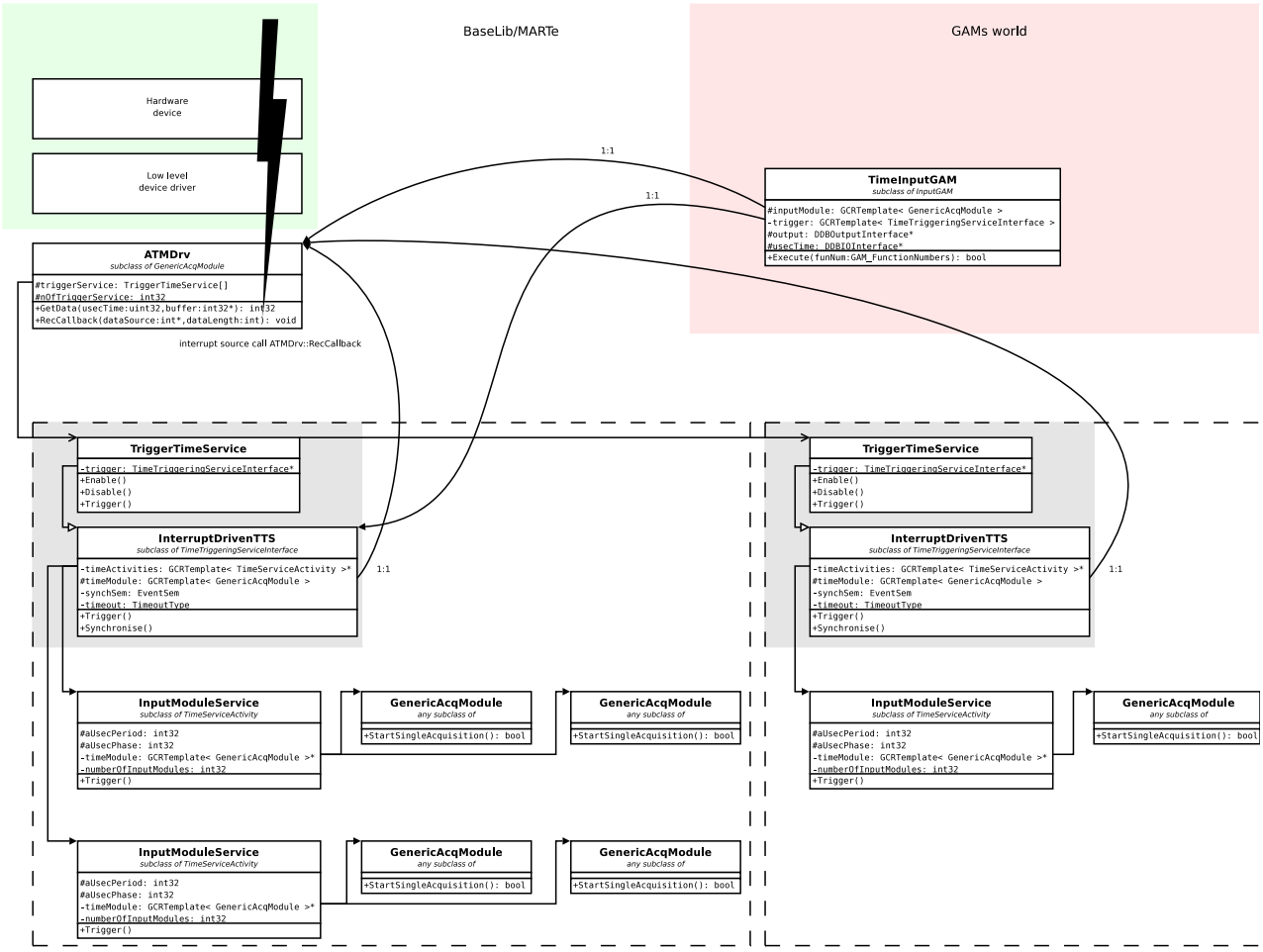

Figure 9.3: Existent links between a `GenericAcqModule` a `TimeInputGAM` and the TTSI infrastructure

```
if(inputModule->GetData(time,output->Buffer()) == -1){
   AssertErrorCondition(FatalError,"InputGAM::Execute:␣Module␣%s"
      "GetData␣Failed␣for␣driver␣%s",Name(),inputModule->Name());
   return False;
}
for(int sig=0; sig<numberOfInputs; sig++)
   if(needsCalibration[sig])
      floatBuffer[sig] = (intBuffer[sig]*cal1[sig] + cal0[sig]);
```

This code imply that the internal data exchange between GAMs can be or in `integer` or in `float` data format. If the attribute `needsCalibration[]` is true values are in `float` otherwise in `int`.

All attributes that follow are used to calculate a new calibration value `cal0` automatically. A first set of `offsetCompensation` values are computed at startup in `startUpCycleNumber` iterations, then, during offline `cal0` factors are updated. The code that follows is executed at `InputGAM::Execute(functionNumber=GAM` after the board has read inputs.

```
if(automaticOffsetCompensation == False) return True;
startUpCycleNumber++;
for(int sig = 0; sig < numberOfInputs; sig++){
   if(needsCalibration[sig]) {
      floatBuffer[sig]          = (intBuffer[sig]*cal1[sig] + cal0[sig]);
      offsetCompensation[sig] +=  floatBuffer[sig] - cal0 \begin{center}
\vspace{0.5cm}
```

```
%  \includegraphics[width=30mm]{PPCClogoRed.eps}
  \includegraphics[width=30mm]{PPCClogoBlack.eps}
 \end{center}[sig];
      }
   }
   performOffsetCheck = True;
   if(startUpCycleNumber == offsetCompensationNumberOfCycles)
      calibrationRequested = False;
```

The attribute `automaticOffsetCompensation` enables automatic offset compensation. The attribute `startUpCycleNumber` is a counter of computed startup cycles in which the `offsetCompensation` values were updated for each channel. `offsetCompensation` is a float array where each value is equal to the average of the input signal during offline operations and computed during `GAMStartUp` phase.

The flag `performOffsetCheck` marks the transition between `GAMStartUp` and `GAMOffline`, at this point the code performs the maximum correction check, if the check fails, the GAM return `False`, causing the `RealTimeThread` to perform safety operations. `offsetCompensationNumberOfCycles` is the number of startup cycles to perform for each calibration. `calibrationRequested` is a flag set to `True` when a data calibration is requested.

Other attributes are used in the following piece of code, `InputGAM::Execute(functionNumber=GAMOffline)`:

```
if(performOffsetCheck){
   for(int sig = 0; sig < numberOfInputs; sig++){
      if((needsCalibration[sig]) && (startUpCycleNumber != 0)){
         if(noOffsetAdjustment[sig] == 1) continue;
         offsetCompensation[sig] = offsetCompensation[sig]/startUpCycleNumber;
         cal0[sig] = physicalOffsetValue[sig] - offsetCompensation[sig];
         float offsetPercent = fabs((cal0[sig] - oldCal0[sig])/(cal1[sig]*MAX32BitsValue));
         if(offsetPercent > percentageCorrection){
            AssertErrorCondition(Warning,
               "InputGAM::Execute::␣Module␣%s:␣Offset␣NOT␣corrected"
               "␣%f␣>allowed␣%f␣for␣driver␣%s,␣signal␣%d,␣oldcal0␣=␣%e,␣newcal0=%e",
               Name(),offsetPercent,percentageCorrection,inputModule->Name(),sig,
               oldCal0[sig],cal0[sig]);
            cal0[sig] = oldCal0[sig];
         }
      }
   }
   startUpCycleNumber = 0;
   ConfigurationDataBase cdb; ObjectSaveSetup(cdb, NULL); UpdateGAMPersistentCDB(cdb);
   performOffsetCheck = False;
}
```

The attribute `noOffsetAdjustment` is an array of `int32` and if not 1 lets the calibration routine proceed.

The attribute `physicalOffsetValue` is an array of defualts offset values for each channel, `MAX32BitsValue` is the maximum value of a signed 32 bits variable; `percentageCorrection` is the maximum allowed percentage correction.

The last attribute `currentExecutionState` is the current execution state. Then the list of all attributes come.

```
protected:
   GCRTemplate<GenericAcqModule> inputModule;

   DDBOutputInterface* output;
   DDBIOInterface* usecTime;
   int32 numberOfInputs;

   float* cal0;
   float* oldCal0;
```

```
    float* cal1;
    bool* needsCalibration;

    bool automaticOffsetCompensation;
    int32* noOffsetAdjustment;
    float* offsetCompensation;
    float* physicalOffsetValue;
    float percentageCorrection;

    static const int32 MAX32BitsValue;
    bool performOffsetCheck;
    bool calibrationRequested;
    int32 startUpCycleNumber;
    int32 offsetCompensationNumberOfCycles;

    GAM_FunctionNumbers currentExecutionState;
```

For the first time we have no `ObjectLoadSetup` method; such method is here substituite by the `ReadConfigurationFromCDB` method, in this case a `CDBExtended` is required, not a simpler `ConfigurationDataBase` `ReadConfigurationFromCDB` doesn't suffice and another method `InitialiseTimeInformation` is necessary to initialise the `usecTime` `DDBInterface`. Keep in mind that the `ObjectLoadSetup` is implemented in the `GAM` superclass.

The method `ReadTime` reads timing informations, it is only used if the module is a standard `InputGAM`. The method `Initialise` initialises the module from a CDB, it uses `ReadConfigurationFromCDB` and `InitialiseTimeInformation`; the user must specify the following parameters:

- `BoardName` is the name of the input driver to be used;
- `UsecTimeSignalName` is the name of the time signal in μsec;
- `Signals` which contains the informations for the ddb interface and the calibrations (if needed);

The method `Execute` executes the module functionalities. The functions or states of a GAM are defined by the enum `GAM_FunctionNumbers` in *BaseLib/Level5/GAM.H* an `InputGAM` implements the following states:

- `GAMStartUp`
- `GAMPrepulse`
- `GAMOffline`
- `GAMOnline`

The method `ObjectSaveSetup` implements the saving of the parameters to a `ConfigurationDataBase`; `InputDump` implements the dump of the outputs, with their values, names and calibration factors; `IsSynchronizing` returns `True` if the module has delayed acquisition or the associated driver is synchronising.

The method `ProcessHttpMessage` implements the `HttpInterface` interface and `ProcessMessage` implements the `MessageHandler` interface accepting a message with initialisation requirement.

```
    bool ReadConfigurationFromCDB(CDBExtended& cdb);

    virtual bool InitialiseTimeInformation(ConfigurationDataBase& cdbData);
    virtual int32 ReadTime();

public:
    InputGAM();
    virtual ~InputGAM();
```

```
virtual bool Initialise(ConfigurationDataBase& cdbData);
virtual bool Execute(GAM_FunctionNumbers functionNumber);

virtual bool ObjectSaveSetup(ConfigurationDataBase& info, StreamInterface* err);
bool InputDump(StreamInterface& outputStream);

 virtual bool IsSynchronizing();

 virtual bool ProcessHttpMessage(HttpStream& hStream);
 virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
```

FilteredInputGAM

[FilteredInputGAM.h, FilteredInputGAM.cpp]
The class FilteredInputGAM inherits from InputGAM and acts as a medianfilter on downsampling samples of every data source.

