libagents-2.0

Actor Model C++ Library

Reference Manual

(c) Information Technology Group
www.itgroup.ro

(c) Virgil Mager

doc_id 2.0.2 / 2020-04-23

About this document

This document is the reference manual of the libagents library.

This document is organized such that it *gradually* introduces the terminology used in each chapter, all while trying to *minimize any forward references* to yet-undefined terms and concepts; it is thus ***strongly recommended to read the information presented in this document progressively, in the order it is presented***, from the beginning to the end of this document.

Document versions

The libagents documentation uses a three-number document ID 'x.y.r', *which is identical to the version number of the library it documents*.

Target audience

The reader of this document is assumed familiar with the C++ programming language and the STL. The libagents library is implemented in ANSI C++11; however, the libagents API is compatible with C++98, such that understanding and using the libagents library features does not require knowledge of the C++11-specific (or later) language features.

Additionally, the reader of his document is assumed to have (at least) basic knowledge about object-oriented, event-driven programming, and about event-driven frameworks and APIs.

Document format

This document has been authored with the OpenOffice 3.4.1 suite and saved in OpenDocument 1.2 format (a.k.a. ODF 1.2). The ODF 1.2 HybridPDF version of this document can be read with any modern PDF reader (PDF 1.4+), and it can be edited natively with the LibreOffice 4+ office suite.

Editing this document

If you intend to make changes to this document in view of republishing, please make sure you use the latest version of this document, listed in the "Download" chapter, as the basis for your edit; additionally, please save a copy of your modified version in Open Document Text format (.odt) and send it to itgroup.ro@gmail.com, together with the list of changes that you have made.

License

About the libagents library

The libagents library provides a C++ implementation of the "Actor Model" paradigm, thus enabling the development of pure C++ multi-threaded applications structured as ***a collection of asynchronous event-driven execution agents which all run concurrently and communicate with each other via an asynchronous messaging system***. The libagents v2.0.x library implements a reliable message delivery protocol between agents, which provides a failed-delivery notification mechanism for messages that cannot be delivered to their target recipients.

- *Note:* a message will fail to be delivered to its target recipient *if and only if* said target recipient's input message buffer is full, in which case [multiple] send retries must be performed by the sender for messages that must unconditionally reach their destination

The asynchronous, event-driven, agents-based data processing paradigm as implemented by the libagents library requires approaching the program implementation problem from an *agent-oriented perspective.* At one end of the spectrum an application can consist of any number of agents that exchange messages with each other and execute specific subroutines *exclusively as the result of receiving an incoming message*, while at the other end of the spectrum an application can consist of a single agent which executes a standard sequential program flow from start to finish, without implementing any event-driven functionality.

The concept of "agent" is fundamental to, and is implemented as a base class "Agent" in, the libagents library. Each libagents user application has to create its own types of agents (tailored to the functionality required by the application) by deriving custom agent classes from the libagents' "Agent" base class.

- ✗ for example, let us consider an application that must implement a one-second Timer which generates a "tick" message every one second, and a programmable Divider which takes the Timer ticks as input messages and generates an output message every N Timer ticks: in this case the application's internal architecture can consist of a Timer agent and a Divider agent, both implemented as custom agents derived from the libagents Agent base class, with the Timer agent's "tick" output messages being directed to [the input of] the Divider agent
- ✗ alternatively, let us consider an application which takes some user input from the standard console, processes each new input as it receives it, and prints the end result when it detects a special "finish" token in the input data: in this case we'd just need to create a single agent which waits for, and then reads, the successive user inputs, processes each input one at a time, and when it detects the "finish" token it prints out the result and exits, i.e. the program will consist of a single agent which implements a simple wrapper object around a standard sequential program which successively reads and processes each user input

License

How to use the libagents library

The libagents v2.0.x library is provided exclusively in source code format, with the library source files organized inside a single top-level folder 'libagents-2.0.x'. The libagents v2.0.x library is distributed as a compressed archive 'libagents-2.0.x.zip', and it must be be decompressed on the host file system in a location whose path contains ***exclusively alpha-numeric characters, '_' (underline), '-' (minus), and '.' (dot)***.

Because the libagents v2.0.x library is provided exclusively in source code format, building a libagents-based application requires making both the application-specific source files, *and the libagents library source files*, part of the application project, and then *all these files have to be compiled and linked together* when building the libagents-based application.

- *Note:* the process of compiling and linking an application is host system-dependent (whether static or dynamic linking is used), such that its specifics are not covered in this document

The libagents configuration file

The libagents library includes a *"library configuration file"* 'libagents-config.h' which contains various object declarations, inline functions, constants (declared as enums), and #defines, which are used internally in the libagents library source code, and/or which can be used in a libagents-based user application; additionally, the libagents library has several configuration options which are defined in the library configuration file 'libagents-config.h', and which can be changed according to the requirements of each specific application and/or host operating system:

- OS_TICK: ***this constant must be set to the application's host OS tick, in milliseconds.*** The default value is 1ms, which is correct for Win32/64, WinRT, Linux, FreeBSD, OS X, Android, and iOS
- MESSAGE_BUFFER_SIZE: this constant defines the *default* size of the message buffers used for all message exchanges; the default value provided in the library configuration file should be adequate for most applications but it can be explicitly specified in the user application (details later in this document), and ***it should only be changed towards a higher value*** if thus required by the application

Creating a libagents application project

As previously described at the beginning of this chapter "How to use the libagents library", in order to build a libagents application, *both the application-specific source files and the libagents library source files* must be present on the host system where the application is to be built; then, the libagents application project must be set up to contain the following files:

- the application-specific source files (i.e. the application's {.h+.cpp} source files)

```
<application-specific source files> {.h+.cpp}
```

- the libagents library source files:

```
libagents-2.0.x/libagents-config.h
libagents-2.0.x/libagents.h
libagents-2.0.x/libagents.cpp
```

Additionally, the following compiler settings must be used when building a libagents-based project:

- the compiler must [be configured to] support ***C++11***
- ***the compiler's #include path must be set to contain the full path to the 'libagents-2.0.x' library root folder*** (e.g. 'c:\libagents-2.0.x' if the libagents library root folder resides directly on a Windows system's C: drive)

Example applications

A 'stopwatch' example application is available in the libagents library package. This is a simple GUI application which uses the libagents library for the core logic, and the Qt 5.9.5 framework for its GUI interface. The application can be built by first installing the Qt 5.9.5 framework on the user's host system, then the application project file 'libagents-examples\libagents-example-stopwatch\libagents-example-stopwatch.pro' must be opened in Qt Creator IDE, and finally the project must be built by invoking Qt Creator's “Build project” command from Qt Creator's main menu.

- *Note 1:* the line “CONFIG += c++11” in the 'libagents-example-stopwatch.pro' project file configures Qt Creator's qmake build system to enable the C++11 features
- *Note 2:* the stopwatch application structure is detailed in the application's “README.TXT” file; this file can be found inside the stopwatch application folder, and it is also listed in the “other files” folder in Qt Creator's project navigator
- *Note 3:* the stopwatch application has been built and tested with the Qt 5.9.5 framework SDK on Xubuntu 18.04 LTS 64-bit edition and Windows 7 32-bit edition. The “libagents-example-stopwatch.pro” project file is compatible with Qt Creator 4.5.2 IDE, which is part of the Qt 5.9.5 SDK

The architecture of a libagents application

The top-level architecture of a libagents-based application consists of ***two autonomous modules which communicate with each other***, and which together "sit on top of" the application's underlying ***"host framework"***; this is illustrated in Fig.1 below:

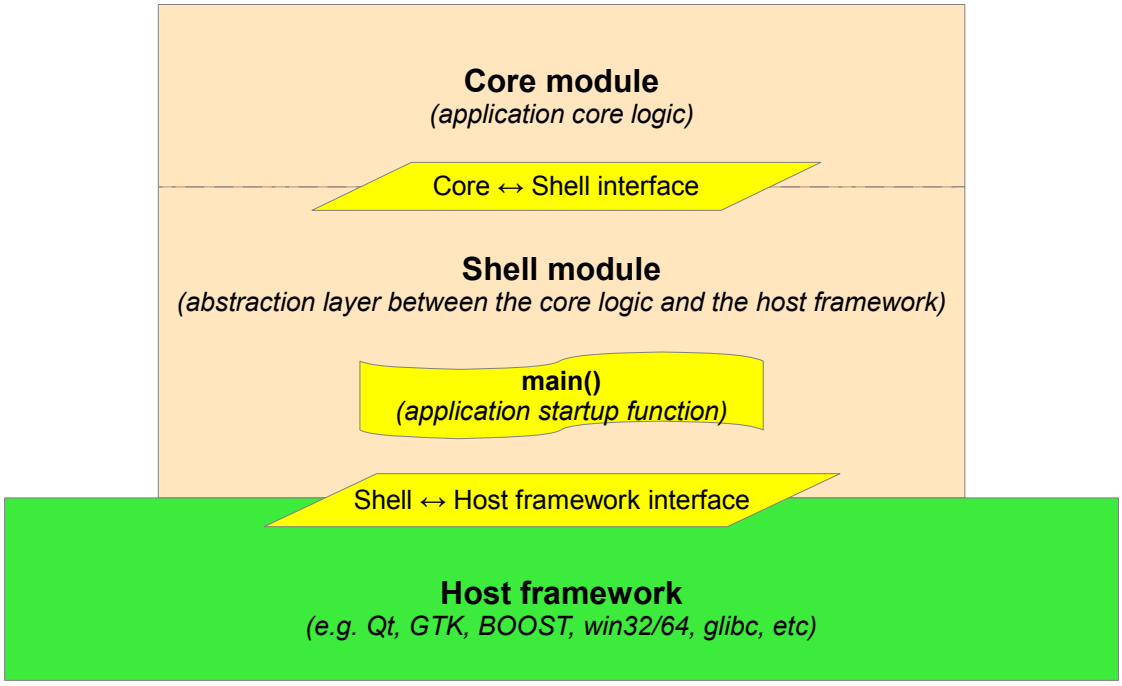

Fig.1: top-level architecture of a libagents-based application

- ***the application "Core module"*** contains the implementation of the entire core logic of the application, and it should be implemented by using exclusively the standard C++ language features, the features offered by the libagents library, and possibly features provided by the STL and/or other *platform-independent* standard libraries
  - IMPORTANT: ***the libagents library mandates a specific internal architecture for the Core module of a libagents-based application***; the architectural constraints imposed on the Core module of a libagents application are discussed in "The Core module" chapter below
- ***the application "Shell module"*** contains the *application startup function* illustrated as "main()" in Fig.1 above (i.e. the function which is automatically executed when the application is started, e.g. "int main(int argc, char **argv)" for a console application, or "WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)" for a win32 application, etc), and its role is to provide the Core module with a *platform-independent interface* to any and all of the environment-integration functionalities that the Core module requires during application execution (e.g. networking functions, access to the host file system, etc)
  - IMPORTNAT: the libagents library does not mandate any specific architecture for the Shell module of a libagents-based application; instead, ***the only architectural constraint that the libagents library imposes on the Shell module architecture is that the Shell module contains the application's startup function***
- ***the application's "host framework"*** is the foundation upon which *the Shell module* of a libagents application is built, i.e. the application's host framework provides any and all of the *low-level, platform-specific functions* "upon which" the functionalities of the Shell module are implemented

- *Note*: throughout this document we will use the term “host framework” for any *API and/or framework* “on top of which” the Shell module is built (e.g. Qt, GTK, wxWidgets, win32/64, glibc, etc, will all be designated as “host frameworks”)

The following paragraphs in this chapter detail the internal structure and the functionality of the libagents Core and Shell modules.

The Core module

As previously explained in “The architecture of a libagents application” paragraph above, the Core module is responsible for implementing the application's core logic. In terms of internal architecture, ***the Core module of a libagents application consists of a single top-level “Core” object***, which, in turn, contains a hierarchy of “Task”, “Thread”, and “Agent” objects; this is illustrated in Fig.2 below:

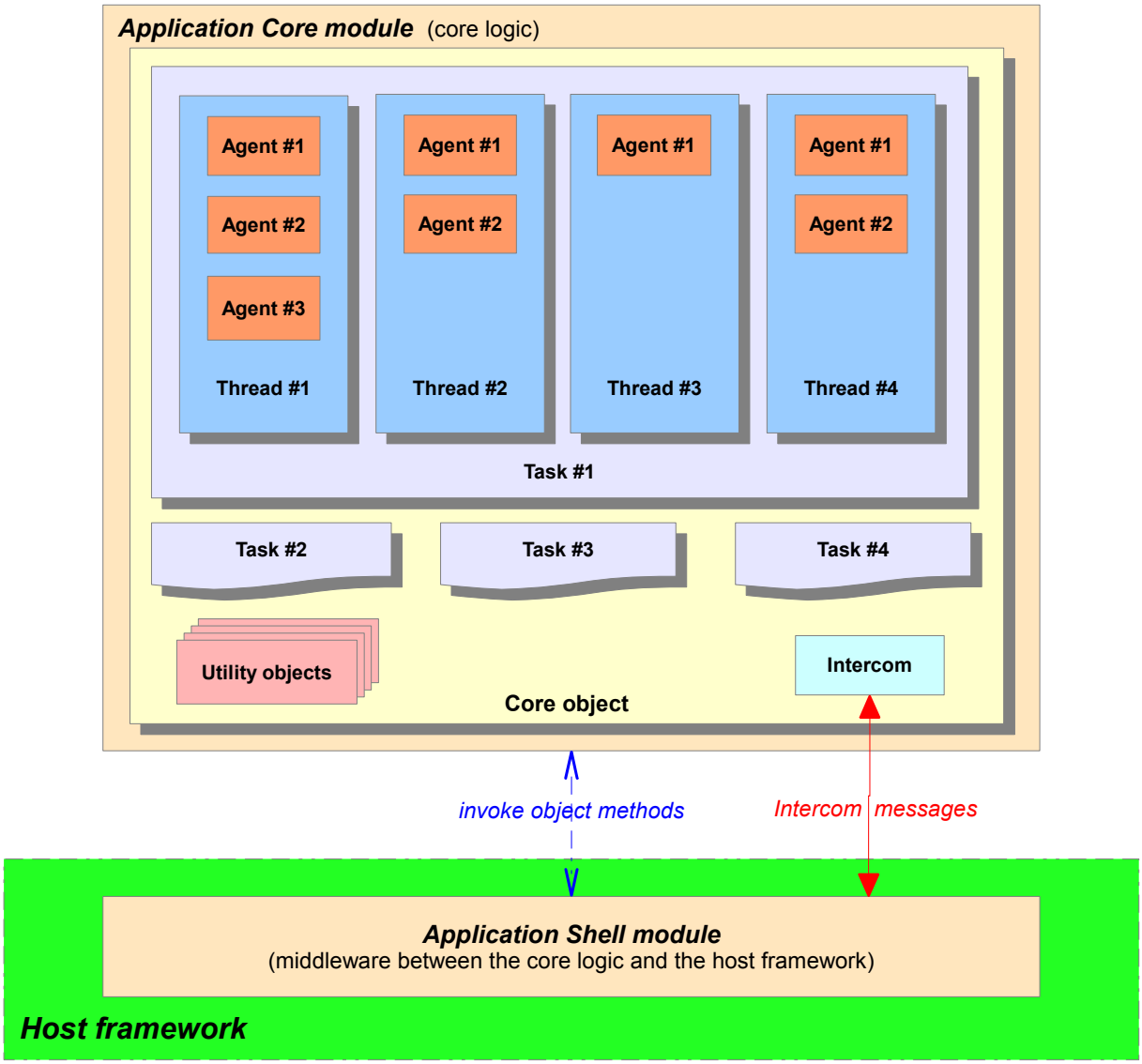

Fig.2: the mandatory top-level architecture of a libagents application Core module

- the ***top-level 'Core' object*** is a singleton instance of a user-defined class derived from the 'Core' base class provided by the libagents library
  - the Core application object contains one or more *“**Tasks**”*, which are instances of user-defined classes derived from the 'Task' base class provided by the libagents library
    - each task executes one or more parallel ***“Threads”***, which are instances of user-

defined classes derived from a 'Thread' base class provided by the libagents library

- each thread hosts one or more ***"Agents"***, which are instances of user-defined classes derived from the 'Agent' base class provided by the libagents library

- the Core application object *may* contain a collection of ***optional "Utility objects"***, whose role is to provide various functions that are commonly used by the application *but which are not related to the integration of the application with its operating environment* (e.g. encryption functions, mathematical functions, etc)
- the Core application object contains an ***"Intercom port"***, which is an object that enables the Core module to exchange ***"I/O messages"*** with the Shell module

The role and implementation of the constituent components of a libagents application's Core module as illustrated in Fig.2 above are detailed in the following paragraphs.

The Agent objects

A libagents "Agent" object is *the elementary data processing units* of a libagents application, and it consists of [an instance of] a user-defined class derived from the libagents "Agent" base class.

The entire functionality of an agent object is implemented as a single "onMessage()" method which is the exclusive data processing function of an Agent object, and whose execution is automatically triggered [by libagents's internal mechanisms] each time a new message is received by an agent. The "onMessage()" method is a pure virtual method declared inside libagents's "Agent" base class, and it must be implemented by each user-defined agent class.

In other words, a [user-defined] agent object is a data processing unit that sits idle awaiting for an incoming message, and for each message it receives it executes its "onMessage()" method which contains specific code branches/algorithms associated with the received message and possibly with the value of a [scalar or aggregate] state variable at the time when the message was received (in this latter case the agent implements a state machine); this is illustrated in Fig.3 below:

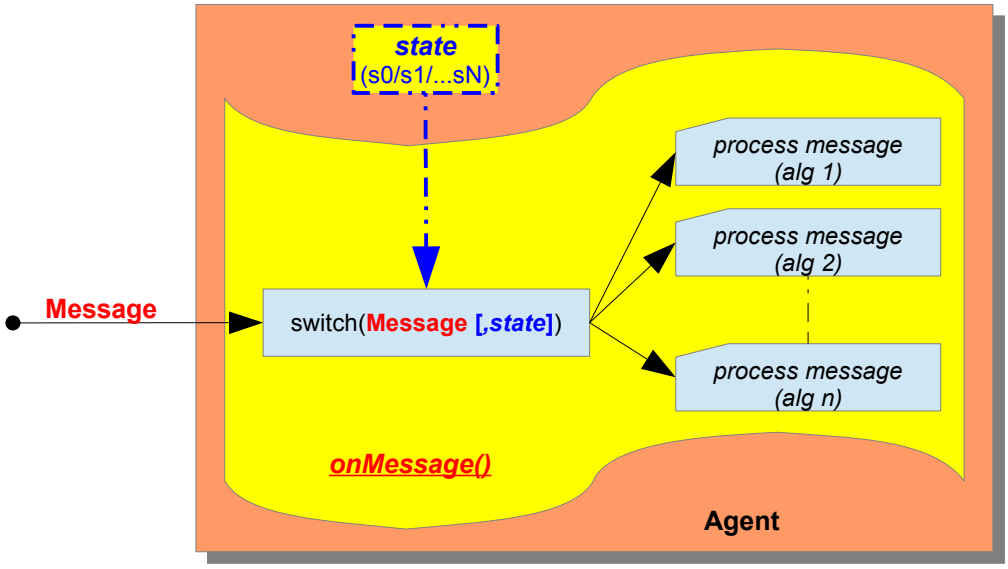

Fig.3: every data processing sequence performed by an agent is triggered by an incoming message and it is performed exclusively by the agents "onMessage()" method

Apart from the "onMessage()" method, the libagents "Agent" base class also provides methods for sending a targeted message to another agent which is part of the same Task, for broadcasting a message to all, or part of, the other agents in the same Task, and for sending messages to the application's Shell module (the details on the libagents messaging system are presented in "The messaging system" paragraph later in this chapter).

The Thread objects

A libagents “Thread” object consists of [an instance of] a user-defined class derived from libagents's “Thread” base class, and it provides methods for creating and destroying agents “inside” a Thread object (these methods are detailed in “The libagents API” chapter later in this document).

A libagents “Thread” object is the elementary execution-scheduling unit of a libagents application, and it logically groups together agents whose message processing algorithms (namely, their “onMessage()” methods) are executed by the same OS thread within a [potentially multi-threaded] application; we will here-forth refer to the OS thread that executes the “onMessage()” method of the Agent objects that are part of a given Thread object as “the underlying OS thread” associated with the Thread object.

The following scheduling rules apply:

1. for any two agents that belong to two different Thread objects, their “onMessage()” methods are always executed concurrently and independently by the two OS threads that correspond to the two Thread objects that contain the two agents
2. for any two agents that belong to the same Thread object, their “onMessage()” methods are executed atomically, in succession, by the underlying OS thread of their Thread object container. In other words, once an agent in a given Thread object receives a message and starts executing its “onMessage()” method, all the other agents that belong to the same Thread object will not be receiving, nor will they be processing, any messages until the currently-running agent's “onMessage()” method completes execution

The scheduling mechanism described above is illustrated in Fig.4 below:

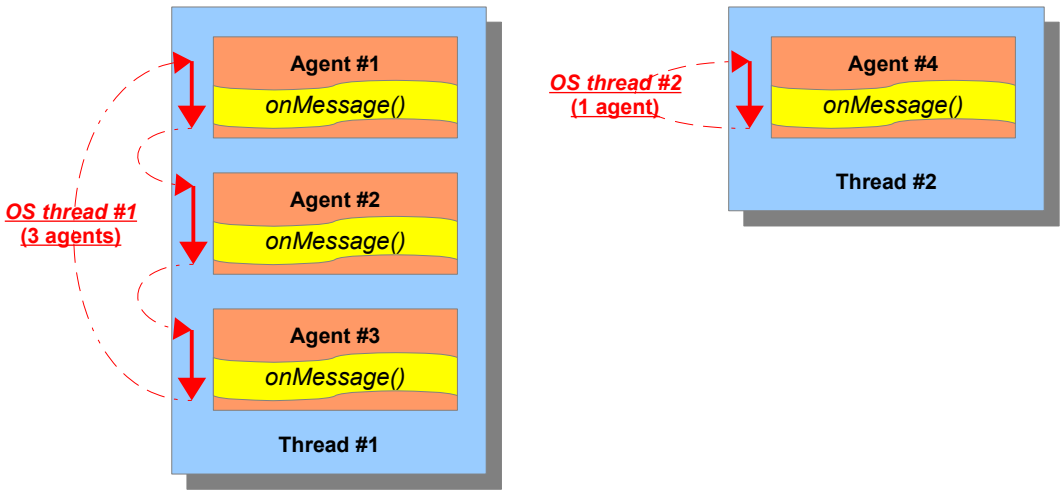

Fig.4: each Thread object is associated with a distinct OS thread,
and the “onMessage()” methods of agents belonging to the same Thread object
are executed atomically, in a round-robin scheme, by said Thread object's underlying OS thread

- IMPORTANT: the round-robin scheduling scheme for agents that belong to the same Thread object (see Fig.4 above) has the important consequence that [the execution of] an agent's “onMessage()” method blocks all the other agents in its Thread object until the method completes execution; thus, whenever the execution time of a given agent's “onMessage()” method can be prohibitively long, and/or can adversely impact other “sibling” agents that run in the same OS thread with itself, said agent should be implemented as a stand-alone component of a dedicated Thread object, such that it can run concurrently with, while having no blocking impact over, any other agents in the application; this situation is exemplified by Agent#4 in Thread#2 in Fig.4 above

The Task objects

A libagents “Task” object logically groups together one or more “Thread” objects, and it consists of [an instance of] a user-defined class derived from libagent's “Task” base class.

- *Note:* a Task object can also be though of as [indirectly] grouping together all the agents that are part of all its contained Thread objects

The functional role of a “Task” object is to allow organizing the agents in a libagents application based on how they can exchange messages with each other: namely, *all agents that are part of the same Task can exchange direct messages with one-another, while agents that are part of different Tasks can only exchange messages via a special relaying procedure* (more details on said relaying procedure are presented in the “Inter-task communication” paragraph later in this chapter).

The messaging restrictions presented above are illustrated in Fig.5 below, where the red lines represent direct messages that can be exchanged between agents that are part of the same Task object, and the blue lines represent inter-Task messages that can be exchanged between agents only by using the above-mentioned special relaying procedure:

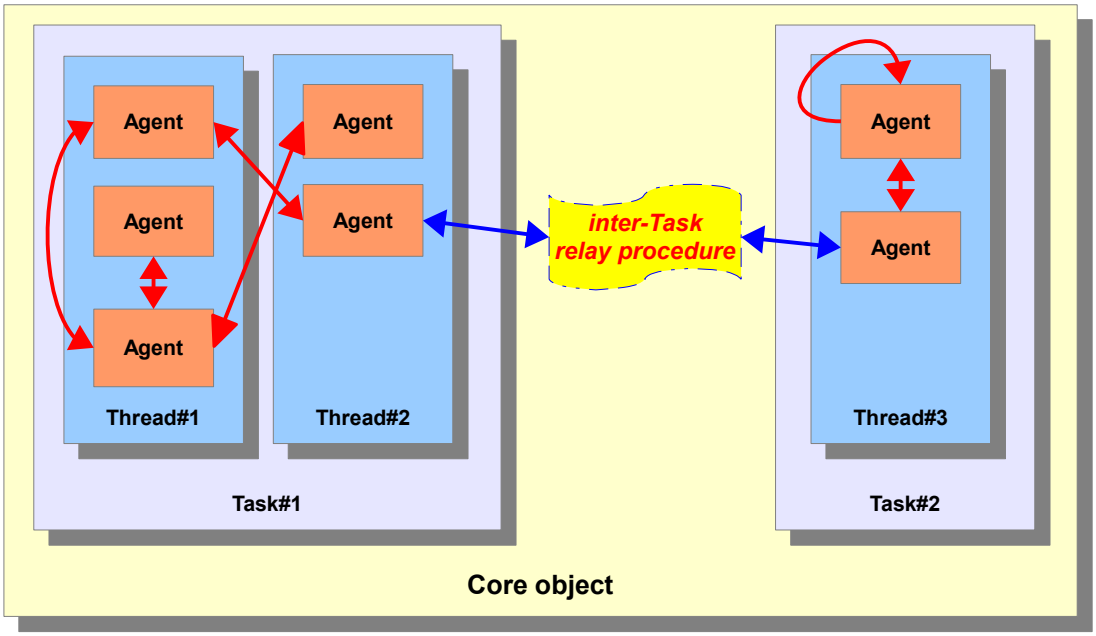

Fig.5: direct messages can only be exchanged between agents in the same Task object, while inter-Task messaging must use a special relaying procedure

The libagents “Task” base class provides methods for creating and destroying “Thread” objects (these methods are detailed in “The libagents API” chapter later in this document), as well as methods that allow an agent to be subscribed/unsubscribed to/from messages that are broadcasted by another agent (broadcasted messages are presented in the “Broadcasted messages” paragraph later in this chapter).

The Core object

The “Core” object of a libagents application logically groups together all the application's “Task” objects (and, indirectly, all the application's “Thread” and “Agent” objects), and it consists of a *singleton* instance of a user-defined class derived from the libagents “Core” base class.

The libagents “Core” base class provides methods for creating and destroying “Task” objects (these methods are detailed in “The libagents API” chapter later in this document), for facilitating the communication among the individual agents in an application, and for exchanging I/O messages between the Core module and the application Shell module (more details on I/O messages are provided in the “I/O messages” paragraph later in this chapter).

The Utility objects

The “Utility objects” are *optional* application-specific, user-defined objects which may, or may not, be implemented by an application (i.e. the libagents library does not provide a base class for the Utility objects), and whose role is to provide various functionalities that are commonly used by the application but which are not related to the integration of the application with its operating environment (e.g. encryption functions, advanced mathematical functions, image processing functions, etc). The Utility objects should be instantiated by (i.e. “contained in”) the top-level Core object, thus making them accessible both to the Core object and to all the other objects that are contained in the Core object (i.e. the application's Agent, Thread, and Task objects).

- IMPORTANT: ***any Utility object whose methods and/or internal data may be accessed from multiple OS threads must implement a proper multi-threading protection***; this is discussed in the “Multi-threading in the Core module” paragraph later in this document
- *Note:* the Utility objects contained in the Core object can also be accessible from the Shell module; this is discussed in “The Shell Controller object” paragraph later in this document

Object local data

As is was previously explained throughout this document, the Agent, Thread, Task, and Core objects of a libagents application's Core module are created by deriving a custom class from a libagents base class, and then creating instances of said derived classes as required by the application (e.g. an Agent object myAgent of type MyAgent is created by first declaring the MyAgent class via derivation from the libagents “Agent” base class, and then instantiating myAgent = new MyAgent, etc); however, none of the libagents Agent, Task, Thread, and Core base classes provide any inbuilt user-accessible data storage elements, such that *any user-defined type of object derived from said libagents base classes must implement its own local data structures if needed, tailored for keeping the state information and other associated data as required by said user-defined type of object* (the objects' local data, implemented as object properties, can then contain scalar data, embedded or dynamically allocated data structures, arrays, etc).

An example of a libagents application Core module which implements local data in conjunction with each of its Agent, Task, Thread, and Core component objects is illustrated in Fig.6 below:

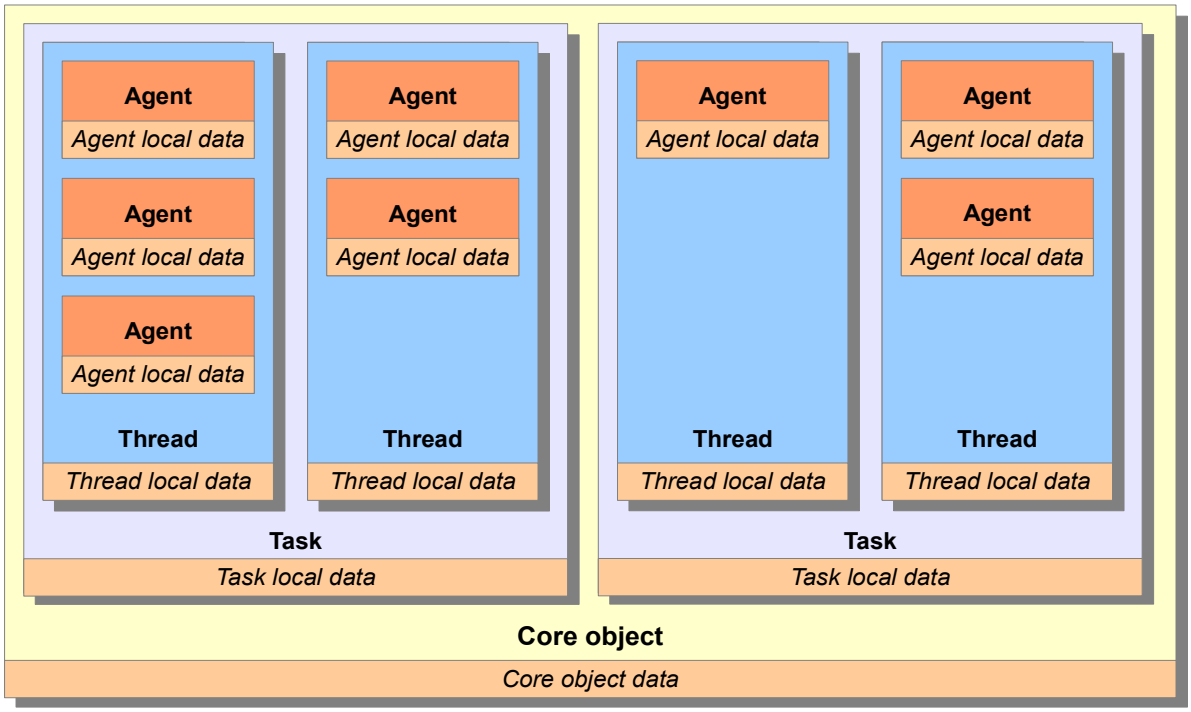

Fig.6: example of a libagents application where each component includes a local data store

- IMPORTANT: ***any object local data which may be accessed from multiple OS threads must be properly protected against multi-threading data races***; this is discussed in the "Multi-threading in the Core module" paragraph later in this document

- *Note:* the libagents library does not impose any restrictions as to which, or how many, of the application's Core module objects can, or should, implement their own local data, nor on whether an object's local data should be implemented as private or public properties

Initializing the Core module

As previously explained in "The Core module" paragraph earlier in this chapter, the Core module of a libagents application consists of a single top-level "Core" object, which, in turn, contains a hierarchy of "Task", "Thread", and "Agent" objects; in this context, ***the process of initializing the Core module consists exclusively of creating, and then "starting", the top-level Core object***, and it will be the Core object's responsibility to further create, and then "start", its sub-component objects.