The core of this class is the Execute method; each GAM_FunctionNumbers that differ from GAMPrepulse execute the following code:

```
// Reset the OutputInterface
   int sig;
   for(sig=0; sig < output->BufferWordSize(); sig++) floatBuffer[sig]=0.0;
// Copy the data in the buffer and peform filtering activities
   for(int32 i = 0; i < downSampling; i++ ) {
      if(inputModule->GetData(time,(int32*)buffer,-i) == -1) {
         AssertErrorCondition(FatalError,"InputGAM::Execute:: Module %s GetData Failed"
            " for driver %s", Name(), inputModule->Name());
         return False;
      }
      for(sig=0; sig < output->BufferWordSize(); sig++)
         floatBuffer[sig] += buffer[sig]*filterCoefficients[i];
   }
// Copy last value of integer entries in the DDB
   for(sig = 0; sig < output->BufferWordSize(); sig++) {
      int32 *intB = (int32 *)buffer;
      if(inputModule->GetData(time,(int32 *)buffer,0) == -1) {
         AssertErrorCondition(FatalError,"InputGAM::Execute:: Module %s GetData Failed"
            " for driver %s",Name(), inputModule->Name());
         return False;
      }
      if(needsCalibration[sig])  intBuffer[sig]=intB[sig];
   }
```

In such code the floatBuffer array is first filled with zeroes and then each buffer of the circular data buffer of the GetData method of a GACQM is readed, the values in the array are updated with the currently readed ones multiplicate by the filterCoefficients[] factor. The integer downSampling attribute is the frequency divide factor, i.e. every downSampling samples readed from the acquisition board a single event is reported to all the system. Last cycle in the above code seemes to be wrong, probably to be removed.

As we can know from the previous code data are first collected inside the driver, using for example a circular buffer, then copied to the area pointed to the buffer attribute and after some simple calculation saved in the DDB.

This component assume to has float values as input and float values as output, i.e. the GACQM associated with this GAM must produce float values. This component require that the associated GACQM has a circular buffer of at least downSampling buffers.

All methods that follows behave in the same way as described for the class InputGAM.

```
private:
   float* buffer;
   float* filterCoefficients;
   int32 downSampling;

public:
   FilteredInputGAM();
   virtual ~FilteredInputGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool IsSynchronizing();
```

TimeFilteredInputGAM

[TimeFilteredInputGAM.h, TimeFilteredInputGAM.cpp]
A TimeFilteredInputGAM class inherits from FilteredInputGAM adding a reference to a triggering service facility (GCRTemplate<TimeTriggeringServiceInterface>). This timing triggering service facility is used in the Execute method that basically call the same method of the superclass first waiting on the Synchronise method of the attribute added.

```
private:
   GCRTemplate<TimeTriggeringServiceInterface> trigger;

   virtual bool InitialiseTimeInformation(ConfigurationDataBase& cdbData);
   virtual int32 ReadTime();

public:
   TimeFilteredInputGAM();
   virtual ~TimeFilteredInputGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   bool InputDump(StreamInterface& outputStream);
   virtual bool IsSynchronizing();
```

TimeInputGAM

[TimeInputGAM.h, TimeInputGAM.cpp]
A TimeInputGAM as a TimeFilteredInputGAM does on a FilteredInputGAM adds a reference to a triggering service facility to the InputGAM class from which it inherits.

The only thing that changes, compared to the InputGAM class, is that on the Execute method before calling the superclass's one the TimeTriggeringServiceInterface::Synchronise method is called.

```
private:
   GCRTemplate<TimeTriggeringServiceInterface> trigger;

   virtual bool InitialiseTimeInformation(ConfigurationDataBase& cdbData);
   virtual int32 ReadTime();

public:
   TimeInputGAM();
   virtual ~TimeInputGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
```

```
virtual bool Execute(GAM_FunctionNumbers functionNumber);

virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
bool InputDump(StreamInterface& outputStream);

virtual bool IsSynchronizing();
```

OutputGAM

[OutputGAM.h, OutputGAM.cpp]
The class OuputGAM extends GAM; OutputGAM is a special GAM that move data, when is executed, from a memory buffer in the DDB to a GACQM.

An OutputGAM represent a single device driver that output data; the first attribute is infact a GCRTemplate templetized on a GenericAcqModule, i.e. it refers to a single class GenericAcqModule, in other words to one device driver. The attribute input is a pointer to a DDBInputInterface that is a DDBInterface that has the Read method, thanks to complying to this interface the OutputGAM has a buffer where to **read** in the DDB. This IOGAM has an DDBInputInterface because looking at the logical block OutputGAM it consumes data, at the start of the data consumation (all data has to be readed to the DDB) the code calls DDBInputInterface::Read() that request valid data on the buffer to the DDB.

There is another DDBInterface attribute that defines another input activity, this is usecTime. This attribute is define as input interface because an output board can only read time stamps.

The attribute dataWordSize is the number of output of the word or also the size of packet to be written on the board in words, a word in this case is an int32 (see *Level5/DDBInterface.h:BufferSize()*).

Then it follows differents calibration factors vector attributes that lets the software convert data to the board data format, cal0, cal1, maxOutputValue and minOutputValue. needsCalibration is an array used to specify if a signal needs calibration/conversion. The following piece of code is an example from OutputGAM::Execute() about how the previous software calibration variables are used to perform signal value conversion from float to integer. cal1[] is a multiplicative factor and cal0[] is a additive offset.

```
for(int sig=0; sig < dataWordSize; sig++)
   if(needsCalibration[sig]) {
      float output = floatBuffer[sig];
      if(output > maxOutputValue[sig]) output=maxOutputValue[sig];
      if(output < minOutputValue[sig]) output=minOutputValue[sig];
      float temp  = (output - cal0[sig])*cal1[sig];
      int tempInt = FastFloat2Int(temp);
      intBuffer[sig] = tempInt;
   }
```

Obviously the attribute maxOutputValue is the maximum output value and minOutputValue is the minimum output value; cal0 and cal1 has the same meaning as in the InputGAM module.

```
private:
   GCRTemplate<GenericAcqModule> outputModule;

   DDBInputInterface* input;
   DDBInputInterface* usecTime;
   int32 dataWordSize;

   float* cal0;
   float* cal1;
   float* maxOutputValue;
   float* minOutputValue;
   bool* needsCalibration;
```

The method `Initialise` initialises the IOGAM reading the configuration from a CDB; `Execute` executes the data transfer from memory in the DDB to the GACQM, the only implemented `GAM_FunctionNumbers` states are: `GAMOffline` and `GAMOnline`.

```
public:
   OutputGAM();
   virtual ~OutputGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   bool OutputDump(StreamInterface& outputStream);
   virtual bool IsSynchronizing();
```

9.1.1 Design Notes

In MARTe the directory *MARTe/IOGAMs* contains a set of classes that are to compiled in a unique sub library of the framework, the library in a UNIX system has the name *libIOGAM.so*. By the way exist the file `IOGAM.cpp` and is empty, this file is needed to generate *libIOGAM.so* and makes the compiler happy.

Some more words about the IOGAMs. The difference between an `InputGAM` and a `TimeInputGAM` is basically in the `Execute` method, moreover a `TimeInputGAM` has a reference to a `TimeTriggeringServiceInterface` Such interface lets the `TimeInputGAM` call its `Synchronise` method before executing the normal `Execute` method of the GAM.

How an IOGAM works depends heavily on how a GACQM is implemented. The most important method regarding our discussion is the `GetData`. The `GetData` method must be non blocking letting a caller read an internal data buffer. Some implementation, like the ATMDrv (that we will see in the next section), lets the user deciding if the `GetData` must be blocking or non blocking. Such solution probably is not the best one possible because can confuse if seen at high level. For example it change completely how a `FilteredInputGAM` works; but also change the way in which a `TimeInputGAM` react. Thinking about the simplest case, i.e. the `TimeInputGAM`, on `Execute` it calls `InterruptDrivenTTS::Synchronise` that waits for interrupt, then it calls `GenericAcqModule::GetData` that is blocking and waits from another interrupt dividing the frequency of data arrivals. Keep in mind that if you have a synchronising `GetData` your synchronising method must always return `True`. A stronger design must be applied. Something similar happens with the `FilteredInputGAM` and the `TimeFilteredInputGAM`. Furthermore such classes act as a downsampler not as a median filter.

The following code require a double discussion. First of all is notable that the same area of memory can be handled as a float array or as a integer array, consistency about type casting aslo at runtime is compulsatory.

A second notes is about performance and efficiency last two code lines can be executed by FPU optimized SIMD instructions. Think about the Altivec unit, it has one single assemply instruction that operates on the same data the instruction is the *Multiply and Add*. Other processor have its own SIMD instruction that must be exploited by our code. Note that SIMD instruction are nowadays a common requirement of all brand new processors also for home user computers.

```
   float* floatBuffer = (float*)output->Buffer();
   int*   intBuffer   = (int*)  output->Buffer();

// ...
```

```
for(int sig=0; sig<numberOfInputs; sig++)
   if(needsCalibration[sig])
      floatBuffer[sig] = (intBuffer[sig]*cal1[sig] + cal0[sig]);
```

9.2 Generic Acquisition Modules (GACQM)

In this section we are going to analyse the directories contained in the directory *MARTe/IOGAMS*. Note that sources contained in such directory are just been commented in the previous section. Directories we analyse here contain a device driver (GACQM) each. In this section we are going to speak about device drivers, i.e. `GenericAcqModules`. Figure 9.4 is the UML class diagram of each device driver class object existent in the previous cited directory.

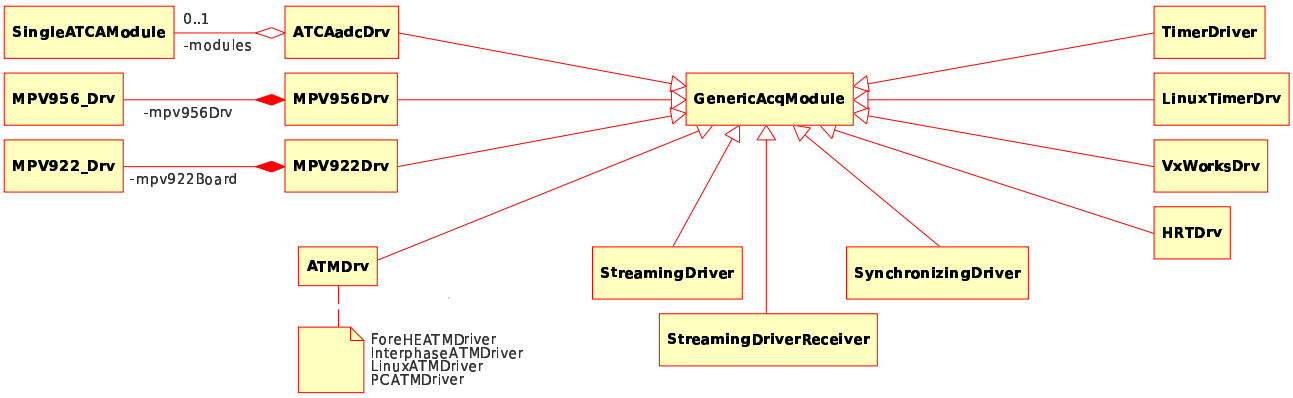

Figure 9.4: MARTe `GACQMs`

We are not going to analyse all device drivers in the library because each device driver is application specific and is not interesting in general. We just focus our attention to two different type of device driver's sychronization. A GACQM that synchronise itself via a `InterruptDrivenTTS` interface (`ATMDrv`) and a GACQM that synchronise itself via a `DataPollingDrivenTTS` interface (`ATCAadcDrv`).