Fig.7 below illustrates the succession of events that follow the invocation of the Core object's constructor (step A in Fig.7 below) and its "Start()" method (step B in Fig.7 below):

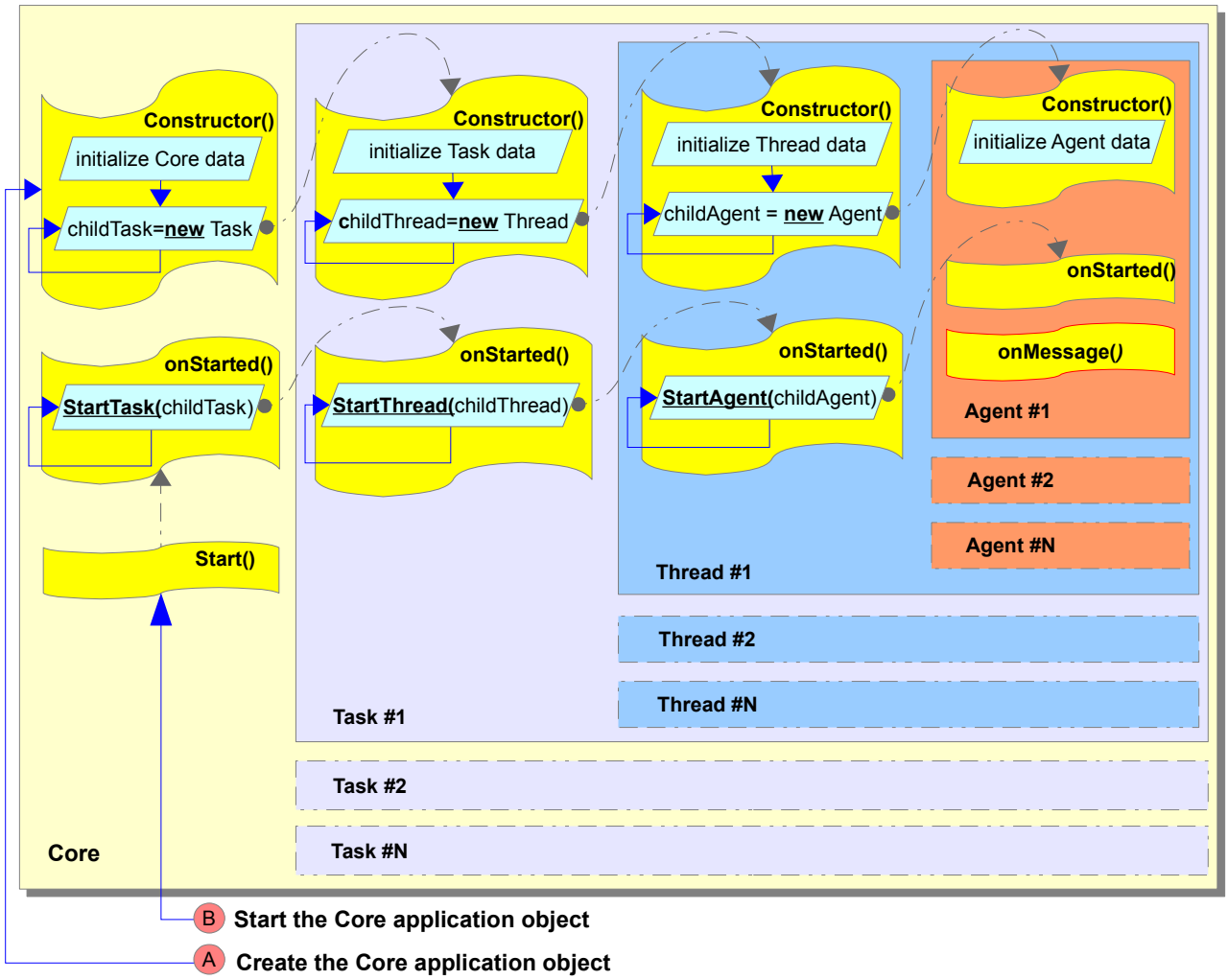

Fig.7: initialization of a libagents application's Core object

The following notations are used in Fig.7 above:

- the yellow ribbons inside the Core, Task, Thread, and Agent objects represent methods of the corresponding objects

- the cyan parallelograms inside the object methods represent blocks of user-defined code which must implement the functionality annotated inside each parallelogram

- the blue line arrows represent the program flow between two blocks of user-defined code, or the explicit invocation of a method from a block of user-defined code
- the dotted gray line arrows represent ***automatic invocations of methods***, i.e. the methods that these arrows point to ***must never be explicitly invoked***

The following paragraphs detail each of the methods illustrated in Fig.7 above.

The object constructors

As it can be seen in Fig.7 above, the functional role of the Core, Task, Thread, and Agent object constructors is to initialize their corresponding *local* data structures (if any), and *create the sub-component objects* (if any) of each particular container object.

- IMPORTANT: each sub-component object of a given container object must be created by the container object's constructor ***strictly via the "new" operator***, e.g. creating a new Task object "myCounterTask" of type "MyCounterTask" must be implemented by the Core object's constructor as "MyCounterTask *myCounterTask = new MyCounterTask", etc

The "onStarted()" methods

The "onStarted()" methods illustrated in Fig.7 above are all *pure virtual* methods provided by the libagents "Core", "Task", "Thread", and "Agent" base classes, and they must be implemented by each Core, Task, Thread, and Agent object used in a libagents application. The mandatory functional role of each object's "onStarted()" method is to *start all the sub-component objects (if any)* of each container object, e.g. the "onStarted()" method of the Core object must start the application's Task objects, the "onStarted()" method of each Task object must start said Task object's sub-component Thread objects, etc (see Fig.7 above); specifically:

- each sub-component object of a given container object must be started by its container object via said container object's Start*XXX*() method; for this purpose, the Core base class provides a "StartTask()" method, the Task base class provides a "StartThread()" method, and the Thread base class provides a "StartAgent()" method
- the Core object is the only [type of] object which provides its own "Start()" method, and starting the Core object is accomplished by invoking the Core object's own "Start()" method. The Core object's "Start()" method is provided by the libagents "Core" base class
  - *Note:* the fact that the Core object is the only object which provides its own "Start()" method directly results from the Core object being the top-level object in the Core module hierarchy, such that there is no higher-level object in the Core module hierarchy which could provide a "StartCore()" method for starting the Core object
- IMPORTANT: ***all the start methods mentioned above (i.e. "Start()", "StartTask()", "StartThread()", and "StartAgent()") automatically invoke, as the last step of their execution, the "onStarted()" method of the object that they start*** (e.g. the Core object's "Start()" method automatically invokes the Core object's own "onStarted()" method, the Core object's "StartTask()" method automatically invokes the "onStarted()" method of the Task object that is starts, etc), such that the user-defined blocks of code which start a given object (by invoking said object's "StartXXX()" method, or the Core object's own "Start()" method) must ***never*** explicitly invoke the "onStarted()" method of the objects that they start

Initialization thread

For most libagents-based applications, the initialization of the application's Core object (and its sub-component objects) can, and should, consist *exactly* of the orderly execution of the two steps (A)

and (B) illustrated in Fig.7, and *it should be performed by the application's startup function* (i.e. the function illustrated as "main()" in Fig.1) as part of the application's startup procedure; in this case, *both the constructors, and the "onStarted()" methods, of all the Core module objects will be executed by the OS thread that runs the application's startup function* (because both the constructor, and the "Start()" method, of the Core object are invoked by the application's startup function).

- *Note 1:* the sequence of steps that have to be performed by the startup function of a libagents application is detailed in "The application startup function" paragraph later in this document
- *Note 2:* a detailed description of all the methods that play a role in the initialization process of the Core module as illustrated in Fig.7 is provided in "The Core module API" paragraph later in this document

Dynamically changing the Core module at runtime

The internal structure of libagents application's Core module can be changed dynamically at runtime by "killing" any of the Core object's sub-component objects, and/or by creating new sub-component objects of the Core object:

- each container object can dynamically "kill" any of its sub-component objects via a dedicated Kill*XXX*() method provided by said container object's base class; namely, the libagents "Core" base class provides a "KillTask()" method, the "Task" base class provides a "KillThread()" method, and the "Thread" base class provides a "KillAgent()" method.

  When an object is killed via the Kill*XXX*() method of its container object, *the entire hierarchy of sub-component objects of the killed object is recursively killed* (e.g. when the Core object kills one of its Task sub-component objects, all the Thread objects that are part of the killed Task are also killed, and all the Agent objects that are part of each killed Thread are also killed)

  - *Note:* because the Agent objects do not contain [a hierarchy of] other objects, killing an Agent object kills exclusively said Agent object
- each container object can dynamically create and start a new sub-component object by following a two-step sequence: specifically, a container object must first create the new sub-component object via the "new" operator, and then start the newly created object via the corresponding StartXXX() method. For example, an application's Core object 'myAppCore' can dynamically create a new Task object of type 'MyTask' (e.g. inside its onMessage() method in response to a received message) by first using "MyTask *myTask = new MyTask" to create the Task object, and then it must start the newly created Task object by using "StartTask(myTask)"

A detailed description of the Kill*XXX*() and Start*XXX*() methods is provided in "The Core module API" paragraph later in this document.

The messaging system

The application Core module's messaging system is *the foundation of the libagents event-driven data processing model*, and it provides the means by which *an Agent can exchange messages with other Agents in the application, as well as with the application's Shell module*.

- IMPORTANT: the libagents messaging system makes extensive use of internal message buffers, with ***one message buffer used for all the agents contained in each Thread object***, which temporarily queue each message in transit from its source to its destination; this internal implementation characteristic has the important consequence that ***there are no guarantees regarding the delay between the moment a message is sent from a source and the moment is it received at its destination***; additionally, said delay can be relatively large if

the intended receiver of a message is engaged in some heavy processing and/or if a relatively large number of messages (with some of them potentially requiring heavy processing) are already queued in the receiver's input buffer

The libagents v2.0.x library implements three types of messages, namely *"targeted messages", "broadcasted messages", and "I/O messages"*. The following paragraphs present the essential characteristics of the application Core module's messaging system, while the exact message formats and message-exchange methods are presented in "The libagents API" chapter later in this document.

Targeted messages

The "targeted messages" are messages sent by an agent that is part of a given Task object directly to another agent *that is part of the same Task object*. Specifically, the libagents "Agent" base class (and thus any user-defined agent derived from the "Agent" base class) implements a *"SendMessage()"* method which allows a source Agent object to send a targeted message to a specific destination Agent object, where said destination agent must be part of the same Task object as the source agent.

In other words, the "targeted messages" *are one-to-one direct messages exchanged between two agents that are part of the same Task object*.

- IMPORTANT: because targeted messages can be exchanged only between agents that are part of the same Task object, *inter-Task targeted messaging is not supported*

Broadcasted messages

The "broadcasted messages" are the foundation mechanism for the subscription-based communication model in a libagents application, specifically:

- a broadcasted message is a message sent by a source agent *towards a group of destination agents*, where said group of destination agents can consist of either all the agents that are part of the same Task as the source agent, or of all the agents that belong to a specific Thread object which itself is part of the same Task as the source agent; in other words, *a message can be broadcasted exclusively "within the boundaries" of an individual Task object, namely the Task object that contains the broadcaster agent*, while cross-task broadcasting is not supported
- when a source agent broadcasts a message towards a group of destination agents (as described above), *only a sub-set of said group of destination agents will actually receive the broadcasted message*: specifically, each destination agent of a broadcasted message will actually receive the message *if and only if it has been previously subscribed* to receive broadcasted messages from the source agent, while any and all agents which are part of the broadcasted message's destination group of agents but which are not subscribed to receive broadcasted messages from the source agent will not receive the broadcasted message

The libagents "Agent" and "Task" base classes provide a set of methods that control the flow of broadcasted messages from their source agent to their destination agents, specifically:

- each libagents Agent object (i.e. derived from the "Agent" base class) inherits a method from the "Agent" base class, namely *"Agent::BroadcastMessage()",* which allows it to broadcast a message towards *a specified group of destination agents* contained in its own Task object: namely, an agent object can broadcast messages either to all the agents contained in the broadcaster agent's own Task object, or to all the agents that are contained in a specific Thread object that is part of the broadcaster agent's own Task object, and
- each libagents Task object (i.e. derived from the "Task" base class) inherits two methods from the "Task" base class, namely *"Task::AddBroadcastSubscription()"* and

"Task::RemoveBroadcastSubscription()", which can be used to create/remove a *"broadcast subscription"* between a specified source agent and a specified subscriber agent, where *both the source agent and the subscriber agent must be part of the Task object for which the method has been invoked*:

- the *"Task::AddBroadcastSubscription()"* method adds a persistent communication link for broadcasted messages (i.e. for messages sent by an Agent object's "BroadcastMessage()" method) between a source agent and a subscriber agent [which are both part of the Task object for which the "Task::BroadcastMessage()" method has been invoked], i.e. this method *subscribes an agent to receive the messages that are broadcasted by a specified transmitter agent*
- the *"Task::RemoveBroadcastSubscription()"* method removes a persistent communication link (i.e. a broadcast subscription) that has been previously established via the "AddBroadcastSubscription()" method

In other words, by using the AddBroadcastSubscription() / RemoveBroadcastSubscription() methods of a given Task object, any number of receiver Agent objects that are part of said Task object can be subscribed /unsubscribed to/from receiving the messages that are broadcasted by a sender Agent object that is part of that same Task object

- IMPORTANT: because broadcasted messages can only be transmitted within the boundaries of a Task object, *inter-task broadcasting is not supported*

Fig.8 below exemplifies four agents which have been subscribed to a source agent, with all said agents (source and subscribed) belonging to multiple Threads in the same Task: thus, all the messages that are broadcasted by the source agent *towards all the agents in its own Task* are automatically received by, *and only by*, the four subscribed agents:

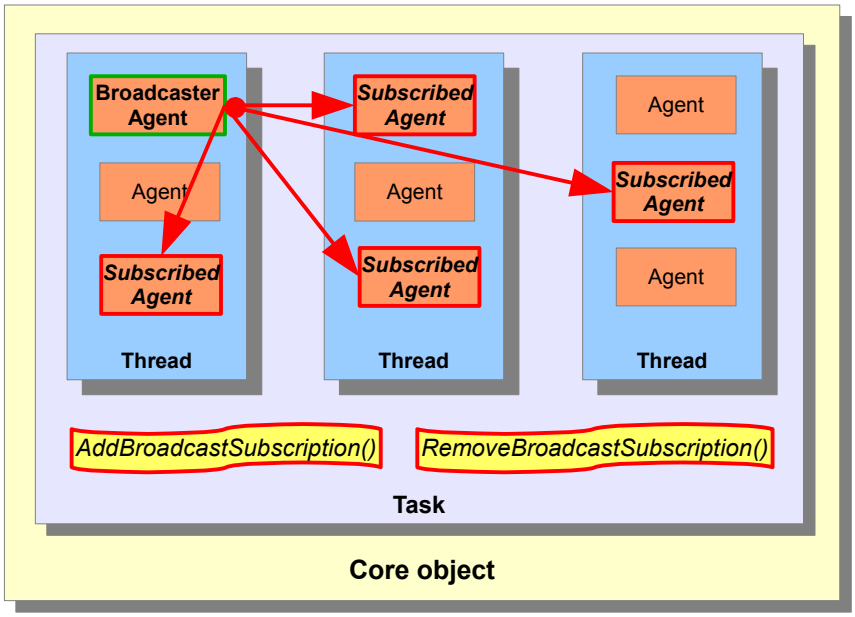

Fig.8: example of four broadcast subscriptions: each can be established/removed via the Task class' methods "AddBroadcastSubscription()"/"RemoveBroadcastSubscription()"

I/O messages

The "I/O messages" are messages that are exchanged between the Core and Shell modules of a libagents application via a dedicated "Intercom" communication port contained in the Core object.

The I/O messages routing scheme is illustrated in Fig.9 below:

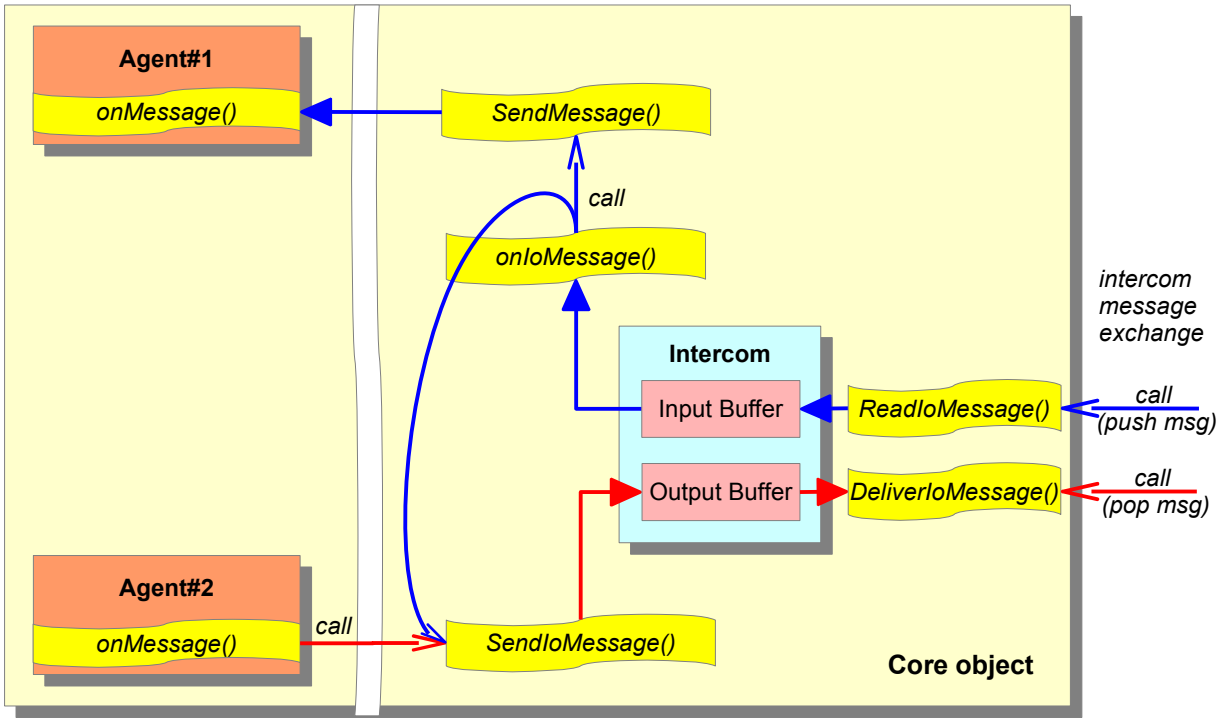

Fig.9: routing of the libagents I/O messages inside the Core object

- *Note 1:* for the sake of simplicity, the Task and Thread objects that "contain" the two agents Agent#1 and Agent#2 in Fig.9 above have been omitted from the diagram; said two agents (i.e. Agent#1 and Agent#2) can be part of the same, or a different, Thread within the same, or a different, Task
- *Note 2:* the solid block arrows in Fig.9 above represent messages (i.e. data structures) that are exchanged between objects, and the line arrows represent method invocations (i.e. function calls)

Following is a detailed description of each of the Intercom port-related methods illustrated in Fig.9 above:

- *"ReadIoMessage()"*: this method allows [a component of] the Shell module to place an I/O message in the Intercom's Input Buffer (this is a private Core object and is *not* user-accessible)
- *"DeliverIoMessage()"*: this method allows [a component of] the Shell module to extract an I/O message from the Intercom's Output Buffer
  - IMPORTANT: ***the "ReadIoMessage()" and "DeliverIoMessage()" methods can be invoked from any OS thread on the Shell module side, thus providing an inter-thread communication interface between an application's Core and Shell modules***
- the *"SendIoMessage()"* method: this is a *public* method of the Core base class which can be invoked by [a sub-component of] the Core module to place a message in the Intercom's Output Buffer
- the *"onIoMessage()"* method provided by the "Core" base class: this is a protected pure virtual method which is automatically triggered by libagents's internal mechanisms for each incoming I/O message that is received in the Intercom's Input Buffer (i.e. this method is similar to the Agent base class' "onMessage()" method, but it is triggered by an incoming I/O message), and it must be implemented by the Core object of a libagents application as part of the derivation process from the "Core" base class.

  The functional role of the "onIoMessage()" method is to act as an application-wide gateway for all the incoming I/O messages that are received by the Core module from the Shell

module: specifically, the “onIoMessage()” method must parse each incoming message, identify its content and intended recipient(s), and then, based on the message content and intended recipient:

- it can perform various specific actions based on the message content, then
- it can edit (read: modify) the message content, and then
- finally
  - it can dispatch the message to its intended recipient(s) by invoking the Core object's “SendMessage()” method (see the description of the “SendMessage()” method below), or
  - it can decide to re-route the message to another destination (including sending it back to the Shell module by placing it in the Intercom's Output Buffer - see the description of the “SendIoMessage()” method above), or
  - it can completely “filter out” (read: block) the message from being relayed to any destination at all

➢ IMPORTANT: the “onIoMessage()” method is executed by a dedicated OS thread whose exclusive role is to execute this method (plus any other functions and/or object methods that the “onMessage()” method may invoke during its execution)

- the *“SendMessage()”* method: this is a protected method provided by the “Core” base class which allows the Core application object to send a targeted message to any Agent object in the application (i.e. this method is similar to the Agent base class' “SendMessage()” method).
  The functional role of the “SendMessage()” method is to allow the Core object to route the messages it receives on its Intercom Input Port (see description of the “onIoMessage()” method above) to one or more Agents, depending on the message content

Inter-task communication

As previously described in “The Task objects” paragraph earlier in this chapter, the functional role of a Task object is to group together a set of agents that can *directly* exchange messages with each other by sending/receiving targeted messages and/or broadcasted messages, but which cannot directly exchange messages with agents that are part of a different Task object. However, agents belonging to different Tasks can communicate with each other *indirectly via the Core object's Intercom,* with the caveat that said messages must be formatted in such a way as to allow the “onIoMessage()” method to detect them as “inter-task” messages and properly forward them to their intended destination.

The inter-Task communication procedure described above is illustrated in Fig.10 below:

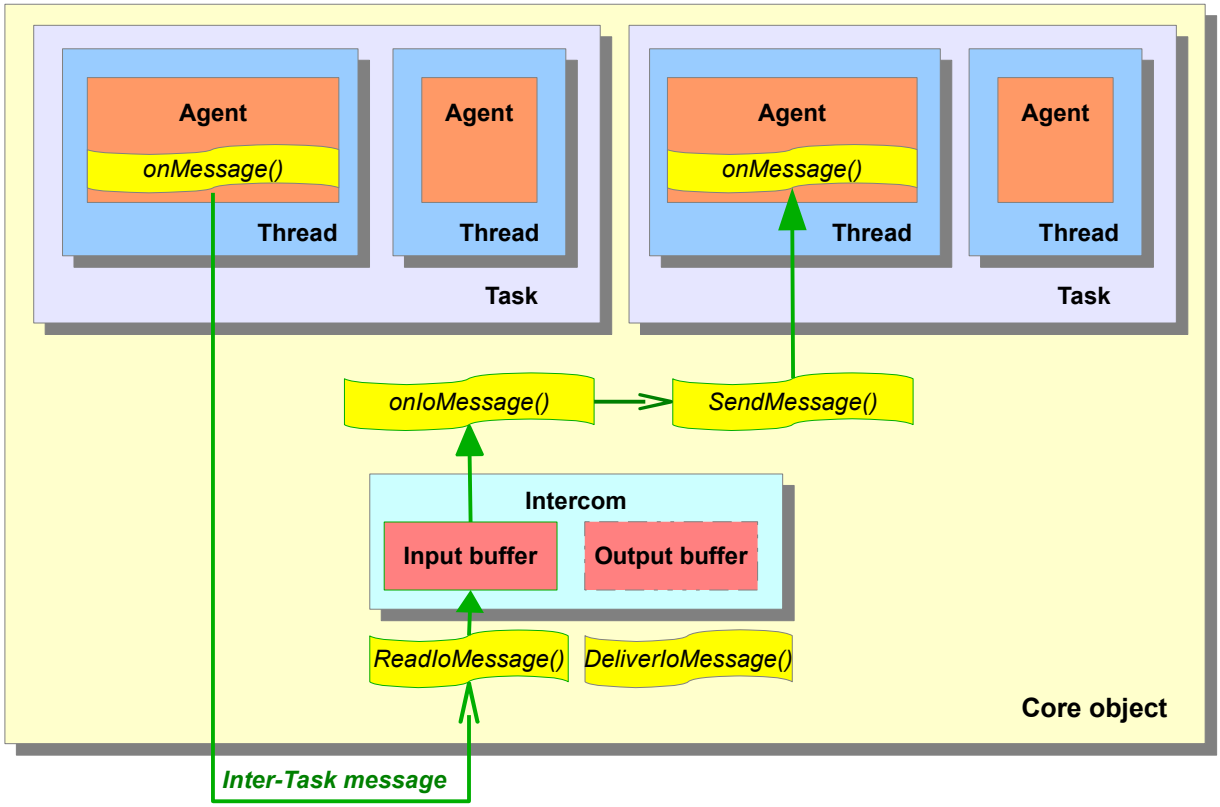

Fig.10: inter-Task communications via the Core object's Intercom

Scheduled messages

The message sending methods of the Agent base class (and thus of any type of user-defined agent derived from the "Agent" base class) implement a *message scheduling* function: specifically, when an agent object invokes one of its "SendMessage()" or "BroadcastMessage()" methods, said methods can be instructed to *delay* sending the message for a specified amount of time, and/or to *repeatedly* send the message either a specified number of times or until the auto-repeated send operation is explicitly canceled. This message scheduling facility enables the implementation of various kinds of time-dependent functionalities, e.g. messaging protocols that depend on retransmissions until the sent message is acknowledged, or condition monitors, or counter/timer objects, etc (the various message scheduling options are detailed in "The libagents API" chapter later in this document).

Message delivery

All the object methods which are sending a message (e.g. the Core object's SendMessage(), or the Agent objects' SendMessage() method, etc) return a value of '0' or 'false' if the message could not be placed in the destination message buffer, *which can only happen if the destination buffer is full* (the specific return values of each of these methods are detailed in "The libagents API" chapter later in this document); thus, if a certain piece of code in an application sends a message *assuming* that the message's destination buffer has enough room to store the sent message, it is highly recommended to cover said message-sending method in an 'assert()' statement (e.g. 'assert(myAgent.SendMessage(...))', etc) in order to force the application to shut down with an 'Assertion failed' error message if the destination buffer does not have enough room to receive the sent message.

- *Note:* a 'force()' statement may also be used instead of 'assert()' for application debugging, see the "Debugging support" paragraph for details

Multi-threading in the Core module

Because of the multi-threaded nature of the Core module, ***any and all functions and object methods that may be invoked concurrently from different OS threads must have a thread-safe implementation*** (e.g. they can be re-entrant, or they can be protected with multi-threading semaphores, etc), and ***any and all data elements that may be accessed concurrently from different OS threads must be explicitly protected against data races*** (e.g. they can be implemented as objects that are featured with a thread-safe read-modify-write method)

- IMPORTANT: ***the above multi-threading protection caveat applies to the application's Core, Task, Thread, and Agent objects, as well as to any user-defined*** Utility objects ***and/or application-wide global functions***

- *Note:* because the "onMessage()" methods of all the Agent objects that are contained in the same Thread object are executed by the same OS thread (see "The Thread objects" paragraph earlier in this chapter), any data elements that are accessed exclusively by Agent objects that are contained in the same Thread object are implicitly data race-free
  - for example, if a Thread object implements local data storage (see the "Object local data" paragraph earlier in this chapter), and said local data is accessed exclusively (directly or indirectly) by the "onMessage()" methods of the Agent objects contained in the Thread object, then said Thread object's local data is implicitly data race-free; this is illustrated in Fig.11 below:

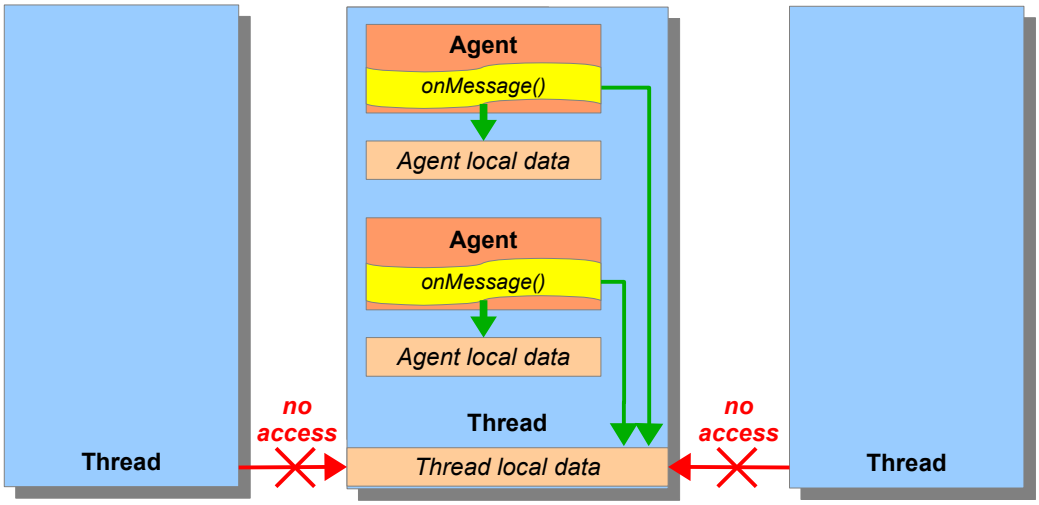

Fig.11: restricted data access on a Thread object's local data which guarantees the Thread object's local data is data race-free

The Shell module

As previously explained in "The architecture of a libagents application" chapter earlier in this document, the Shell module of a libagents application serves the role of interfacing the application core logic with the application's operating environment, and ***the libagents library does not impose any restrictions on the Shell module architecture other than it must contain the application's startup function and that it must provide a proper interface for communicating with the application's Core module*** (see Fig.1).

- *Note:* a reference Shell module architecture which can be used in conjunction with most of the host frameworks that are actively used and/or maintained at the time of writing this document is presented in the "Reference Shell module architecture" paragraph later in this document

The libagents API

This chapter describes the [user-accessible] data types and base classes of the libagents library, and explains how they should be used when creating a libagents application. Each class is presented with a synopsis of its declaration, and then each of its methods is described in detail.

The libagents libary namespace

The libagents library's API is declared inside the ***'AGENTS_Lib' namespace, and a convenience alias 'libagents'*** is #defined in the libagents configuration file 'libagents-config.h' (see "The libagents configuration file" paragraph earlier in this document); thus, in order to use any of the libagents API's objects/functions/constants in a libagents-based application, each such object/function/constant must be prefixed with 'AGENTS_Lib::' or 'libagents::', or the libagents namespace must be made visible in the application source file(s):

- for example, the libagents library provides a class 'Agent' which must be used as the base class for any agent object that is declared in a user applicaiton; then, in order to use the 'Agent' class in a user application, the class name must be prefixed with the libagents library namespace, or the libagents library namespace must be made visible in the source file, etc:

```
class MyAgent : libagents::Agent {...};
```

or:

```
using namespace libagents;
class MyAgent : Agent {...};
```

or:

```
using AGENTS_lib::Agent;
class MyAgent : Agent {...};
```

etc...

The libagents header file

The complete libagents API is "published" in the ***'libagents.h' header file, which must be #included in the header file of any file unit {.h+.cpp} which uses the libagents API***.

- for example, consider a libagents application 'MyApplication' whose Core object is called 'MyCoreObject', where said 'MyCoreObject' is be derived from the 'Core' base class provided by the libagents library (see "The Core object" paragraph earlier in this document): in this case, 'MyCoreObject' will be declared and defined in a file unit e.g. {mycoreobject.h+mycoreobject.cpp}, and, because 'MyCoreObject' uses (namely, is derived from) the 'Core' base class provided by the libagents library, the header file 'mycoreobject.h' must #include the 'libagents.h' header file:

```
// mycoreobject.h
#ifndef _MYCOREOBJECT_H_
#define _MYCOREOBJECT_H_

#include "libagents.h"
class MyCoreObject: libagents::Core {
   ...
};
#endif

// mycoreobject.cpp
#include "mycoreobject.h"
...
```

The libagents data types

The libagents data types are classes which are *used as arguments and return values* to/from various methods of the libagents base classes, and they are meant to be used *as-is* in a libagents application (i.e. these data types generally need not, and should not, be derived from in a user application).