In Figure 9.5 there is the UML class diagram of such GACQMs. Note that between different implementation of the `GenericAcqModule` class there are just little differences.

9.2.1 InterruptDrivenTTS GACQM (ATMDrv)

A `GenericAcqModule` has as an attribute, an array of `TriggerTimeService` each of that can hold an implementation of `TimeTriggeringServiceInterface`. Let's start with the case of a GACQM holding at least one `InterruptDrivenTTS`. Let's also assume that our device driver generate interrupts on data arrival. What happens on the `TimeInputGAM` side when executed by the `RealTimeThread`? We are assuming that the `GenericAcqModule::GetData` is not blocking, i.e. the module is not synchronising. In Figure 9.6 the UML sequence diagram show the method calling sequence.

We try to illustrate what happens in the schema in Figure 9.6 step by step in the following:

0. the `RealTimeThread` calls `TimeInputGAM::Execute` method;
1. `TimeInputGAM::Execute` as first action calls `InterruptDrivenTTS::Synchronise` method;
2. this method simply calls `EventSem::ResetWait` method that pauses the `RealTimeThread` giving the control of the CPU to the operating system that virtually can execute another process.

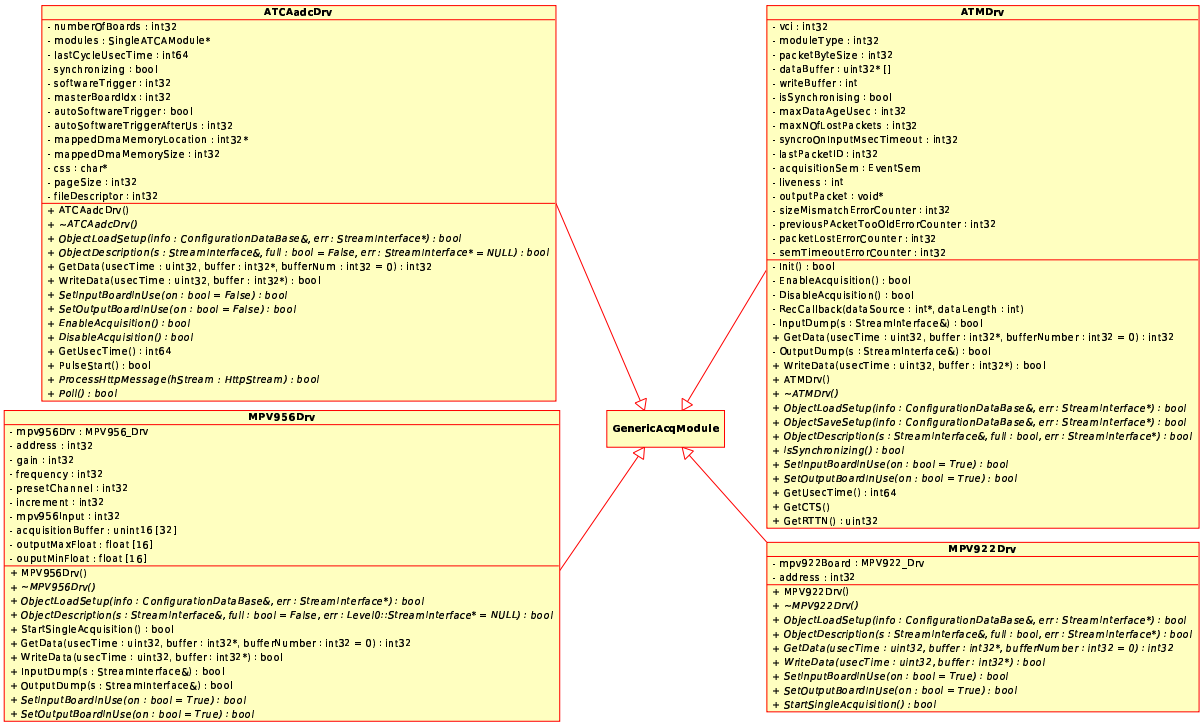

Figure 9.5: MARTe's hardware `GACQMs`

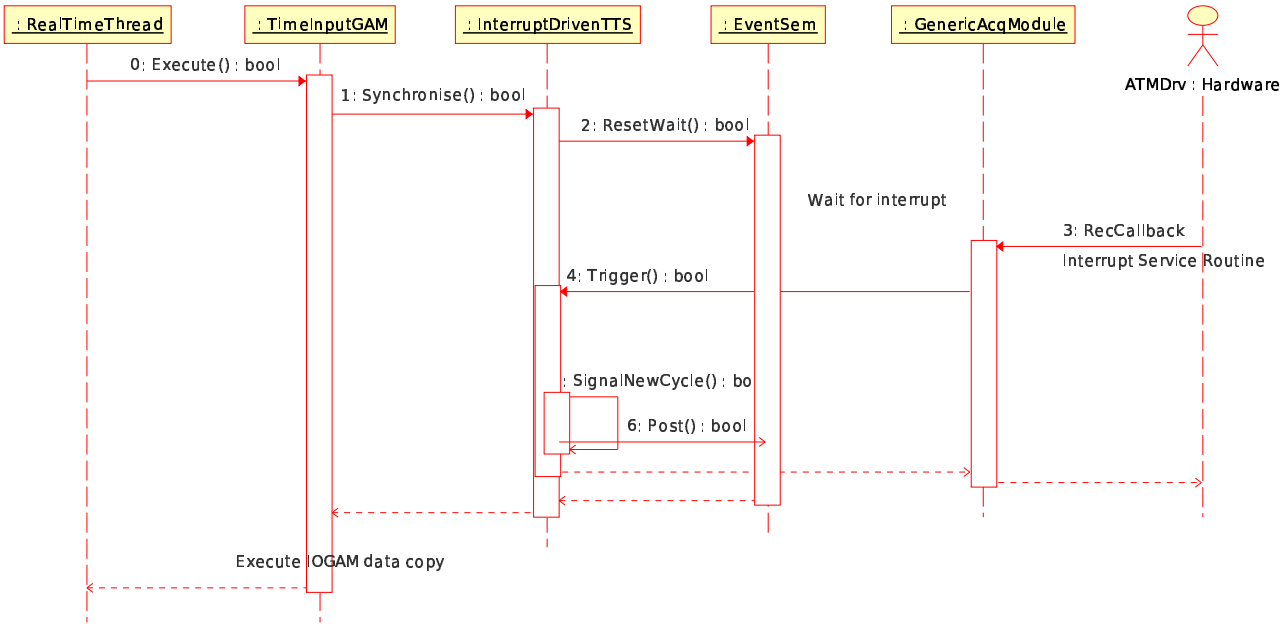

Figure 9.6: MARTe UML sequence diagram of an `InterruptDrivenTTS` compatible `GenericAcqModule`

3. the device generate an interrupt, i.e. some data is ready to transfer to the main memory (or the transfer is just finished), the associated ISR is called (`GenericAcqModule::RecCallback`;

4. `GenericAcqModule::RecCallback` as the last instruction calls method `Trigger` of all its registered `TimeTriggeringServiceInterface`;

5. the method `InterruptDrivenTTS::Trigger` as its last instruction calls `SignalNewCycle`;

6. the only action that performs `SingnalNewCycle` is to call `EventSem.Post` that wakeup the `RealTimeThread` that can continue its execution exiting from `InterruptDrivenTTS::Synchronise` and copying the data to the DDB (calling `GenericAcqModule::GetData` from the `TimeInputGAM` that is not blocking.

Follows the usual class analysis of the implementing driver, we have to punctualise that the previous described behaviour must only be achieved without a blocking `GenericAcqModule::GetData` method.

The following `ATMDrv` can be associated with an `InputGAM` but also with a `OutputGAM` because can receive and transmit data.

ATMDrv

`[ATMDrv.h, ATMDrv.cpp]`
The code that follow is specifically from the *ForeHEATMDriver* directory; all other ATM driver implementations have a similar high level driver that interface to the BaseLib/MARTe.

The class `ATMDrv` inherits only from `GenericAcqModule`. An ATM network board is not very common today but permits deterministic transmission times exploiting QoS techiniques both on the trasmitter/receiver side and on the switches of the network. The first attribute, `vci` is the virtual channel identifier number of the ATM network channel associatedwith the class instance. `moduleType` identifies if the instance of the module behave as an output or as an input, it can also be undefined if is not just setted. The attribute `packetByteSize` is the input/output buffer size in byte.

The following attributes are used by a receiver channel. `dataBuffer` is a triple buffer for receiving data, `nOfDataBuffers` is defined as 3 in the source code; the class's code allocate three data buffer of size `packetByteSize` and saves its addresses in the `dataBuffer` array. `writeBuffer` is the index of the write only buffer, i.e. the index of the buffer that will be written by the next interrupt data transfer ISR. The attribute `isSynchronising` specifies if the receiver module is synchornizing or get the latest fullfilled buffer.

The attribute `maxDataAgeUsec` it is used to decided if the read data are ready or not; `maxNOfLostPackets` is the maximum number of lost packets after a rollover; `syncroOnInputMsecTimeout` is the timeout in *msec* to wait on a synchronization event; `lastPacketID` is the ID of the last packet received.

The attribute `acquisitionSem` is an event semaphore to be used in case of a synchornization acquisition mode, i.e. a blocking `GetData`. The attribute `liveness` get the liveness of the driver.

The attribute `outputPacket` is a pointer to the actual packet to be sent. It's the only attribute associated to a transmitter module.

Finally follows a series of counter attributes, used in transmitter and receiver modules; `sizeMismatchErrorCount` counts the number of size mismatch events; reset it as soon as the first correct packet is received. `previousPacketTooOldErrorCounter` counts the number of packet that was too old; it has to be resetted as soon as the first correct packet is received. `packetLostErrorCounter` counts the number of lost packets; it is resetted as soon as the first correct packet is received; `semTimeoutErrorCounter` counts the number of `EventSem` happens timeouts.

```
private:
   int32 vci;
   int32 moduleType;
   int32 packetByteSize;

   uint32* dataBuffer[nOfDataBuffers];
   int writeBuffer;
   bool isSynchronising;

   int32 maxDataAgeUsec;
```

```
   int32 maxNOfLostPackets;
   int32 syncroOnInputMsecTimeout;
   int32 lastPacketID;
   EventSem acquisitionSem;
   int liveness;

   void* outputPacket;

   int32 sizeMismatchErrorCounter;
   int32 previousPacketTooOldErrorCounter;
   int32 packetLostErrorCounter;
   int32 semTimeoutErrorCounter;
```

Now we switch to methods of the `ATMDrv` class. The `Init` method inits all module entries. `EnableAcquisition` and `DisableAcquisition` enables and disables board data acquisition. The method `RecCallback` is called from the ISR when the board generate interrupts for data transfer from the board to the main memory.