The data_t type

The “data_t” data type is a data cell that can store *either* a string value, *or* an integer value, *or* a numeric double-precision value, *or* a void pointer value, by providing constructors from std::string, from int, from double, and from void*. A data_t data cell can be compared for equality/inequality with another data_t data cell via the '==' and '!= ' operators, and it has four methods 'number()', 'integer()', 'string()', and 'pointer()' which return the double, int, string, and respectively the void* value contained in the cell, or they throw a runtime exception if trying to extract an undefined value from a data cell (e.g. using the 'number()' method for a cell that has been initialized with a string value will throw a runtime exception).

- IMPORTANT: ***a 'data_t' data cell cannot be assumed to accurately store arbitrarily large integer values in the libagents v2.0.x library implementation*** because numeric values are stored as 'double' inside a 'data_t' data cell and the precision of 'double' numbers can be lower than the size of large integers (e.g. IEEE-754 defines 'single' with 24 bits of precision, while 'int' is implemented as a 32-bit integer on most systems); *however, it can be safely assumed that* ***a 16-bit integer value, either signed or unsigned (i.e. of type int16_t or uint16_t), can be accurately stored inside a 'data_t' data cell*** *on virtually any modern computer system.* ***In any case, if an attempt is made at runtime to load a 'data_t' data cell with an integer-type value which is too large to be accurately stored in the data cell, then the application will terminate with a runtime error***

Synopsis:

```
class data_t {

public:
   inline data_t() = default;                // creates an undefined-type data cell
   inline data_t(const char *s);             // creates a string-type data cell
   inline data_t(const std::string &s);      // creates a string-type data cell
   inline data_t(int i);                     // creates a numeric-type data cell
   inline data_t(unsigned u);                // creates a numeric-type data cell
   inline data_t(double d);                  // creates a numeric-type data cell
   inline data_t(void *p);                   // creates a void pointer-type data cell
   inline bool is_number() const;            // 'true' iff cell contains numeric value
   inline bool is_integer() const;           // 'true' iff cell contains integer value
   inline bool is_string() const;            // 'true' iff cell contains string value
   inline bool is_pointer() const;           // 'true' iff cell contains pointer value
   inline double number() const;             // return the cell's numeric value iff num
   inline int integer() const;               // return the cell's numeric value iff int
   inline const std::string &string() const; // return the cell's string value iff str
   inline void *pointer() const;             // return the cell's pointer value iff ptr
   inline bool operator==(const data_t &x) const;
   inline bool operator!=(const data_t &x) const;
};
```

Note: the *'number()', 'integer()', 'string()', 'pointer()'* methods above terminate the application with a runtime error if the *'data_t'* cell is not of the corresponding type

Examples:

```
data_t d1=12.34;        // d1 is assigned with data_t(12.34)
data_t d2=d1;           // d2 is assigned with d1 (with d1's value and d1's type)
data_t d3=0;            // d3 is assigned with data_t(0)
data_t s1=”abc”;        // s1 is assigned with data_t(“abc”)
data_t p=(void*) &d1;   // p is assigned with data_t(&d1)
```

```
bool b=d1.is_number();  // b is true: d1 contains a numeric value
bool b=d1.is_integer(); // b is false: ld contains a non-integer value
bool b=d1.is_string();  // b is false: d1 does not contain a string value
bool b=d1.is_pointer(); // b is false: d1 does not contain a pointer value
double d=d1;            //compile-time error: no implicit conversion d1 to numeric value
std::string s=s1;       // compile-time error: no implicit conversion s1 to string value
char *c=(char*)p;       // compile-time error: no implicit conversion p to pointer value
data_t d4=*(data_t*)p.pointer(); // valid: d4 gets assigned with d1
d=d1.number();          // valid: double d gets assigned with 12.34
s=s1.string();          // valid: string s gets assigned with "abc"
s=d1.string()+"x";      // run-time exception: d1 does not hold a string
d1==s1;                 // false: d1 holds a double while s1 holds a string
d1!=s1;                 // true: d1 holds a double while s1 holds a string
d1==d2;                 // true: both the types and the values match
d1==d3;                 // false: the types match but the values don't
d1=="xyz";              // false: d1 holds a double while data_t("xyz") holds a string
d1==12.34;              // true: d1 holds a double and it is equal to data_t(12.34)
d1>=12.34;              // compile-time error: data_t::operator>=(data_t) undefined
d1.number()>=12;        // true: d1 holds the numeric value 12.34
d1.integer()<13;        // run-time exception: d1 does not hold an integer value
```

The compid_t type

The "compid_t" data type is an alias of the "data_t" data type, and it is used (mostly) to represent the IDs (read: names) of Agent objects, Thread objects, and Task objects in a libagents application.

Because the Agent, Thread, and Task object names are of type "compid_t", they can have have a string value or an enum value*;* using enum values can significantly enhance the maintainability of an application.

- IMPORTANT: *an Agent, Thread, or Task object ID (of type "compid_t") which has a string value can contain* ***only alphanumeric characters, '.' (dot), '-' (minus), and '_' (underline),*** *and assigning an object name with a compid_t data which holds* ***a pointer value is not allowed***

The message_t type

The "message_t" data type is the data type of all the messages used by the libagents messaging system (i.e. targeted messages, broadcasted messages, and I/O messages, see the "The messaging system" paragraph earlier in this document), and it consists of *an array of one or more "data_t" cells*, plus a std::map<int, data_t> container which may hold various metadata that might occasionally need to be transferred between objects.

We will here-forth refer to the first "data_t" cell in a "message_t" message as the ***"message name"***, and to the remaining "data_t" cells as the ***"message payload"***. *A message must have at least one cell*, i.e. it may have no payload cells, but it must always contain the message name cell**.**

The "message_t" data type provides several convenient constructors, methods for accessing a cell in the message, methods for appending, inserting, and deleting a cell, for finding the message length (in number of cells), and for serializing and deserializing the message to/from a std::string.

Synopsis:

```
class message_t {

public:
   std::map<int,std::string> meta; // metadata with application-defined semantics
   message_t() = default;          // new empty message
   message_t(const std::string &s);// new one-cell message, cell assigned with a string
   message_t(const char *c);       // new one-cell message, cell assigned with a string
   message_t(int i);               // new one-cell message, cell assigned with a number
   message_t(unsigned u);          // new one-cell message, cell assigned with a number
   message_t(double d);            // new one-cell message, cell assigned with a number
   message_t(void *p);             // new one-cell message, cell assigned with a pointer
```

```
   message_t(const data_t &a);     // new one-cell message,cell assigned with data_t
   void clear();                   // clear message (message will contain zero cells)
   const data_t &operator[](unsigned i) const;       // const access to a cell
   data_t &operator[](unsigned i);                   // r/w access to a cell
   message_t &operator<<(const data_t &p);           // append a cell
   message_t &insert(unsigned pos, const data_t &p); // insert a cell
   bool erase(unsigned pos);                         // delete a cell if cell exists
   unsigned size() const;                  // message size in # of cells
   std::string prettyPrint(std::string separator="", bool decoration=0) const;
   bool to_string(std::string &s) const;   // serialize message into a std::string
   bool from_string(const std::string &s); // restore message from serialized format
};
```

Note: the 'message_t::to_string()' method can serialize messages with at most 255 data cells; any attempt to invoke the 'message_t::to_string()' method for a message larger than 255 data cells will terminate the application with a runtime error

Examples:

```
enum {MSGNAME_i1=1, MSGNAME_i2, MSGNAME_i3};        // message names with integer values
std::string MSGNAME_str="message name is a string"; // message name with string value
message_t ms(MSGNAME_str);      // ms is {"message name is a string"}
message_t m1=MSGNAME_i1;        // m1=message_t(MSGNAME_i1): m1 is {1}
data_t a2=MSGNAME_i2;           // a2=data_t(MSGNAME_i2): a2 is assigned integer value 2
message_t ma=a2;                // ma=message_t(a2): ma becomes {2}
data_t a3="string cell", a4(4); // a3=data_t("string cell"), data_t a4(4)
message_t m2;    // create empty m2
m2<<MSGNAME_i3;  // insert m2's name (i.e. cell #0): m2 becomes {3}
m2<<a3<<a4;      // append two cells to m2: m2 becomes {3, "string cell", 4}
m2.insert(1,3.3);// insert cell at position 1: m2 beomes {3, 3.3, "string cell", 4}
m2.erase(3);     // erase cell #3, return true: m2 becomes {3, 3.3, "string cell"}
int s=m2.size(); // m2.size() is 3
m2.erase(3);     // cannot erase cell #3, return false: message only contains cells 0..2
std::string pp=m2.prettyPrint(",",1) // pp becomes (w/o quotes): "#3,#3.2,$string cell"
```

The Core module API

As previously described in "The Core module" paragraph earlier in this document, the Core module of a libagents application consists of a singleton Core object whose type is derived from the libagents "Core" base class, and said Core object in turn contains a collection of application-specific objects whose types must be derived from the libagents "Agent", "Thread", and "Task" base classes. Based on the inclusion criteria among the libagents types of objects (i.e. Core object: {Task objects:{Thread objects:{Agent objects}}}), this paragraph is organized as a bottom-up presentation of the above-mentioned base types' APIs, starting with the Agent base class and ending with the Core base class.

- IMPORTANT: ***all the libagents component objects of the Core module (i.e. the objects whose types are derived from the Agent, Thread, Task, and Core base classes) must be created as dynamically allocated objects, i.e. they must be created via the 'new' operator***. *Trying to instantiate the above-mentioned objects as automatic or static variables will cause the application to have an undefined behavior*

The Agent class

The Agent class is the base class from which all user-defined [types of] Agent objects must be derived. The Agent objects are the elementary data processing units in a libagents application (see "The Agent objects" paragraph earlier in this document), and they are logically grouped together into Thread objects (see "The architecture of a libagents application" paragraph earlier in this document). Agent objects can send and receive messages to/from other agents, and to/from the Core application's Intercom object (see "The messaging system" paragraph earlier in this document).

Synopsis:

```
class Agent {

protected:
   virtual void onStarted()=0;
   virtual bool onIoMessage(const message_t &msg,
                            const compid_t &sourceThreadId,
                            const compid_t &sourceAgentId)=0; // MUST return 'true'!!!
   uint64_t SendMessage(const message_t &msg,
                        const compid_t &destThreadId,
                        const compid_t &destAgentId,
                        float schedule=0,
                        int repeat=1,
                        int expire=INT_MAX);
   uint64_t BroadcastMessage(const message_t &msg,
                             const compid_t &destThreadId="*",
                             float schedule=0,
                             int repeat=1,
                             int expire=INT_MAX);
   bool CancelMessage(uint64_t msgId);
public:
   Agent(const compid_t &id);
   virtual ~Agent();
   compid_t agentId();
   Thread *parentThread();
   Core *core;
};
```

Details:

- **Constructor:** the Agent constructor has to be provided with a const compid_t &id argument which specifies the Agent object's id, and it has to be invoked by the constructor(s) of all user-defined agent types (derived from the Agent base class)
  - ***this method is executed by the OS thread that creates the Agent object***
- **Destructor:** performs some internal house keeping
- **agentId():** this method returns the Agent object's id (of type 'compid_t')
- **onStarted():** this is a pure virtual method which has to be implemented by all the user-defined Agent objects of a libagents application, and it is automatically invoked right after the Agent object has been started
  - an Agent object is started by its Thread parent object's 'StartAgent()' method which is provided by the libagents 'Thread' base class; the 'Thread::StartAgent()' method is detailed in the following "The Thread class" paragraph
  - ***an Agent object's onStarted() method is executed by the OS thread that invokes the Thread::StartAgent() method*** (see also the "Initializing the Core module" and "The Thread objects" paragraphs earlier in this document)
- **onMessage():** this is a pure virtual method which has to be implemented by all the user-defined Agent objects of a libagents application, and it is the exclusive data processing function of an Agent object. This method is automatically invoked every time an Agent object receives a new message, with its arguments specifying the incoming message (const message_t &msg) and the source of the message (const compid_t &sourceThreadId, const compid_t &sourceAgentId)
  - this method is executed by the underlying OS thread associated with the Thread object that contains the agent (see "The Thread objects" paragraph earlier in this document)
  - if any of sourceThreadId==”” or sourceAgentId==”” then it is guaranteed that both sourceThreadId==”” and sourceAgentId==””, and the incoming message is an I/O message that has been sent from the Core object (via "Core::SendMessage()" method)

- if any of sourceThreadId!=”” or sourceAgentId!=”” then it is guaranteed that both sourceThreadId!=”” and sourceAgentId!=””, and the incoming message is either a broadcasted message or a targeted message received from another Agent object
- the return value of this method is not relevant in this version of libagents; however, ***this method must return 'true' when using the libagents v2.0.x library***

- **SendMessage():** invoking this method will send a targeted message to another Agent object that belongs to the same Task object as the sender agent (including to itself). The method arguments specify the message to be sent (const message_t &msg), the target of the message (const compid_t &destThreadId, const compid_t &destAgentId), and the message scheduling scheme (float schedule, int repeat, int expire)
  - the return value of SendMessage() is a unique message id of the sent message, or zero if the message cannot be sent: a targeted message will fail to be sent *if and only if* the internal message buffer of the receiving agent is full at the time when the method is invoked
  - each of the destAgentId and destThreadId arguments can have the special value “$” which represents the names of the current Agent and respectively of the current Thread; e.g. if destAgentId==”$” and destThreadId==”$” then the message is sent as a targeted message to the sender agent itself
  - the schedule argument specifies the delay between the moment when the method is invoked and the moment when the message will actually be sent, and it has the following syntax:
    - the integer part specifies the delay “base” value, in milliseconds
    - the fractional part specifies the “dispersion” value for the delay, in percents
    - for example, schedule=1000.00 means the message will be sent with a delay of exactly one second from the moment when the method is invoked, while schedule=1000.50 specifies that the send delay will be a random value between 500ms and 1500ms
  - the repeat argument specifies how many times the message will be automatically [re]sent *for one invocation* of the method
  - the expire argument specifies the maximum delay, in milliseconds, allowed between the moment when the scheduled message should be received at its destination (according to the sending schedule) and the moment when the message can actually be delivered (and start being processed) at its destination: specifically, a message cannot be delivered to a given destination agent at the required time if/while at that time any agent that is running in the same OS thread with the destination agent is busy processing another message (i.e. until the busy agent's onMessage() method completes execution); in this case, the message delivery is postponed (by the libagents internal mechanisms) until the busy agent's onMessage() method first completes its “current” execution, and then the message is actually delivered to its target agent only if the delay between the moment the message should haven been delivered [to the target agent] and the moment it can actually be delivered does not exceed the expire period; alternatively, if said delay exceeds the expire period, then the message will no longer be delivered to the target agent and the message will be permanently lost
- **BroadcastMessage():** invoking this method will broadcast a message to [all the agents in] the specified thread, where *the specified thread must belong to the same Task object as the sender agent*. The method arguments specify the message contents (const message_t &msg), the Thread object where the message is to be broadcasted (const compid_t &destThreadId), and the message scheduling scheme (float schedule, int repeat, int expire)

    - the broadcast subscriptions, i.e. which of the agents in the specified thread will be receiving the broadcasted message, are set up by the AddBrodcastingRoute() and RemoveBrodcastingRoute() methods of the Task base class; these methods are detailed in "The Task class" paragraph later in this chapter
    - the return value of BrodcastMessage() is a unique message id of the broadcasted message, or zero if the message cannot be sent: a broadcasted message will fail to be sent *if and only if* the internal buffer used by the libagents messaging system to queue the broadcasted message is full at the time when the method is invoked
    - for the special value destThreadId=="$" the message is broadcasted to all the agents that belong to *the same Thread object as the broadcasting agent*
    - for the special value destThreadId=="*" the message is broadcasted to all the agents that belong to *the same Task object as the sender agent* (i.e. to all the Agent objects that belong to all the Thread objects that are part of the same task as the broadcasting agent)
    - the schedule, repeat, and expire parameters have the same semantics as their SendMessage() method's counterparts
- **CancelMessage():** this method can be invoked by an agent to cancel all pending dispatches of a message that has been scheduled via SendMessage() or BroadcastMessage()
    - the msgId argument specifies the message id of the message that has to be canceled, as it has been returned by SendMessage() or BroadcastMessage()
    - if at the time when this method is invoked the message has not yet been dispatched to its target(s) the full number of times specified by the 'repeat' argument used when the message has been scheduled (i.e. the message still has to be dispatched one or more times to its target(s) at the moment this method is invoked), then all pending scheduled dispatches of the message will be canceled and the method returns 'true'
    - if at the time when this method is invoked the message has already been dispatched to its target(s) the full number of times specified by the 'repeat' argument used when the message has been scheduled, then this method has no object (i.e. there are no pending dispatches for the message to be canceled) and the method returns 'false'
- **parentThread():** this method returns a pointer to the Thread object that "contains" the agent; the return value is of type Thread*
- **core:** this property holds a pointer to the application's Core object (of type Core*)

The Thread class

The Thread class is the base class from which all user-defined [types of] Thread objects must be derived. The Thread objects are the elementary execution-scheduling units of a libagents application, and they group together agents whose "onMessage()" methods are executed by the same OS thread (see "The Thread objects" paragraph earlier in this document).