`InputDump` dumps input data to the passed by argument stream; `GetData` copies data from the device driver to the DDB. `OutputDump` dumps output data to the passed by argument stream; `WriteData` copies data from the DDB to the ATM board memory and command a send action to the hardware.

The method `IsSynchronizing` return the value of the attribute `isSynchronising`. `SetInputBoardInUse` set the board as an input module an that is in use. The same thing is done from the `SetOutputBoardInUse` setting the in use state on output.

`GetUsecTime` return the time of the last acquistion event, the last two methods, `GetCTS` and `GetRTTN` are old and plant (JET) specific and must be removed.

```
private:
   bool Init();

   bool EnableAcquisition();
   bool DisableAcquisition();

   void RecCallback(int* dataSource,int dataLength);
   bool InputDump(StreamInterface& s) const;

public:
   int32 GetData(uint32 usecTime,int32* buffer,int32 bufferNumber=0);

private:
   bool OutputDump(StreamInterface& s)const;

public:
   bool WriteData(uint32 usecTime,const int32* buffer);

private:
   ATMDrv(const ATMDrv& atm);
   ATMDrv& operator=(const ATMDrv& atm);
public:
   ATMDrv();
   ~ATMDrv();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectDescription(StreamInterface& s,bool full,StreamInterface* err);

   virtual bool IsSynchronizing();
   virtual bool SetInputBoardInUse(bool on = True);
   virtual bool SetOutputBoardInUse(bool on = True);

   int64 GetUsecTime();
   uint16 GetCTS();
```

```
uint32 GetRTTN();
```

9.2.2 DataPollingDrivenTTS GACQM (ATCAadcDrv)

As said in the previous subsection a `GenericAcqModule` has as an attribute, an array of `TriggerTimeService` each of that can hold an implementation of `TimeTriggeringServiceInterface`. In this subsection we analyse the case of a GACQM holding at least one `DataPollingDrivenTTS`. What happens on the `TimeInputGAM` side when executed by the `RealTimeThread`? We are assuming that the `GenericAcqModule::GetData` is not blocking, i.e. the module is not synchronising. In Figure 9.7 the UML sequence diagram show the method calling sequence.

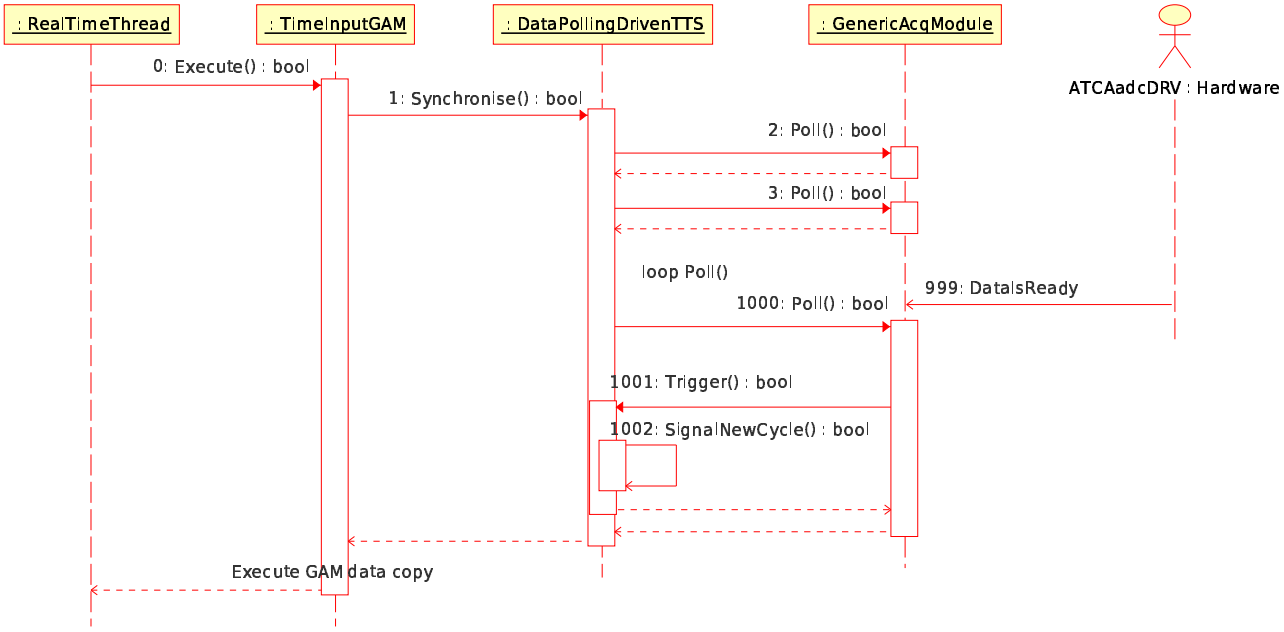

Figure 9.7: MARTe UML sequence diagram of a `DataPollingDrivenTTS` compatible `GenericAcqModule`

We try to illustrate what happens in the schema in Figure 9.7 step by step in the following:

0. the `RealTimeThread` calls `TimeInputGAM::Execute` method;
1. `TimeInputGAM::Execute` as first action calls `DataPollingDrivenTTS::Synchronise` method;
2. this method implements a loop in which in each iteration `GenericAcqModule::Poll` is called, if returns `False` the loop continue;

999. the device acquires some data and moves them to the main memory;

1000. we are still in the loop in the `DataPollingDriver::Synchronise` method, a new call to `Poll` happens;

1001. the method `GenericAcqModule::Poll` identifies that new data is just arrived then calls the `DataPollingDrivenTTS::Trigger` method;

1002. `DataPollingDrivenTTS::Trigger` as its last instruction calls `SignalNewCycle`;

1003. the method `DataPollingDrivenTTS::SignalNewCycle` modifies an internal variable of the `DataPollingDrive` class that lets the `Synchronise` method exit from the loop and returno control to the `TimeInputGAM`;

as the same time the `Poll` method return `True` letting the loop exit in any case. Then follow the data copy activity carried on by code implemented in the `InputGAM`.

Follows the usual class analysis of the implementing driver.

ATCAadcDrv

`[ATCAadcDrv.h, ATCAadcDrv.cpp]`
The class `ATCAadcDrv` is the high level driver for the ATCA ADC module, an hardward board developed from the JET plant; the class inherits exclusively from `GenericAcqModule`.

The first attribute `numberOfBoards` is the number of boards in the crate; readed during the initialisation; for each board the class has to create a new instance of the `SingleATCAModule` class, at the initialization an array of `SingleATVAModule`s is created and the pointer is holded by the `modules` attribute. `masterBoardIdx` is the index of the master board in the crate.

The attribute `lastCycleUsecTime` holds the last cycle time in μsec; `synchronizing` is `True` if the module is synchronizing, i.e. the `GetData` is blocking.

The attribute `softwareTrigger` configure the board for software triggered acquisition; if the attribute `autoSoftwareTrigger` is set to true then a software trigger will be sent every `autoSoftwareTriggerAfterUs` μsec. This mechanism works if the the value of `autoSoftwareTriggerAfterUs` in the configuration file is greater then 1.
Last attribute, `css` is the pointer to a string that holds the CSS file for the reflection HTTP page.

```
private:
   int32 numberOfBoards;
   SingleATCAModule* modules;
   int32 masterBoardIdx;

   int64 lastCycleUsecTime;
   bool synchronizing;

   int32 softwareTrigger;
   bool autoSoftwareTrigger;
   int32 autoSoftwareTriggerAfterUs;

   const char *css;
```

The method `GetData` copies data from the ATCA board into the DDB; `WriteData` moves data from the DDB to the ATCA board. Note that as the `ATMDrv` also this class can act as input and as output.

The method `PulseStart` is used for simulation purpose it commands a software trigger to the board. The method `ProcessHttpMessage` output a HTML page with the current value in mV of the acquired signals. `Poll` is the polling method.

```
   ATCAadcDrv();
   virtual ~ATCAadcDrv();

   virtual bool ObjectLoadSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ObjectDescription(StreamInterface& s,bool full=False,StreamInterface* err=NULL);

   int32 GetData(uint32 usecTime,int32* buffer,int32 bufferNum=0);
   bool WriteData(uint32 usecTime,const int32* buffer);
   virtual bool Poll();
```

```
virtual bool SetInputBoardInUse(bool on=False);
virtual bool SetOutputBoardInUse(bool on=False);

virtual bool EnableAcquisition();
virtual bool DisableAcquisition();

int64 GetUsecTime();

bool PulseStart();

virtual bool ProcessHttpMessage(HttpStream &hStream);
```

9.2.3 Other GACQMs

Other GACQMs not in Figure 9.5 are depicted in the UML class diagrams of Figure 9.8 and Figure 9.9. We first speak about the IOGAMs in the first Figure cited; such class are:

- `HRTDrv`
- `VxWorksDrv`
- `TimerDriver`
- `LinuxTimerDriver`

All classes, but not `HRTDrv`, can be used associated with a `InterruptDrivenTTS` class. `HRTDrv` is an incomplete developed class, infact can be redeveloped to fullfill to what a `DataPollingDrivenTTS` class need, i.e. the `Poll` method. Infact the `HRTDrv` has only a synchronising `GetData` method that in polling read the HRT counter of the machine we are executing on; `HRTDrv` works on every operating system. Other classes are OS specific.

class	TTSI	run on
`HRTDrv`	none	any
`VxWorksDrv`	`InterruptDrivenTTS`	VxWorks
`TimerDriver`	`InterruptDrivenTTS`	Win32, Linux glibc
`LinuxTimerDriver`	`InterruptDrivenTTS`	Linux

Table 9.1: MARTe `GACQM`s (timing)

Classes OS specific are all `InterruptDrivenTTS` and can be used to generate periodic timings and cycles.

We switch now to analyse Figure 9.9, classes involved in this gruop follow as a list. Classes are of two types: *synchronizing* and `streaming`.

- `SynchronizingDriver`
- `StreamingDriverReceiver`
- `StreamingDriver`

Until the end of this section we just spend some words abuot the `StreamingDriver` class; such class implement a `GenericAcqModule` that basically consits in a thread that run in a loop synchronised by sempahores with the `RealTimeThread`. Each time the `RealTimeThread` writes something to the buffers of the `StremingDriver` those GACQM send them asinchronusly via an UDP/IP socket, in this way we garantee that the `RealTimeThread` never blocks its execution especially if the `StramingDriver`'s thread runs on a different cpu's core or processor.

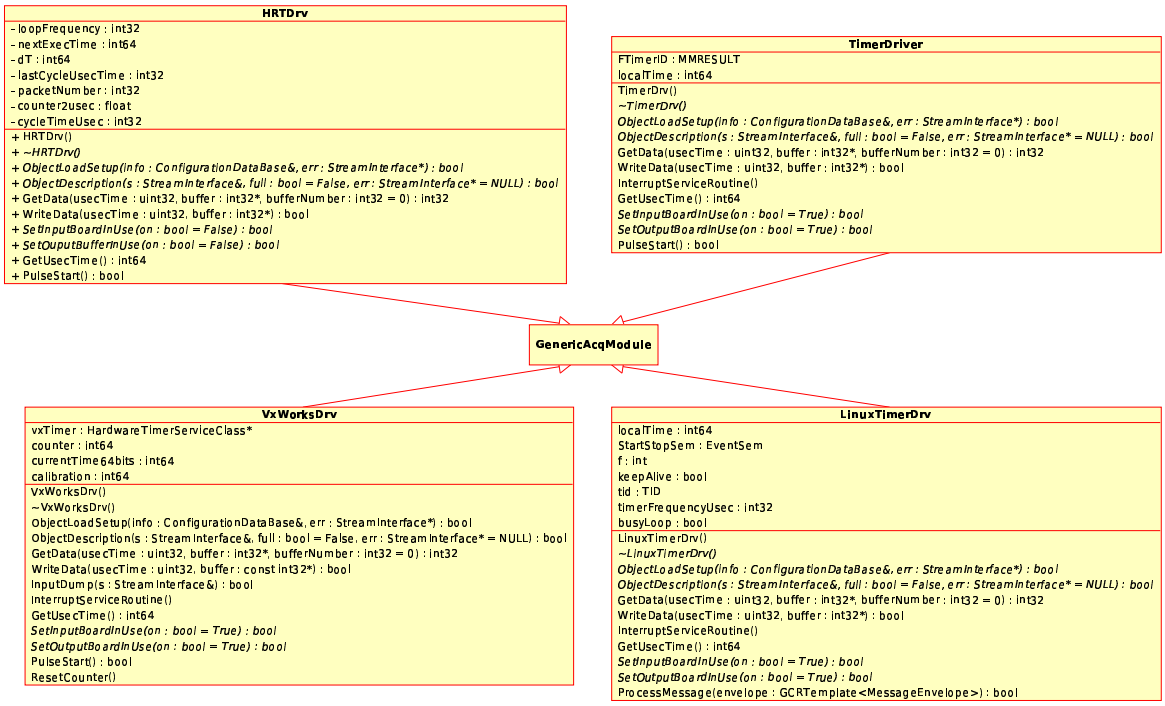

Figure 9.8: MARTe GACQMs (timing)

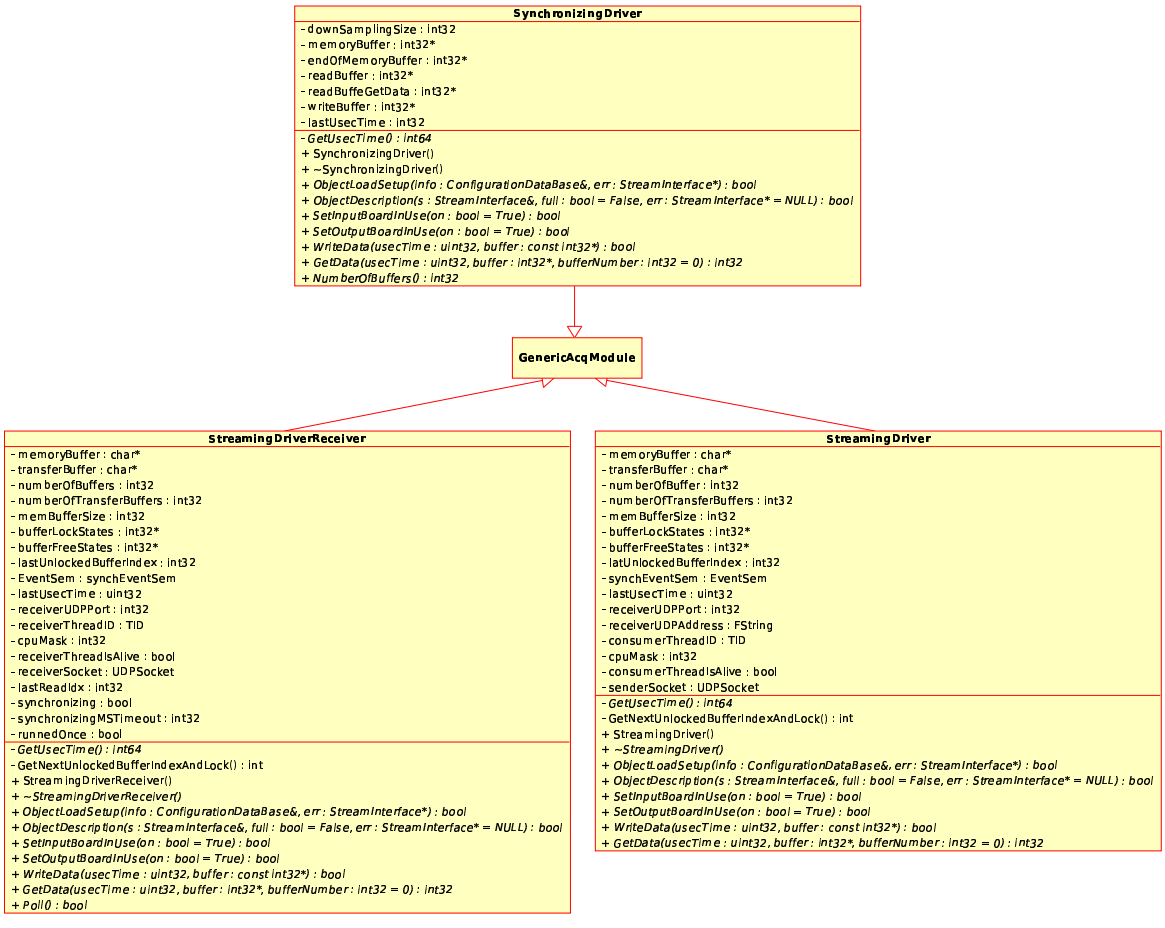

Figure 9.9: MARTe GACQMs (synchronizing and streaming)

9.2.4 Design Notes

The `ATMDrv` and other drivers plant specific has some plant specific methods like `GetCTS` and `GetRTTN` those methods must be moved to other levels where streamed data from the ATM is parsed.

Regarding the `ATMDrv` synchronising facility on the `GetData` method: this choice require to be very carefull when deciding to use it or not. Good choices are:

- ATMDrv **NOT** synchronizing and a `TimeInputGAM`;
- ATMDrv synchronizing and a `InputGAM`.

All other choice make the system unusable.

About the unique polling GACQM founded, the `ATCAadcDrv` class, the doubled polling design is not required. With the "double polling design" we want to focus the attention to the fact that when the `GenericAcqModule::Poll` method return `True` the `DataPollingDrivenTTS::SignalNewCycle` method has just turned into `True` the variable that holds the loop and the `Synchronise` method can return.

Some directory cleanup is also required, for example the `StreamingDriver` source code is in two different directories.

9.3 Generic Application Modules (GAMs)

GAMs are described in the chapter about *BaseLib/Level5* and are the main component to interact with a DDB and the `RealTimeThread`; moreover a GAM implement an activity, i.e. code to execute, to make calculation, run algorithm, statistic or saving datas.

Classes in this section comes from *MARTe/GAMs* directory, any GAM has an its own directory, all classes are depicted in the UML class diagram of Figure 9.10, note that in the subsection that follows we try to divide the classes in different groups to easy the analysis.

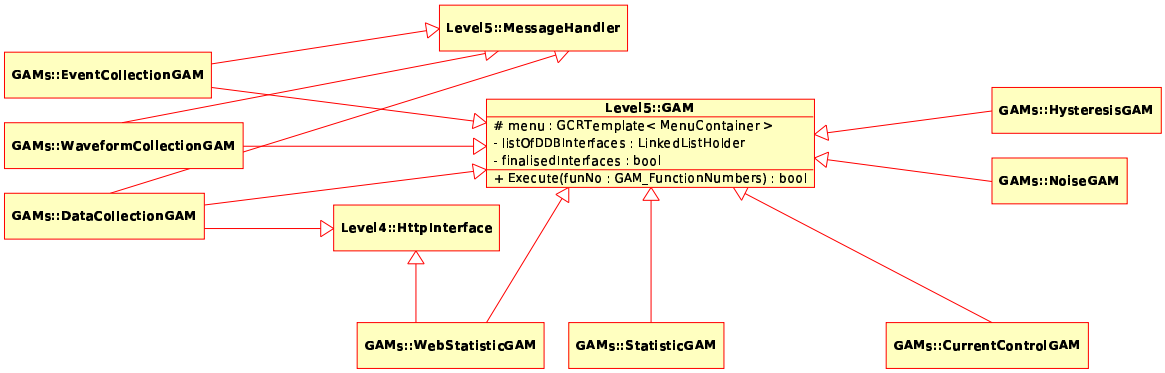

Figure 9.10: MARTe `GAMs`

9.3.1 Data Collection GAMs

This short section introduces GAMs for the data collection, those GAMs just read from the DDB, i.e. from a block diagram point of view such newly introduced GAMs are output blocks (sinks of data). GAMs just read data samples and each time a new data sample is received is compared with the previous one and is maybe marked for data collection. Note that if we receive the same sample for 10.000 times is not necessary to store it 10.000 times but is just necessary to take the first value, as well as its time stamp, and then store the first different sample coming (obviously with time stamp). The

three data collection GAMs presented here just differ in the way they mark samples for data collection. Those GAMs are:

- `EventCollectionGAM`
- `WaveFormCollectionGAM`
- `DataCollectionGAM`

All GAMs are written for the data flow at JET (are born quite plant specific) and variables some times are just called *jpf* something.. JPF is an acronym and means "JET Pulse File" and its a raw data format for each shot. There are many other kind of data files like PPF an CPF but raw data is tored in JPF. To use the data collection GAMs in another system just think as *jpf* variables as involved with raw data from the instrumentation.

All GAMs, as you can see in the UML class diagram of Figure 9.11, use the same subsystem to save acquired data from the DDB to main memory. The memory saving activity is coordinated by the `RTDataCollector` class. All other classes just help this class. Class involved are:

- `RTDataCollector`
- `DataCollectionSignalsTable`
- `DataCollectionSignal`
- `RTDataPool`
- `RTDelaySystem`
- `RTDataStorageSystem`
- `RTCollectionBuffer`

Such data collection mechanism must handle triggered data windows, i.e. imagine we are waiting for a trigger and when it happens we need 1000 samples before and after, so we need to collect the 1000 previous samples also if they are not marked for data storage.

The data collection mechanism saves all data to main memory and at the end of the pulse via messaging from the supervisor state machine sends all saved data to the JPF collection system of the JET plant. This mechanism is not developed for continuous data acquisition, if you need continuous data acqusition you must use a sort of `StreamingDriver`.