Synopsis:

```
class Thread {

protected:
   virtual void onStarted()=0;
public:
   Thread(const compid_t &id, unsigned messageBufferSize=MESSAGE_BUFFER_SIZE);
   virtual ~Thread();
   bool StartAgent(Agent *a);
   bool KillAgent(const compid_t &agentId);
   Agent *childAgent(const compid_t &agentId);
   compid_t threadId();
```

```
    Task *parentTask();
    Core *core;
};
```

Details:

- **Constructor:** the constructor has to be provided with a const compid_t &id argument which specifies the Thread object's id, and optionally with a const unsigned messageBufferSize argument which determines the size of the message buffer used for inter-Agent communication within the Thread object (all Agent objects within a given Thread object share the same message buffer). The Thread base class constructor has to be invoked by the constructor(s) of all user-defined thread types (derived from the Thread base class)
  - ***this method is executed by the OS thread that creates the Thread object***
  - if the messageBufferSize argument is not specified then the inter-Agent message buffer size is set to the libagents default value
- **Destructor:** stops the Thread object, then unregisters and Kills all the Agent objects that are part of the Thread object (by successively invoking KillAgent() for each Agent contained in the Thread object, see the description of the 'KillAgent()' method below), and finally the Thread object is removed from memory
  - IMPORTANT: ***the destructors of the application-specific user-defined Thread objects derived from the 'Thread' base class need not, and must not, explicitly delete the Agent objects that are part of the Thread object,*** as these operations are automatically performed by the destructor of the 'Thread' base class
- **threadId():** this method returns the Thread object's id (of type 'compid_t')
- **onStarted():** this is a pure virtual method which has to be implemented by all the user-defined Thread objects of a libagents application, and it is automatically invoked right after the Thread object has been started
  - a Thread object is started by its parent Task object's 'StartThread()' method which is provided by the libagents Task base class; the Task::StartThread() method is detailed in the following "The Task class" paragraph
  - ***a Thread object's onStarted() method is executed by the OS thread that invokes the Task::StartThread() method*** (see also the "Initializing the Core module" paragraph earlier in this document)
- **StartAgent():** invoking this method will cause the specified agent to be registered with (i.e. made part of) the invoking Thread object, and then this method calls the agent's onStarted() method as the last step of its execution (see the "Initializing the Core module" paragraph earlier in this document). When this method completes execution (i.e. when it returns control to its caller), the started agent will be prepared to receive and process new messages, but the started agent will not effectively start receiving and processing incoming messages (i.e. it will not start executing its onMessage() method for each received message) unless/until its container Thread object has also been started
  - IMPORATNT: as it can be seen from the StartAgent() description above, ***in order to have an Agent object start receiving and processing messages, both the Agent object itself and the Thread object that contains the Agent object have to be started;*** a Thread object is started by the 'StartThread()' method provided by the Task base class, which is described in the following paragraph "The Task class", the "StartThread()" method
  - the argument of this method is a *pointer* to an Agent object which has been previously created via the 'new' operator

    - the bool return value of this method is 'true' if the Agent object has been successfully started by this method, and it is 'false' if the Agent object is found to have been already started at the moment this method is invoked

- **KillAgent():** this method *stops and then destroys* the Agent object whose id is specified in the function argument (of type const compid_t&), i.e. the libagents messaging system stops delivering messages to the killed agent and then the specified Agent object is deallocated from the application's runtime memory by invoking its destructor
    - all the broadcasting routes in which the killed Agent was part of are removed, i.e. a killed Agent will no longer be connected, neither as a broadcasting source nor as a broadcasting destination, with any other Agents (see the "Broadcasted messages" paragraph earlier in this document)
    - the bool return value of this method is 'true' if the specified Agent object is found to be registered with (i.e. part of) the invoking Thread object at the moment the method is invoked and the agent has been successfully stopped and destroyed, and it is 'false' if the specified Agent object is not found to be part of the invoking Thread object at the moment the method is invoked or if the specified Agent object could not be [stopped and] destroyed
- **childAgent():** this method returns a *pointer* to the Agent object whose id is specified as the method argument, or it returns nullptr if the specified id does not represent a child agent of the invoking thread; the return value is of type Agent*
- **parentTask():** this method returns a pointer to the Task object that "contains" the thread; the return value is of type Task*
- **core:** this property holds a pointer to the application's Core object (of type Core*)

The Task class

The Task class is the base class from which all user-defined [types of] Task objects must be derived. The Task objects logically group together Thread objects whose contained agents can communicate with one another by exchanging direct messages (see "The Task objects" paragraph earlier in this document).

Synopsis:

```
class Task {

protected:
   virtual void onStarted()=0;
public:
   Task(const compid_t &id);
   virtual ~Task();
   bool StartThread(Thread *t);
   bool KillThread(const compid_t &threadId);
   Thread *childThread(const compid_t &threadId);
   compid_t taskId();
   Core *parentCore();
   Core *core;
   bool AddBroadcastSubscription(const data_t &messageName,
                                 const compid_t &sourceThreadId,
                                 const compid_t &sourceAgentId,
                                 const compid_t &destThreadId,
                                 const compid_t &destAgentId);
   bool RemoveBroadcastSubscription(const data_t &messageName,
                                    const compid_t &sourceThreadId,
                                    const compid_t &sourceAgentId,
                                    const compid_t &destThreadId,
                                    const compid_t &destAgentId);
};
```

Details:

- **Constructor:** the constructor has to be provided with a const compid_t &id argument which specifies the Task object's id, and it has to be invoked by the constructor(s) of all user-defined task types (derived from the Task base class)
  - ***this method is executed by the OS thread that creates the Task object***
- **Destructor:** unregisters and Kills all the Thread objects that are part of the Task object (by successively invoking KillTask() for each Task contained in the Thread object, see the description of the 'KillTask()' method below), and finally the Task object is removed from memory
  - IMPORTANT: ***the destructors of the application-specific user-defined Task objects derived from the 'Task' base class need not, and must not, explicitly delete the Thread objects that are part of the Task object,*** as these operations are automatically performed by the destructor of the 'Task' base class
- **taskId():** this method returns the Task object's id (of type 'compid_t')
- **onStarted():** this is a pure virtual method which has to be implemented by all the user-defined Task objects of a libagents application, and it is automatically invoked when the Task object is started
  - a Task object is started by its Core parent object's 'StartTask()' method which is provided by the libagents Core base class; the 'Core::StartTask()' method is detailed in the following "The Core class" paragraph
  - ***a Task object's onStarted() method is executed by the OS thread that invokes the 'Core::StartTask()' method*** (see also the "Initializing the Core module" paragraph earlier in this document)
- **StartThread():** invoking this method will cause the Thread object passed as its argument to be registered with (read: made part of) the invoking Task object, then a new OS thread will be allocated to the specified Thread object (see also "The Thread objects" paragraph earlier in this document), and finally this method calls the newly started Thread object's 'onStarted()' method (see the "Initializing the Core module" paragraph earlier in this document)
  - the argument of this method is a *pointer* to a Thread object which has been previously created via the 'new' operator
  - the bool return value of this method is 'true' if the Thread object has been successfully started by this method, and it is 'false' if the Thread object is found to have been already started at the moment this method is invoked
- **KillThread():** this method *stops and destroys* the Thread object whose id is specified in the function argument (of type const compid_t&), *together with all the Agent objects that belong to the specified Thread object*, i.e. both the Thread object *and all the contained Agent objects* are deallocated from the application's runtime memory by invoking their corresponding destructors
  - the bool return value of this method is 'true' if the specified Thread object is found to be part of the invoking task at the moment when this method is invoked and the thread has been successfully stopped and destroyed, and it is 'false' if the specified Thread object is not found to be part of the invoking task at the moment when this method is invoked or if the method failed to stop and destroy the specified Thread object
- **childThread():** this method returns a pointer to the Thread object whose id is specified as method argument, or it returns 'nullptr' if the specified id does not represent a child thread of

the invoking task; the return value is of type Thread*

- **parentCore():** this method is a getter for the 'core' property (see below), and it has been included in the API only for consistency with the Thread and Agent classes' '*parentXXX()*' methods (the 'core' pointer can be used directly instead)
- **core:** this property holds a pointer to the application's Core object (of type Core*)
- **AddBroadcastSubscription():** invoking this method sets up a subscription (i.e. a persistent communication link *for broadcasted messages*, see the "Broadcasted messages" paragraph earlier in this document) between two agents that belong to the Task object that invoked the method. Each subscription that has been set up between two agents is stored as a separate entry in an internal libagents data structure that we'll refer to as the libagents "*broadcast subscriptions table"*
  - the {sourceThreadId, sourceAgentId} pair of arguments specifies the source agent
  - the {destThreadId, destAgentId} pair of arguments specifies the destination agent
  - the messageName argument restricts the subscription of the specified destination agent to the specified source agent's broadcasted messages based on the message name, specifically:
    - the "message name" is the first cell in a message, see "The message_t data type" paragraph earlier in this document
    - if the 'messageName' argument has the special value of "*" then all the messages broadcasted by the source agent will be received by the destination agent
    - if the 'messageName' argument has any other value except "*" then the destination agent will be subscribed *only to those messages [broadcasted by the source agent] for which the message name matches the messageName argument*, while all the other messages [broadcasted by the source agent] will not be received by the destination agent
  - the specified source and destination endpoints must be valid (read: must be registered agents) at the time when this method is invoked; if any or both endpoints are not valid the application will terminate immediately with a runtime error
  - the bool return value is 'true' if the broadcast subscriptions table does not already contain the specified subscription at the time when this method is invoked (i.e. the specified subscription has been added as a new entry in the broadcast subscriptions table), and it is 'false' if an entry for the specified subscription already exists in the broadcast subscriptions table at the time when the method is invoked (i.e. a new entry did not have to be created)
- **RemoveBroadcastSubscription():** this method removes a subscription that has been previously set up by the 'AddBroadcastSubscription()' method; the arguments of this method have the same semantics as their 'AddBroadcastSubscription()' counterparts
  - after this method completes its execution it is guaranteed that the subscription will no (longer) exist in the broadcast subscriptions table (whether or not the subscription exists in the broadcast subscriptions table when the method is invoked)
  - the bool return value is 'true' if the specified subscription has been found in the broadcast subscriptions table (and it has thus been removed), and it is 'false' if the specified subscription has not been found (i.e. there was no subscription as specified by the method arguments to be removed from the broadcast subscriptions table)

The Core class

The Core class is the base class from which a libagents application's Core [type of] object has to be derived. The Core object of a libagents application logically groups together all the application's component objects, i.e. the application's Task, Thread and Agent objects (see "The Core object" paragraph earlier in this document), and it contains methods for managing the I/O messages that are exchanged between the Core object and the Shell module (see the "I/O messages" paragraph earlier in this document).

Synopsis:

```
class Core {

protected:
   virtual bool onIoMessage(const message_t &msg)=0; // MUST return true'!!!
   bool SendMessage(const message_t &msg,
                    const compid_t &destTaskId,
                    const compid_t &destThreadId,
                    const compid_t &destAgentId);
   virtual int onStarted(const message_t &args)=0;
public:
   Core(unsigned int messageBufferSize=MESSAGE_BUFFER_SIZE,
        unsigned debugLevel=DEBUG_NONE)     // NYI in libagents v2.0.x !!!;
   virtual ~Core();
   int Start(message_t args="");
   bool StartTask(Task *t);
   bool KillTask(const compid_t &taskId);   // NYI in libagents v2.0.x !!!
   Task *childTask(const compid_t &taskId);
   bool SendIoMessage(const message_t &msg);
   bool DeliverIoMessage(message_t *msg);
   bool ReadIoMessage(const message_t &msg);
};
```

Details:

- **Constructors:** the Core base class constructor takes two arguments 'messageBufferSize' 'debugLevel' which both have default values, and it has to be invoked by the constructor(s) of a libagents application's Core object (derived from the Core base class)
  - ***this method is executed by the OS thread that creates the Core object***
  - the 'debugLevel' argument contains a bitwise set of debug-related flags, and its default value is 'DEBUG_NONE'
    - *Note:* the 'debugLevel' argument has no effect in libagents v2.0.x library
  - the 'messageBufferSize' argument can be used to override the default size of the Core object's Intercom message buffers (expressed in number of messages that each Intercom buffer can store)
    - ✗ *for example,* the constructor of the release version (DEBUG_NONE) of an application's Core object of type 'MyCoreApplication' which uses Intercom buffers of 500 messages will read:

      ```
      MyCoreApplication::MyCoreApplication() : Core(500) {...}
      ```

- **Destructor:** *this is a dummy destructor in libagents v2.0.x* (i.e. it does not contain any cleanup code); libagents v2.0.x assumes that the application will not perform any meaningful actions and will immediately exit after its Core object has been destroyed
- **Start():** invoking this method will perform a number of internal initialization steps required for the Core application object, and then this method will call the Core object's onStarted() method as the last step of its execution (see also the "Initializing the Core module" paragraph earlier in this document)

    - this method's argument is of type 'message_t' and it is passed over to the Core object's onStarted() method; in this way, any number of 'data_t' values can be passed to this method as part of the message_t argument
- this method returns the value returned by the Core object's onStarted() method
- **onStarted():** this is a pure virtual method that has to be implemented by the Core object of a libagents application, and it is automatically invoked when the Core object is started
    - the Core object is started by invoking its own 'Core::Start()' method which is provided by the "Core" base class (see also the "Initializing the Core module" paragraph earlier in this document)
    - this method's argument and return value are passed over by, and respectively returned to, the core object's Start() method (see the 'Start()' method description below)
    - ***this method method is executed by the OS thread that invokes the Core object's Core::Start() method*** (see also the "Initializing the Core module" paragraph earlier in this document)
- **onIoMessage():** this is a protected pure virtual method which has to be implemented by the Core object of a libagents application, and it is automatically invoked each time an I/O message is received from the Shell module
    - ***this method is executed by a dedicated OS thread*** whose sole role is to extract, pre-process, and then dispatch *sequentially, one at a time,* the I/O messages that have been received in the Core object's Intercom Input Buffer (see also the Core object's "SendIoMessge()" method below); thus, ***this method is guaranteed to not be invoked concurrently by multiple OS threads***
    - the return value of this method is not relevant in this version of libagents; however, ***this method must return 'true' when using the libagents v2.0.x library***
- **SendMessage():** invoking this method will send a targeted message to a destination Agent object
    - the message to be sent is specified in the const message_t &msg argument
    - the destination Agent object can be part of any task, and it is specified by the const compid_t &destTaskId, const compid_t &destThreadId, and const compid_t &destAgentId arguments
    - returns 'true' if the targeted message can be sent to its destination, or 'false' if the message cannot be sent: a targeted message will fail to be sent *if and only if* the internal message buffer of the receiving agent is full at the time when the method is invoked
- **SendIoMessage():** invoking this method will queue an I/O message in the Core object's Intercom Output Buffer
    - returns 'true' if the I/O message has been successfully placed in the Core object's Intercom Output Buffer, or 'false' if the I/O message cannot be sent (this latter situation can occur *if and only if* the Intercom Output Buffer is full at the time when this method is invoked)
- **DeliverIoMessage():** this method extracts a message from the Intercom's Output Buffer
    - ***this method is meant to be invoked from the Application Shell*** in order to read an I/O message sent from the Core object via the 'Core::SendIoMessage()' method
    - if the Core object's Intercom Output Buffer contains any messages then:
        - extracts the oldest message from the buffer and places it in the location specified by

the '*msg' parameter
        - returns 'true'
    - else:
        - clears the message location specified by the '*msg' parameter
        - returns 'false'
- **ReadIoMessage():** this method pushes a message into the Core object's Intercom Input Buffer
    - ***this method is meant to be invoked from the Application Shell*** in order to send an I/O message to the Core object (the message will be captured by the 'Core::onIoMessage()' method)
    - returns 'true' if the message has been successfully placed in the Core object's Intercom Input Buffer, or 'false' if the message could not be placed (this latter situation can occur *if and only if* the Intercom Input Buffer is full at the time when this method is invoked)
- **StartTask():** this method performs a number of internal initialization steps in preparation for starting the specified task, and then it will invoke the specified Task object's 'onStarted()' method (see the "The application startup function" paragraph earlier in this document)
    - the argument of this method is a *pointer* to a Task object which has been previously created via the 'new' operator
    - the bool return value of this method is 'true' if the Task object has been successfully started by this method, and it is 'false' if the Task object is found to have been already started at the moment this method is invoked
- **KillTask():** *this method is not implemented in the libagents v2.0.x library*; trying to use this method in an application will cause a compilation error
- **childTask():** this method returns a *pointer* to the Task object whose id is specified as method argument, or it returns 'nullptr' if the specified id does not represent any of the application's tasks (i.e. no Task object with the specified id is part of the application); the return value is of type Task*

The Sys class

The Sys class is a ***static class*** and it contains utility functions which can be invoked by explicitating the classname 'Sys'.