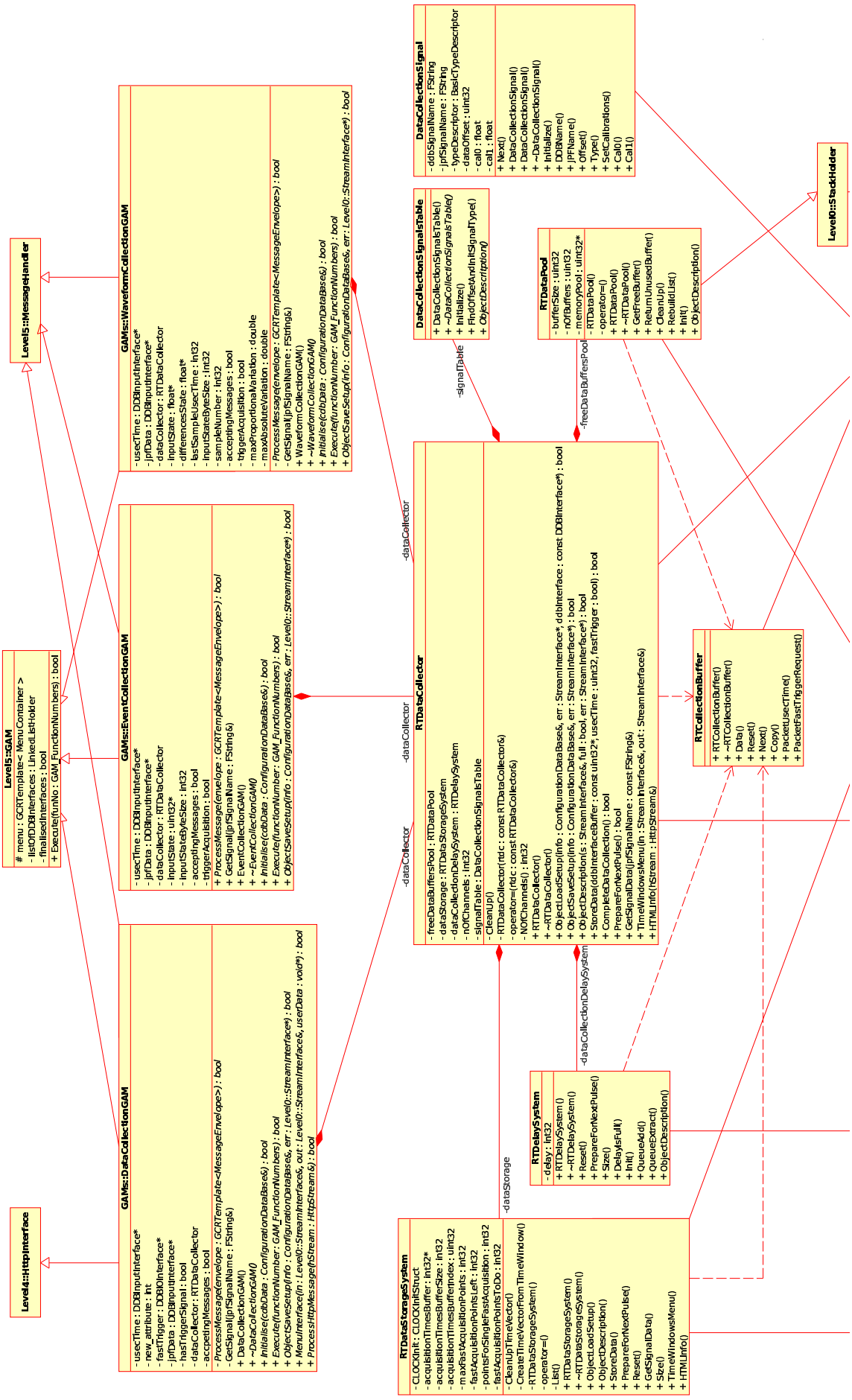

Figure 9.11: MARTe data collection GAMs

EventCollectionGAM

[EventCollectionGAM.h, EventCollectionGAM.cpp]
The EventCollectionGAM is a data collection GAM that holds a private copy of the current state vector (channels values); on data arrival it compares the current state vector with the previously saved, if at least one value differ the code mark the vector for data storage. The EventCollectionGAM inherits from GAM and implements MessageHandler like the other GAMs in this section.

Such class its conceptually an output GAM (not an IOGAM because it hasn't got a matching device driver), i.e. it only read data from the DDB but doesn't write data to it. It has two DDBInputInterface one for reading the time, usecTime, and the other to read the signal, jpfData. Like the other data collection GAMs it is an high level interface between the RTDataCollector infrastructure and the GAM's world.

The core of this class is, like any other GAM, the Execute method and exactly the following lines of code:

```
for(int i=1; (i < (inputStateByteSize/sizeof(int32))) && (!triggerAcquisition); i++)
   if(jpfDataBuffer[i] != inputState[i]) triggerAcquisition=True;
switch(functionNumber){
   case GAMOnline:{
      if(!dataCollector.StoreData(jpfDataBuffer, usecTimeSample, triggerAcquisition)){
// ...
```

This code compares each signal of the current signals vector (jpfDataBuffer) with the previously acquired vector (inputState) if at least one signal differ it marks it for storage (setting triggerAcquisition to True) then calls StoreData. Note that the code store the current acquired vector, i.e. jpfDataBuffer.

```
private:
   DDBInputInterface* usecTime;
   DDBInputInterface* jpfData;
   RTDataCollector dataCollector;

   uint32* inputState;
   int32 inputStateByteSize;

   bool acceptingMessages;
   bool triggerAcquisition;

   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
   GCRTemplate<SignalInterface> GetSignal(const FString& jpfSignalName);

public:
   EventCollectionGAM();
   virtual ~EventCollectionGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

WaveformCollectionGAM

[WaveformCollectionGAM.h, WaveformCollectionGAM.cpp]
The WaveformCollectionGAM class is quite similar to the previously described EventCollectionGAM, inheriting GAM and implementing MessageHandler. Like the previous class also this one holds a copy of the previous data vector (inputState); WaveformCollectionGAM holds also the last sample time (lastSampleUsecTime) and the last differences vector, i.e. a vector of differences between the current data and the previous data. Using those saved values and the two factors maxProportionalVariation

and `maxAbsoluteVariation` decide if the current set of samples has to be marked for storage or not. The check return `True` if the following formula is verified.

$$|\triangle[k] - \triangle[k-1]| > maxProportionalVariation * |\triangle[k-1]| + maxAbsoluteVariation$$

The $\triangle[k]$ is computed as the difference between the current signals value and the previous saved one. The multiplicative and additive factors are constants. The code that implements this check follows.

```
for(int i = 1; i < (inputStateByteSize/sizeof(int32)); i++){
   float currentDifference = jpfDataBuffer[i] - inputState[i];

   if(fabs(currentDifference - differencesState[i]) >
      (maxProportionalVariation * fabs(differencesState[i]) + maxAbsoluteVariation))
      triggerAcquisition = True;
   differencesState[i] = currentDifference;
}
```

Also such class its conceptually an output GAM (not an IOGAM because it hasn't got a matching device driver), i.e. it only read data from the DDB but doesn't write data to it. It has two `DDBInputInterface` one for reading the time, `usecTime`, and the other to read the signal, `jpfData`. The interface follows.

```
private:
   DDBInputInterface* usecTime;
   DDBInputInterface* jpfData;
   RTDataCollector dataCollector;

   float* inputState;
   float* differencesState;
   double maxProportionalVariation;
   double maxAbsoluteVariation;

   int32 lastSampleUsecTime;
   int32 inputStateByteSize;
   int32 sampleNumber;

   bool acceptingMessages;
   bool triggerAcquisition;

   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
   GCRTemplate<SignalInterface> GetSignal(const FString& jpfSignalName);

public:
   WaveformCollectionGAM();
   virtual ~WaveformCollectionGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

DataCollectionGAM

`[DataCollectionGAM.h, DataCollectionGAM.cpp]`
Finally comes the most different Data Collection GAM. `DataCollectionGAM` is the only one that implements also `HttpInterface`. It has not only the two common `DDBInputInterface` but also a further `DDBIOInterface`, it means that this class not only reads from the DDB but also writes; this third interface to the DDB is called `fastTrigger` and it is used infact to decide if a vector of values are

to be marked for data storage or not. Infact in the `Execute` method to decide if the current vector of values has to be marked for storage a read on the `DDBIOInterface` is performed and if it is `True` then the data is marked for storage. After the data has been collected the readed value, used to decide, is resetted in the DDB.

```
private:
   DDBIOInterface* fastTrigger;
   DDBInputInterface* usecTime;
   DDBInputInterface* jpfData;

   bool hasTriggerSignal;
   RTDataCollector dataCollector;

   bool acceptingMessages;

   virtual bool ProcessMessage(GCRTemplate<MessageEnvelope> envelope);
   GCRTemplate<SignalInterface> GetSignal(const FString& jpfSignalName);

public:
   DataCollectionGAM();
   virtual ~DataCollectionGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);

   virtual bool MenuInterface(StreamInterface& in,StreamInterface& out,void* userData);
   virtual bool ProcessHttpMessage(HttpStream& hStream);
```

9.3.2 Statistic GAMs

This section introduced two GAMs that product statistical data on the signal in input. Such statistical data is not outputted from the GAMs but, only in the case of the `WebStatisticGAM` can be readed in real time using a web browser via the `HttpInterface`. GAMs don't write on the DDB the statistical calcolated data, but at initialization for each signal a group of statistical signals is added to the `DDBOutputInterface` of the classes. The problem is that after that the DDB's output interface is never used, probably is still work in progress.

Another notes is about this two classes, they are identical, except that the `WebStatisticGAM` has more variables and more statistic and also is the only one used right now, in the following we just analyse such GAM.

Figure 9.12 depict the UML class diagram of the classes involved in this section that are:

- `StatisticGAM`, `StatSignalInfo`
- `WebStatisticGAM`, `StatSignalInfo`

Statistic GAM

`[StatisticGAM.h, StatisticGAM.cpp]`

DEPRECATED

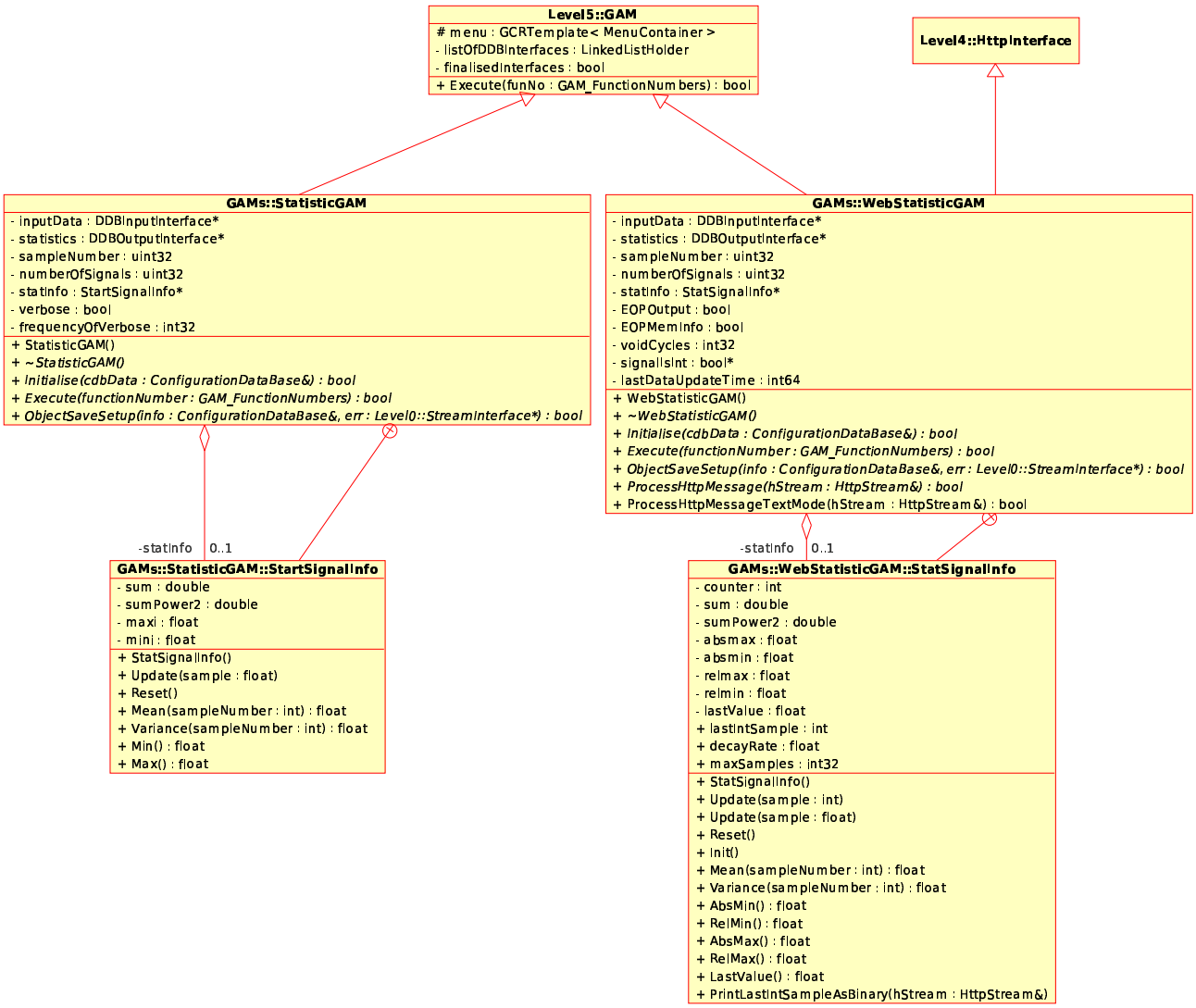

Figure 9.12: MARTe statistic GAMs

Web Statistic GAM

[WebStatisticGAM.h, WebStatisticGAM.cpp]
The WebStatisticGAM mantains statistical data for each signal in the DDBInputInterface (attribute inputData). For each signal it updates its statistical data in an entry of the StatSignalInfo array (statInfo). We first have a look to the StatSignalInfo class.

The first attribute of this private class is counter, i.e. an internal counter used to reset the reset the relative statistics after reached maxSamples. The attribute sum is the sum of values of a signal, it decays with a rate decayRate. sumPower2 is the sum of squared of values of a signal, it also decays with a rate decayRate.

The attribute absmax is the absolute maximum of a signal; absmin is the absolute minimum of a signal. relmax is the relative maximum of a signal, it is resetted every maxSamples; relmin is the relative minimum of a signal, it is resetted every maxSamples. lastValue is the last summed value of a signal.

The attribute lastIntSample is the last value of an integer sample; decayRate is the rate at which past samples are "forgotten".

The first `Update` method updates the `lastIntSample` value; the other `Update` method updates statistics with the value passed by, is to be called during the *online* and *offline* phase. `Reset` resets the relative statistics; `Init` initialises the class, to be called in the *prepulse* phase.

The method `Mean` returns the mean of the signal, `Variance` returns the variance of the signal, `AbsMin` returns `absmin` attribute, `AbsMax` returns `absmax` attribute, `RelMin` returns the relative minimum value of the signal, `RelMax` returns the relative maximum value of the signal. `LastValue` gets `lastValue` attribute. Finally `PrintLastIntSampleAsBinary` prints the last acquired integer value in binary format on the `HttpStream`.

The class's interface follows.

```
class StatSignalInfo{
private:
   int counter;
   double sum;
   double sumPower2;
   float absmax;
   float absmin;
   float relmax;
   float relmin;
   float lastValue;

public:
   int lastIntSample;
   float decayRate;
   int32 maxSamples;

   StatSignalInfo();
   void Update(int sample);
   void Update(float sample);
   void Reset();
   void Init();

   float Mean(int sampleNumber);
   float Variance(int sampleNumber);

   float AbsMin();
   float RelMin();
   float AbsMax();
   float RelMax();
   float LastValue();

   void PrintLastIntSampleAsBinary(HttpStream& hStream);
};
```

We now switch to speak about the `WebStatisticGAM`, that how we said calculates statistical informations about a signal, it inherits from `GAM` and implements `HttpInterface`. Such statistical component only shows the calculated values via an HTTP browser, the `DDBOutputInterface` is not working right now. The only working interface to the DDB is the `DDBInputInterface` that lets the `WebStatisticGAM` collect information to make the statistical measures.

The attribute `sampleNumber` is the current sample number; `numberOfSignals` is the number of signal for which the GAM is keeping statistical data; `statInfo` is a pointer to the first `StatSignalInfo` class.

The attribute `EOPOutput` must be `True` if output on logger is desired at the end of the pulse. `EOPMemInfo` must be `True` if output of memory information is desired at the end of the pulse. The attribute `voidCycles` is the number of cycles before starting to keep statistical information, used to discard the first wrong samples.

As you can remember information on the DDB can be kept in integer or in floating point format, `signalIsInt` is an array of booleans, one for each signal, an enrty is `True` if the relative signal is an `int32`, `False` if it is a `float`. `lastDataUpdateTime` holds the timestamp of last update of the statistics.

Like in other GAMs statistics are updated in the `Execute` method.

```
private:
   DDBInputInterface* inputData;
   DDBOutputInterface* statistics;

   uint32 sampleNumber;
   uint32 numberOfSignals;

   StatSignalInfo* statInfo;

   bool EOPOutput;
   bool EOPMemInfo;

   int32 voidCycles;
   bool* signalIsInt;
   int64 lastDataUpdateTime;

public:
   WebStatisticGAM();
   virtual ~WebStatisticGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);

   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
   virtual bool ProcessHttpMessage(HttpStream& hStream);
   bool ProcessHttpMessageTextMode(HttpStream& hStream);
```

9.3.3 Other GAMs

There are other GAMs left that are contained in the *MARTe/GAMs* directory, those classes are depicted in the UML class diagram of Figure 9.13 and are listed below:

- `NoiseGAM`
- `HysteresisGAM`
- `CurrentControlGAM`

All those classes are input and output GAMs if seen from the DDB point of view, i.e. they read data on the DDB (via a `DDBInputInterface`) run an algorithm on the data and write back the new values on the DDB (via a `DDBOutputInterface`). Those action are performed in the `Execute` method of the GAMs. By the way every one of the following GAM has the same `Execute` method, an example code follows.

```
1  bool NoiseGAM::Execute(GAM_FunctionNumbers functionNumber) {
2     float* outputData = (float*)output->Buffer();
3     memset((void*)outputData, 0, sizeof(float)*numberOfSignals);
4     switch(functionNumber){
5        case GAMOnline:{
6           input->Read();
7           int32 usecTime = input->Buffer()[0];
8           float* inputData = &(((float*)input->Buffer())[1]);
9           int i = 0;
10          for (i=0; i<numberOfSignals; i++)
11             outputData[i] = AddNoise(inputData[i]);
12       }
13    }
14    output->Write();
15    return True;
16 }
```

The code shows that the first action to perform entering the `Execute` method is to call `input->Read()` then it is possible to read the incoming data, make some calculations on it and then rewrite it in the output buffer (`outputData`), when done you must call `output->Write()`; just remember, `input` is of type `DDBInputInterface` and `ouput` is of type `DDBOutputInterface`.

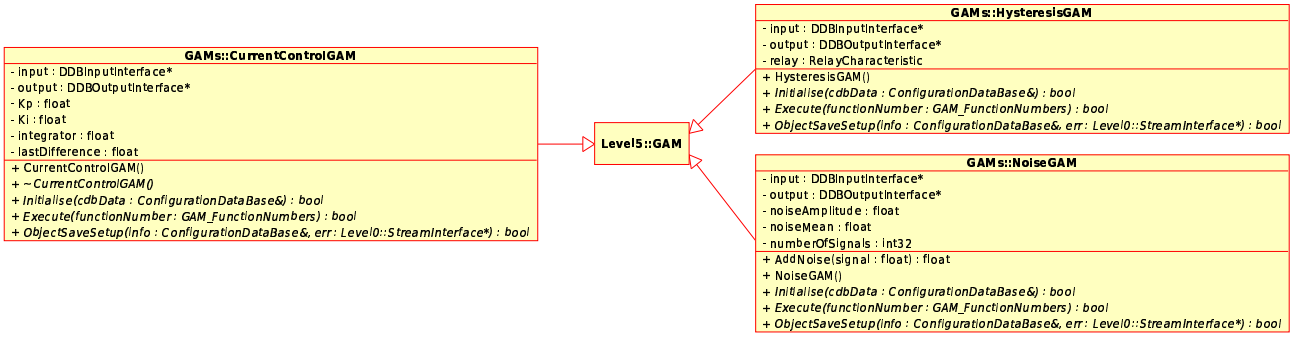

Figure 9.13: MARTe other `GAM`s

NoiseGAM

`[NoiseGAM.h, NoiseGAM.cpp]`
The `NoiseGAM` class is the first DDB's input/output GAM we find. Such GAM adds to each signal readed from the DDB a random value calculated using attributes `noiseAmplitude` and `noiseMean` then writes back the computed signal in the DDB `output` buffer.

In the following code you can how the value of the new signal is calcolated having a look to the inline function `AddNoise`. The attribute `numberOfSignal` returns the number of signals are readed and written by the GAM.

```
private:
   DDBInputInterface* input;
   DDBOutputInterface* output;

   float noiseAmplitude;
   float noiseMean;

   int32 numberOfSignals;

   inline float AddNoise(float signal) {
      return (signal+(rand()*noiseAmplitude)-noiseMean);
   }
public:
   NoiseGAM();
   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

HysteresisGAM

`[HysteresisGAM.h, HysteresisGAM.cpp, RelayCharacteristic.h]`
The `HysteresisGAM` is another DDB's input/output GAM. Such GAM acts on `one` channel at a time only and output a value that follow an hysteresis curve defined by initialization parameters. The hysteresis calculation and the parameters describing it are done by a `RelayCharacteristic` class, this class was done to drive the JET's FRFA Amplifier. Such class must be resetted between each pulse.

```
class RelayCharacteristic {
private:
```

```
   float hysteresisState;
   float hysteresisForcingVoltage;
   float hysteresisActivationThreshold;

   Waveform hysteresisUpperThresholdWaveform;
   Waveform hysteresisLowerThresholdWaveform;

   float maximumAmplifierControlVoltage;
// ...
```

The attribute `hysteresisState` is the value of FRFA control voltage at the previous step, `hysteresisForcingVo` is the value of current to add to the direction of FRFA control to force activation; `hysteresisActivationThreshold` is the activation threshold of the FRFA (2500V); the attribute `hysteresisUpperThresholdWaveform` is the waveform for the hysteresis upper threshold and `hysteresisLowerThresholdWaveform` is the waveform for the hysteresis lower threshold. `maximumAmplifierControlVoltage` is the maximum value for FRFA control voltage.

This class is plant specific (JET) but can be easily utilized in other systems. The code interface of the `HysteresisGAM` follows.

```
private:
   DDBInputInterface* input;
   DDBOutputInterface* output;

   RelayCharacteristic relay;

public:
   HysteresisGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

CurrentControlGAM

`[CurrentControlGAM.h, CurrentControlGAM.cpp]`
The `CurrentControlGAM` is a PI (Proportional Integrative) filter. It only works for **one** channel at a time. Like the previous two GAMs is a DDB's input/output GAM. The GAMs reads data on the `DDBInputInterface` (`input`) and writes on the `DDBOutputInterface` (`output`).

The attribute `Kp` is the proportional gain, `Ki` is the integral gain, `integrator` is the integrative sum and `lastDifference` is the last sample difference between reference and current.

```
private:
   DDBInputInterface* input;
   DDBOutputInterface* output;

   float Kp;
   float Ki;
   float integrator;

   float lastDifference;

public:
   CurrentControlGAM();
   virtual ~CurrentControlGAM();

   virtual bool Initialise(ConfigurationDataBase& cdbData);
   virtual bool Execute(GAM_FunctionNumbers functionNumber);
   virtual bool ObjectSaveSetup(ConfigurationDataBase& info,StreamInterface* err);
```

The PI algorithm is implemented in the `Execute` method of the GAM; the source code of the algorithm, that follows, expects that the sample value in the incoming data buffer as the format defined by this structure, i.e. the GAM doesn't work connected in the DDB to a generic array of signals:

```
typedef struct _InputData{
    int32 usecTime;
    float currentReference;
    float currentMeasured;
}InputData;
```

Then have a look at the source of the algorithm, the steps are:

1. Read input (formatted) data from the DDB, calculating the difference between the reference and the currently measured value (lines 1,2);
2. reading the the current sampling time from the DDB, same `DDBInputInterace` of the data (line 3);
3. calculating `sampleLengthUsec` and saving the current sampling time in the `lastSampleTime` static variable(?) (lines 4-9);
4. charging the integrator (line 11);
5. updating `lastDifference` attribute (line 12);
6. write the computed value to DDB (line 13).

```
1     float diff = ((InputData*)input->Buffer())->currentReference -
2        (InputData*)input->Buffer())->currentMeasured;
3     int32 currentSampleTime = ((InputData*)input->Buffer())->usecTime;
4     int32 sampleLengthUsec;
5     if (lastSampleTime == 0)
6        sampleLengthUsec = 0;
7     else
8        sampleLengthUsec = currentSampleTime - lastSampleTime;
9     lastSampleTime = currentSampleTime;
10
11    integrator += (lastDifference + diff) * (sampleLengthUsec * 1E-6) / 2;
12    lastDifference = diff;
13    *(output->Buffer()) = (Kp*diff)+(Ki*integrator);
```

9.3.4 Design Notes

Data collection GAMs are just quite similar to what a `StreamingDriver` is; i.e. both are data sinks in a data flow schema. The main difference between them is how they export data: a `DataCollectionGAM` saves in Real Time all values in memory and at the end of the pulse sends a big data packet to CODAS systems; the `StreamingDriver` is a GACQM and work behind an IOGAM asynchrounosly sending data using another (a parallel) thread. A `StreamingDriver` coupled with an IOGAM allow continuos data recording. The real problem is philosophical: the `StreamingDriver` doesn't relay on an hardware device, only on the socket interface, why is a `GACQM`? Because it holds a thread? (this could be a good parameter of differentiating GAMs from GACQM, i.e. a GACQM is every piece of code that relay on a device driver or on multi threaded activity, when the `Execute` method is not enough).

In the statical GAMs there are some further activity to do for the cleanup and update:

- delete `StatisticGAM` or derive `WebStatisticGAM` from it, the code as it is right now has no sense; deriving the `WebStatisticGAM` adding the `HttpInterface` is a good solution;

- fully implement the `DDBOutputInterface` to provide further reuse of statistics;
- optimize the computations with SIMD instructions.

Those activity are to be scheduled in next development for the framework.

In Figure 9.14 there is the Component Object Model diagram (Microsoft style) of the `CurrentControlGAM`. Such class holds as internal class its `DDBInputInterface` and `DDBOutputInterface` complying with the DDB specifications, those interface are inner thanks to the `GAM` definition. To mantain a real component based framework interfaces must be better defined and understanded, this means that for example that the `DDBInputInterface` will be an interface not an abstract class like it is right now.

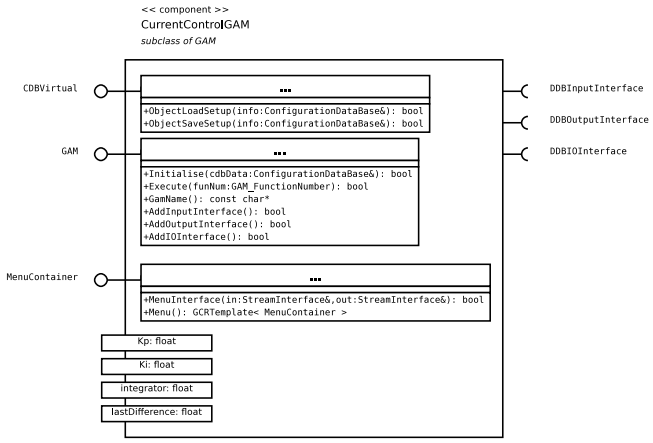

Figure 9.14: MARTe `CurrentControlGAM`, Component Object Model (Microsoft style) schema.

Last notes about DDB buffers. The format of buffers must be more clear. Mostly of the time a GAM expect an `DDBInputInterface` where the buffer has as the first argument a time stamp of 32bit and then an array of values; the `CurrentControlGAM` just differs exepcting a buffer with the following format.

```
typedef struct _InputData{
    int32 usecTime;
    float currentReference;
    float currentMeasured;
}InputData;
```

9.4 Remarks

Just to summarize the content of this chapter we propose Figure 9.15 that shows togheter all GAMs presented here and groups it by input/output capability; Figure 9.16 shows and group GACQM by its input and output capability and highlight the `TimeInput` characterization.

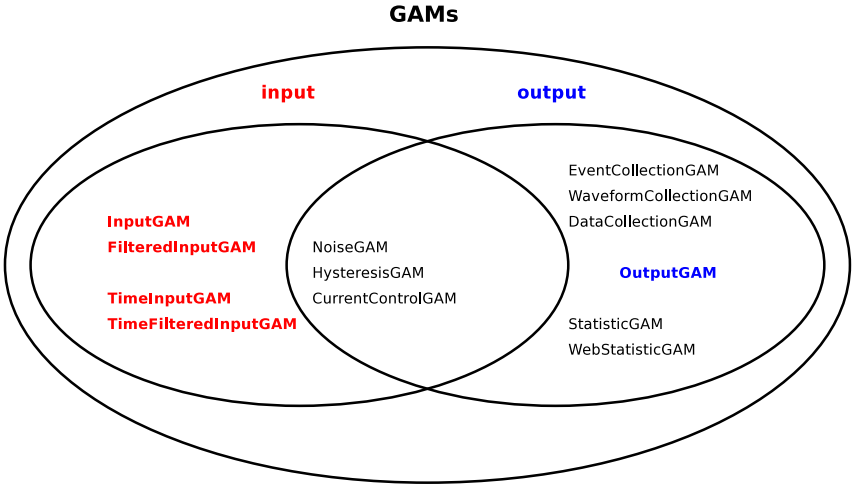

Figure 9.15: MARTe GAMs grouped by I/O capability (source/sink of data) concerning the block diagram.

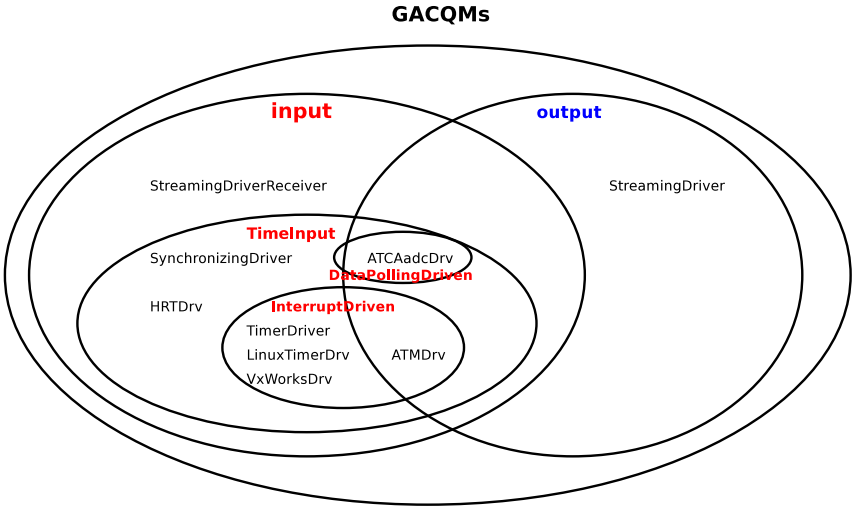

Figure 9.16: MARTe GACQMs grouped by I/O capability (source/sink of data) concerning the environment.

Part IV

Design Analysis

Chapter 10

Comments

TODO

- HTTP to XML redesing (HTTP Iterface must be redesigned for generalization)
- configuration database must continue support to cfg files but must be go to xml (supports different configuration file types)
- streaming to MDSplus
- is a good choice to adopt RTTi instead of the GCFilippo support?
- group all stuff toghetr with a choerent directory hierical (no levels) with stronger architectures and os support
- When XML is done, enable graphical configuration
- Switching to exception catching? (try...catch)
- code inlining e del codice scritto negli .h ok le performance ma quanto codice ce? quanto codice duplicate?
- cleanup the code and file hierical redesign (MARTe, MARTe/MarteSupportLib)

Direi che l'architettura è davvero eccezionale. Or ora sto dedicando del tempo per: -eventuali altri confronti con altri prodotti commerciali/open source; -approfondimento di altri linguaggi di programmazione a msg, vedi Smalltalk e ObjC; in questo modo posso scrivere vantaggi e svantaggi della BaseLib rispetto a questi.

Dal mio punto di vista sarebbe molto interessante poter fare un refeactoring della BaseLib per poterla esporre anche come prodotto open source. Vorrei riorganizzarla: - definendo con più chiarezza le interfaccie (molte sono classi astratte), in modo da SOTTOLINEARE quello che l'aspetto component del sistema; - far discendere tutte le classi da Object togliendo eventuali loop di inheritance (che ora come ora non ci sono perchè tu non derivi tutto da Object, ma potrebbero nascere), in questo modo, assumendo che Object sappia mandare e ricevere messaggi, tutti gli oggetti possono mandare e ricevere messaggi (come in ObjC) perchè sono figli di Object, il codice non sarebbe neanche duplicato. - riorganizzazione delle directory dei sorgenti.

-se gli oggetti avessero anche una completa reflection combinando un db online ed uno offline potresti conoscere tutti i metodi di ogni oggetto e quindi avere una shell tipo VxWorks di debug non solo in kernel