Synopsis:

```
class Sys {

public:
	static uint64_t ticker();
	static int threads();
};
```

Details:

- **ticker():** this method returns the host system "time ticks" that lapsed since the Core application object has been started (i.e. since its 'Start()' method has been invoked), in milliseconds
- **threads():** this method returns the maximum number of concurrent CPU threads that the host system can allocate to the application (caution: the number of physical cores on which the threads are running can be lower, and *can vary* during the application lifetime)

- ✗ *example:* fetch the time (in host system “time ticks”) since an application has been started:

```
// .cpp file
using namespace AGENTS_Lib;
[...]
uint64_t runTime=Sys::ticker();
```

Debugging support

The libagents library configuration file 'libagents-config.h' (see “The libagents configuration file” paragraph earlier in this document) #defines a 'force()' macro as an inline funtion 'force_()', and the 'force()' macro is used in various places inside the libagents library to check for the validity of a condition (in a similar way to the standard 'assert()' macro). Most of the current IDEs will allow setting up a breakpoint at the 'exit(1)' statement in the definition *of the 'force_()' function in the 'libagents-config.h' file*, and the developer can then trace back in the debugger the call path that led to the failed 'force()' test.

- ✗ for example, a 'force' assertion is used inside the libagents library to enforce that an Agent object does not receive a message before said Agent object has been started; thus, if a user program will send a message to an Agent object that has not been started, the above-mentioned 'force' assertion will fail by executing 'exit(1)' statement. In this context, by running the program in debug mode and placing a breakpoint at the 'exit(1)' statement inside the definition *of the 'force_()' function*, the program developer can stop the program right before it exits and check the call trace, the stack variables, and the global variables in the program, and can thus identify the source of the problem
- *Note:* the 'force()' assertion can also be used in the user applications, thus providing the means to stop the application before the application exits, and the stack call trace can then be examined in your IDE debug mode) to determine the position of the failed 'force()' assertion in the program's source code

The libagents library also implements a 'debugLevel' feature which is set in the constructor of a libagents application's Core base class. ***Applications based on the libagents v2.0.x library implementation can use only debugLevel=DEBUG_NONE***; other debugLevel values will be implemented in future versions.

Design considerations

This chapter contains several key design considerations and suggestions regarding the architectural details of a libagents-based application.

Design partitioning

The process of designing a libagents application must start with identifying which, if any, of the various processing functions that the application must perform can be separated into different Tasks, based on the functional characteristic of a Task object of grouping together agents that can easily exchange direct messages with each other (by sending/receiving targeted and/or broadcasted messages), but can only exchange inter-Task messages via the special inter-Task communication procedure previously described in the "Inter-task communication" paragraph.

- for example, a "servent" application (i.e. an application that incorporates both a *serv*er and a cli*ent* for a given communication protocol) may be split in two separate Tasks, with each of the two tasks independently implementing the server functionality and respectively the client functionality; this two-task partitioning of the application functionalities can be (but not necessarily is) well suited for servent[-like] applications because, by the very nature of such applications, it is usually a good practice to enforce a design partitioning that will minimize the communication needs between the server and the client components

After the Task-level structure of the application has been defined, the second step is to distribute each Task object's functionality among multiple Agent objects by first identifying which actions can be performed in parallel (i.e. by separate agents), and then grouping the resulting agents into Thread objects by taking into consideration the ways in which the agents can best interact with each other (e.g. by thread-safely sharing Thread-local data structures, see the "Multi-threading in the Core module" paragraph earlier in this document, etc) and/or how they may interfere with each others' execution depending on whether they are part of the same, or different, Thread objects (see "The Thread objects" paragraph earlier in this document).

- for example, a common task that is particularly well suited to taking advantage of hardware multi-threading is the spell checking function of a text editor: in this case, a possible algorithm for utilizing the full CPU power for the spell checking function would be to first determine the number "T" of available hardware threads on the system (a Core object method 'Core::processors()' is provided for this purpose, see "The Core class" API paragraph earlier in this document), then *create a number of T-1 'Thread' objects with one 'Agent' object in each 'Thread' object*, and then the document to be spell-checked should be split into T-1 roughly equal chunks (taking into consideration the word boundaries) and each Agent in each Thread should be given one of the T-1 document chunks for spell checking: in this way the hardware processor(s) computing power can be utilized at the maximum possible extent, as the operating system will have the opportunity to allocate T-1 *hardware threads* (out of the total number T available) for the spell checking function, and one hardware thread will be reserved for any other processing that might be required during the execution of the spell checking function
- alternatively, consider an application that implements a P2P node which maintains multiple P2P links with a set of peers, where each such link is managed by a separate Agent object which maintains the link's state variables and exchanges data and control messages with the corresponding peer. In such an application, the general rule is that the messaging part of the node's algorithm is not CPU-bound (i.e. the processor will typically be used at far less than 100% of its capacity *for the messaging algorithm itself*) and the actual sending/reception of a new message will typically require only brief bursts of CPU utilization. In this case, the Agent objects that manage the individual P2P links can all be implemented as components

of a single Thread object because each link Agent will be spending most of the time in idle state (waiting to send or receive a new message), and should P2P network traffic occasionally occur on multiple links simultaneously (thus bringing the underlying OS thread that runs the link Agents to higher CPU utilization) this will only represent brief transitory bursts which will not affect the overall functionality of the node in a significant way

Reference Shell module architecture

As previously explained in "The architecture of a libagents application" chapter earlier in this document, the Shell module of a libagents application serves the role of interfacing the application core logic with the application's operating environment. In this context, ***this chapter describes a reference Shell module architecture which allows a simple and well formalized procedure for implementing the Shell module "on top of" most modern host frameworks which are actively maintained at the time of writing this document*** (e.g. win32/64, Qt, GTK, wxWidgets, etc).

- IMPORTANT: as previously explained in "The Shell module" paragraph earlier in this document, ***the reference Shell module architecture discussed in this chapter is not mandated by the libagents library*** in any way; instead, any alternative Shell module architecture can be used in conjunction with a libagents Core module, for as long as the Shell module provides a communication interface with the application's Core module which is compatible with the communication interface implemented by the Core module

Fig.12 below illustrates the top-level view of the reference Shell module architecture that will be discussed throughout this chapter:

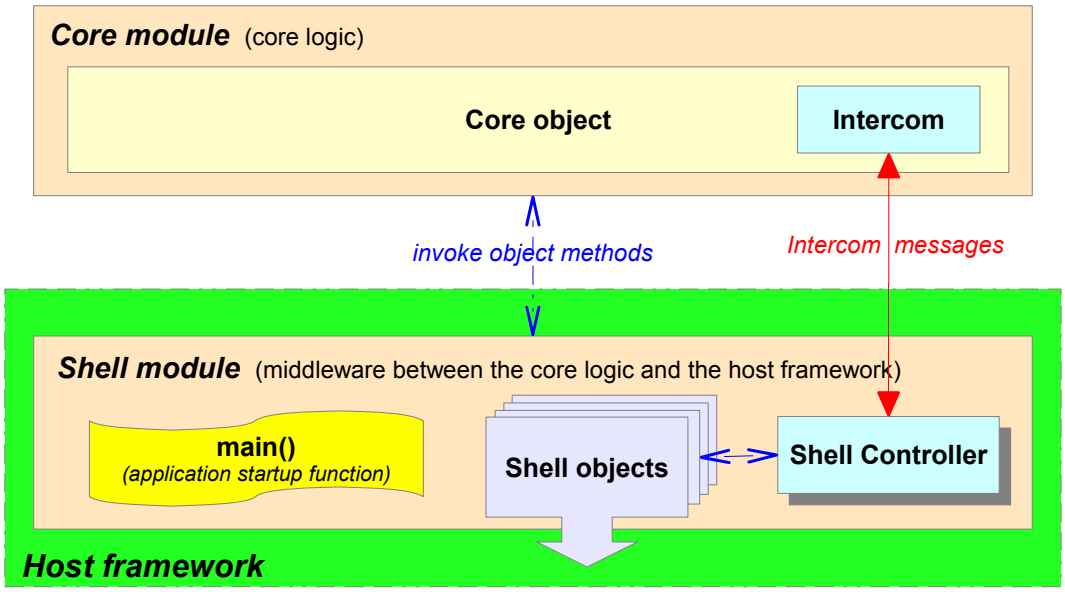

Fig.12: top-level view of the reference Shell module architecture

- the application's ***"startup function"***, illustrated as 'main()' in Fig.12 above: this is the function that is automatically executed at application startup, i.e. it is the equivalent of a console application's function "int main(int argc, char** argv)"
- the ***"Shell objects"***: each Shell object is *implemented "on top of" the application's host framework API,* and it provides the Core module with *a host framework-independent, object oriented API for a given environment-integration function required by the application* (e.g. network communications, file system access, system timers, GUI operations, etc)
  - for example, consider a Shell object which must provide a simple file system interface capable of reading and writing files on the host file system: in this case, said Shell object can be implemented as e.g. a 'FileSystemInterface' object which provides two methods 'ReadFile()' and 'WriteFile()' whose *implementation* will invoke the file system API of

the application's host framework; this is illustrated in Fig.13 below:

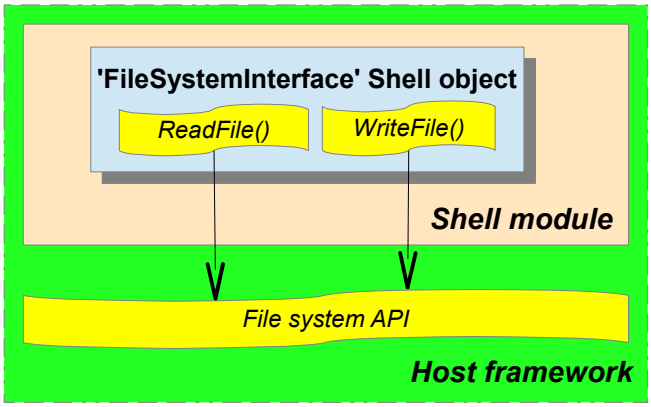

Fig.13: a 'FileSystemInterface' Shell object which provides a host framework-independent interface {ReadFile(), WriteFile()} to the host framework's native file system API

- the ***"Shell Controller object"***: this is a mandatory component of the reference Shell module architecture, whose role is to receive and execute the commands sent by the Core module to the Shell module (usually by invoking the APIs of the Shell objects); additionally, the Shell Controller object also relays all the messages that are sent by the Shell objects to the Core module

Fig.14 below illustrates a detailed view of the reference Shell module architecture illustrated in Fig.12 above, together with the interconnections between the application's Core and Shell modules:

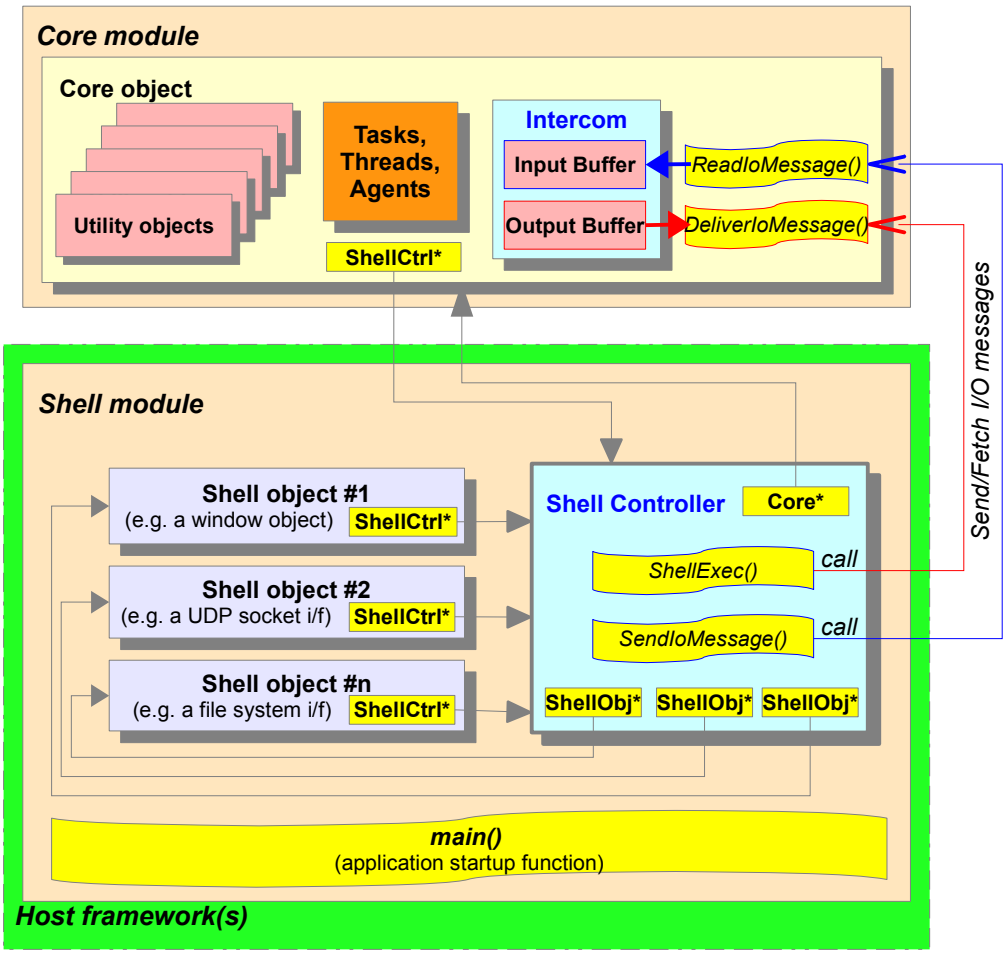

Fig.14: detailed view of the reference Shell module architecture, together with its interconnections with the application's Core module

The implementation details of the Shell module components illustrated in Fig.14 above are detailed in the following paragraphs.

The Shell objects

As it was previously described in the "Reference Shell module architecture" paragraph above, *the role of the Shell objects is to provide a libagents application with all the system integration functions that the application requires during its operation* (e.g. network communications, file system access, GUI, etc). In this context, Fig.15 below illustrates the ***generic architecture*** of the Shell objects that are part of the reference Shell module architecture illustrated in Fig.14:

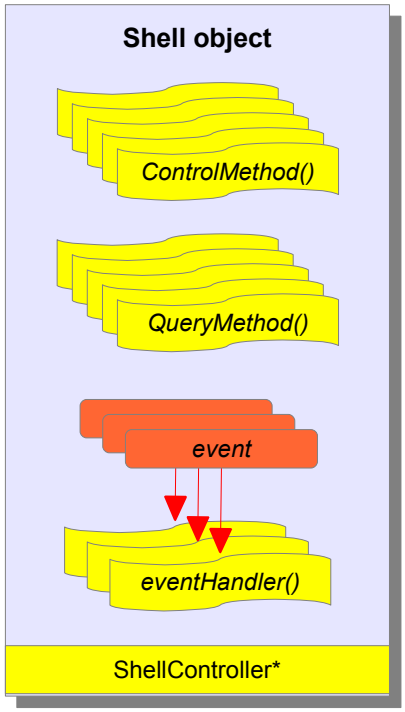

Fig.15: generic architecture of the reference Shell module's Shell objects

The Shell object's generic architecture as illustrated in Fig.15 above consists of:

- a Shell object must provide a ***host framework-independent API***, i.e. the functionality and signature of each Shell object method must depend solely on the functionalities provided by the Shell object, and not on any specifics of the application's host framework; specifically:
  - a Shell object *must* implement ***a set of "control methods"*** which implement the specific commands that the Shell object can execute (e.g. a UDPSocket Shell object must provide a control method for sending a datagram at a specified {address:port} with an optional TTL, a SystemTimer Shell object must provide control methods for starting/stopping the timer and for programming its "tick rate", etc)
  - a Shell object *may*, depending on its type, implement ***an optional set of "query methods"*** which can be used to read data from, and/or status information about, the Shell object (e.g. a FileSystemInterface Shell object will usually provide query methods for reading various status information about a file, a SystemTimer Shell object will usually provide a query method for reading the "current time" on the host system, etc)
  - a Shell object *may*, depending on its type, have to ***detect and process specific events/conditions*** that may occur in the application's operating environment (e.g. a UDPSocket Shell object must be capable of detecting and processing incoming datagrams, a SystemTimer Shell object should [usually] be capable of executing a specified snippet of code at a regular interval and/or at a specified time, etc): in such

cases, the Shell object must implement ***a dedicated "event handler" method for each event/condition that it must detect***, where said "event handler" method will be automatically executed each time the corresponding event/condition is detected by the Shell object (e.g. a 'UDPSocket' Shell object should implement an "onDataReceived()" event handler method which will be automatically executed each time a new datagram is received by the UDPSocket, a 'SystemTimer' Shell object should implement an "onTime()" method which will be automatically executed at a specified time, etc)

- ***the Shell objects must be all interconnected both amongst themselves, and with the application's Core object***: these interconnections are achieved indirectly via pointer cross-linking with the "Shell Controller" object (the "ptr" yellow boxes in Fig.14), and they are detailed in the following paragraph "The Shell Controller object"

➢ IMPORTANT: any Shell object method that may be concurrently invoked from multiple OS threads must have a thread-safe implementation

The Shell objects' architectural details that depend on the characteristics of the application's host framework are discussed in the "Host framework integration" paragraph later in this chapter.

The Shell Controller object

The Shell Controller object is *a mandatory Shell component* which must be implemented as part of the reference Shell module architecture illustrated in Fig.14, and its role is to act as *the "central coordinator" for all the activities that occur inside the Shell module*. In this context, Fig.16 below illustrates the ***generic architecture*** of a Shell Controller object that is part of the reference Shell module architecture:

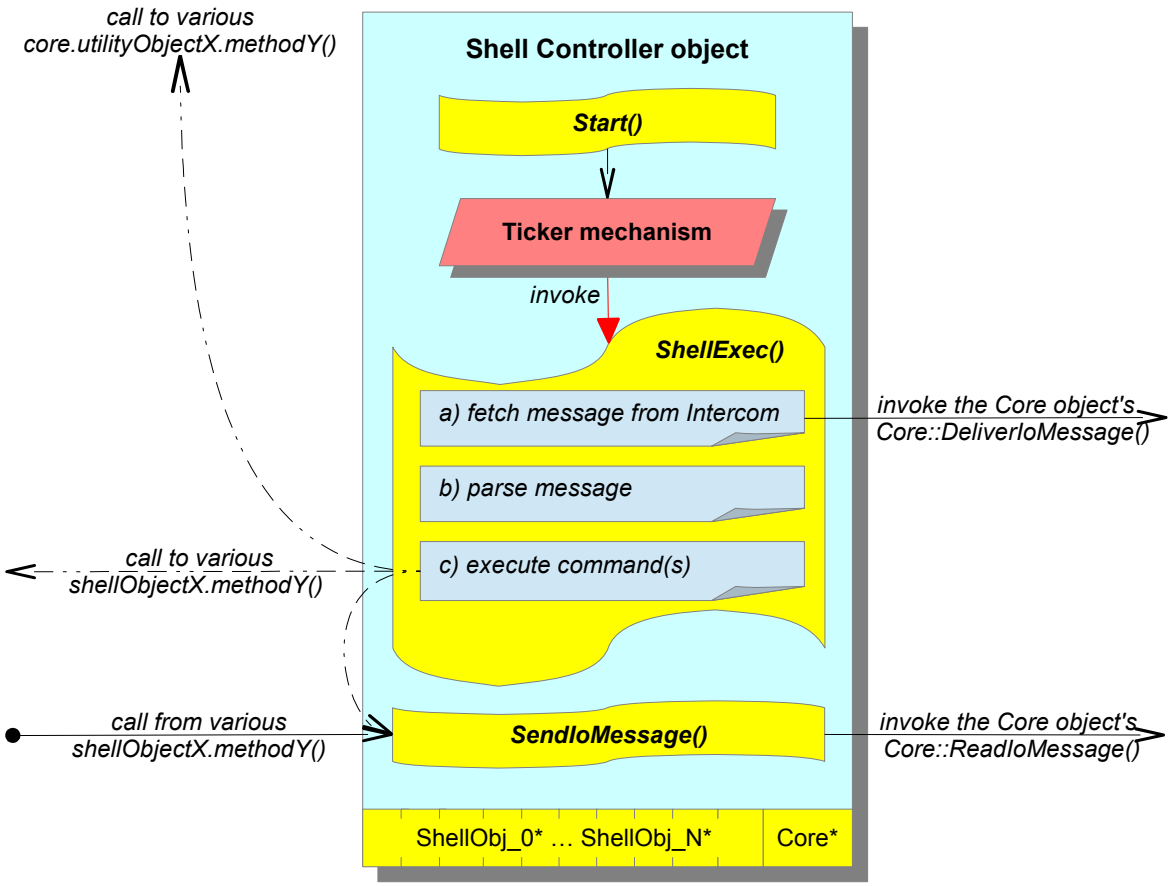

Fig.16: generic architecture of the reference Shell module's Shell Controller object

The Shell Controller's generic architecture as illustrated in Fig.16 above consist of:

- the Shell Controller object must provide a method that we will here-forth refer to as

"SendIoMessage()" (see Fig.14 and Fig.16 above), whose role is to act as *a global gateway for all the messages sent by the Shell objects to the Core module*, i.e. the 'SendIoMessage()' method must be invoked by all the Shell objects for *any and all* I/O messages that they need to send to the Core module.

- *Note:* the 'SendIoMessage()' method can immediately forward to the Core module the messages it receives from the Shell objects (by invoking the Core module's 'ReadIoMessage()' method), or, depending on the application requirements, it can parse the messages and perform specific actions before, or instead of, forwarding them to the Core module (e.g. the 'SendIoMessage()' method can make changes to a message before forwarding it to the Core module, or it can completely filter out the message, etc)
- IMPORTANT: ***if the Shell module has a multi-threaded implementation*** *(e.g. if the event handler methods of the Shell objects are executed by multiple OS threads), then the 'SendIoMessage()' method may potentially be invoked concurrently by the Shell objects from multiple OS threads; in this case,* ***the 'SendIoMessage()' method must have a thread-safe implementation***

- the Shell Controller object must implement a method that we will here-forth refer to as ***"ShellExec()"*** (see Fig.14 and Fig.16 above), whose role is to *fetch from the Core module Intercom's Output Buffer any pending messages* that have been sent by the Core module to the Shell module (this is achieved by invoking the Core module's 'DeliverIoMessage()' method) (a), and then, for each fetched message, it must *parse the message* (b) and *execute any command(s) specified in the message contents* (c)
  - *Note:* the 'ShellExec()' method may invoke, during its execution, the 'SendIoMessage()' method in order to send messages to the Core module, and it may also invoke various methods of various Shell objects and/or of various Core module Utility objects (see Fig.16 above)
- in order to continuously monitor the Core module Intercom's Output Buffer for any pending messages, the Shell Controller object must implement a ***"Ticker mechanism"*** (see Fig.16 above) which must ***regularly invoke the 'ShellExec()' method,*** such that the 'ShellExec()' method can fetch and parse each message sent from the Core module (see the 'ShellExec()' method description above). The rate at which the 'Ticker' mechanism invokes the 'ShellExec()' method must be relatively high in order to minimize the delay between the moment when a message has been placed in the Core module Intercom's Output Buffer and the moment when said message is extracted by the 'ShellExec()' method (e.g. for a "tick rate" of 1000 "ticks"/sec the messages can be extracted from the Intercom's Output Buffer at a maximum rate of 1000 times/sec).
  - IMPORTANT: in order to speed up the processing of commands sent from the Core module, ***it is recommended that the 'ShellExec()' method extracts (and processes) multiple messages at each "tick"***, thus avoiding the [Ticker-induced] one "tick"minimum delay between fetching any two consecutive messages; however, ***the 'ShellExec()' method should not fetch (and process) too many messages at a time*** *because the OS thread that executes the 'ShellExec()' method will be "blocked" until the 'ShellExec()' method finishes its execution (i.e. until it finishes processing all the messages it extracts "in one tick" from the Core Intercom's Output Buffer)*, thus potentially causing the application Shell to become unresponsive (a.k.a. to "freeze") for excessively long periods of time

  The implementation details of the Shell Controller's 'Ticker' mechanism depend on the characteristics of the application's host framework; this is discussed in the "Host framework integration" paragraph later in this chapter
- the Shell Controller object must implement a ***"Start()"*** method (see Fig.16 above) whose

role is to initialize the Shell Controller object and start the Shell Controller's Ticker mechanism; after the Shell Controller's 'Start()' method has been invoked, the Shell module is ready to receive and process commands sent by the Core module (see the "ShellExec()" method above), and it is ready for relaying event notifications sent by the Shell objects to the Core module (see the "SendIoMessage()" method above)

- finally, the Shell Controller object must be ***cross-linked with all the other Shell objects in the Shell module, and also with the Core object***; specifically:
  - the Shell Controller object *must* implement a set of pointers which each point to one of the Shell objects (the 'ptr' yellow boxes inside the Shell Controller object in Fig.14 and Fig.16 above), and each Shell object *must* implement a pointer back to the Shell Controller object (the 'ptr' yellow boxed inside each Shell object in Fig.14): in this way, any Shell object can [indirectly] invoke any method of any other Shell object via the corresponding indirection pointer found in the Shell Controller object, thus allowing any two Shell objects to communicate with each other
  - the Shell Controller object *must* implement a pointer to the Core application object in order to have access to the Core module's "ReadIoMessage()" and "DeliverIoMessage()" methods and/or to the Core module Utility Objects' methods (the 'Core ptr' yellow box inside the Shell Controller object in Fig.14 and Fig.16 above); additionally, the Core object *should* implement a pointer back to the Shell Controller object in order to allow [indirect] access to the various Shell objects via their corresponding pointers contained in the Shell Controller object (the 'Shell ptr' yellow box inside the Core object in Fig.14)

The Shell Controller object's architectural details that depend on the characteristics of the application's host framework are discussed in the "Host framework integration" paragraph later in this chapter.

The application startup function

As it was previously described in the "Reference Shell module architecture" paragraph earlier in this chapter, the startup function of a libagents application (illustrated as 'main()' in Fig.14) is *the application's initialization function whose execution is automatically launched at application startup*, and it is executed by a dedicated OS thread which is allocated to the application by the operating system when the application is started; we will here-forth refer to the OS thread that executes the application's startup function as ***the application's "main thread"***.

In terms of functionality, ***the startup function of a libagents application that uses the reference Shell module architecture must create the startup configuration of all the application's top-level objects***, i.e. the Shell module's Shell objects (e.g. windows, network sockets, etc) and Shell Controller object, and the Core module's top-level Core object.

- *Note:* after the startup function has created (and initialized) a top-level object, *it is said top-level object's responsibility to further create (and initialize) any sub-component object(s) that it may contain* (see also the "Initializing the Core module" paragraph earlier in this document)

As it is the case with any application that is built "on top of" a host framework, the code template (including the signature) of the startup function of a libagents application is defined by the host framework "upon which" the application is built, and it differs widely from one host framework to another; however, regardless of the host framework-specific details, ***the startup function of a libagents application that uses the reference Shell module architecture will always have to perform the generic sequence of steps illustrated in Fig.17 below:***

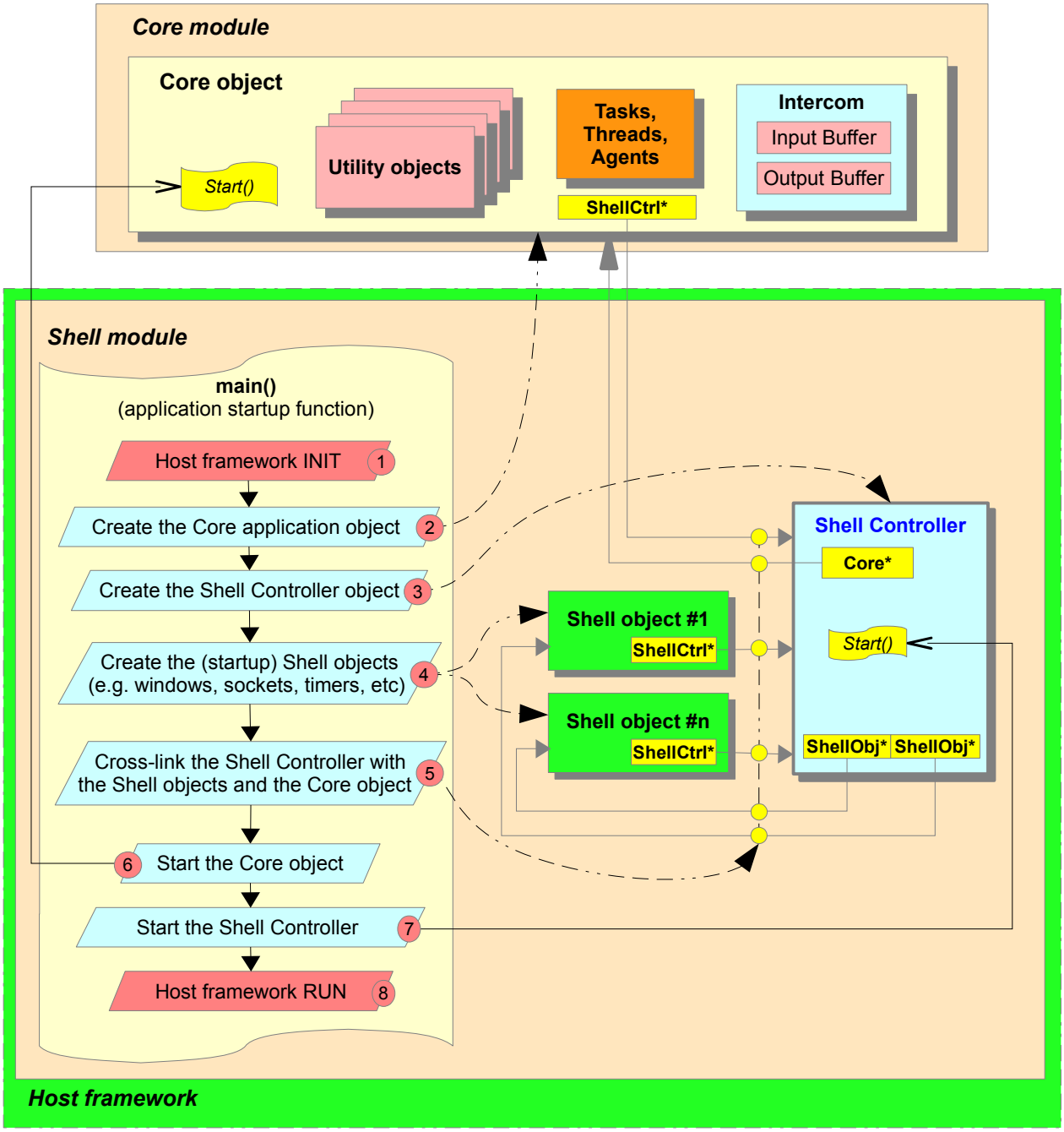

Fig.17: sequence of steps that must be performed by the startup function of a libagents application

Following is a detailed presentation of the sequence of steps illustrated in Fig.17 above:

1. this step is required by some, but not all, host frameworks in order to initialize the application and/or the host framework
   - ✗ for example, the "Host framework INIT" step in a Qt 5-based application will consist of:

     ```
     QApplication qApp(argc, argv); // create the QApplication object & pass params
     ```

2. this step creates the application's Core object: this is achieved by invoking the Core object's constructor, e.g. "MyApplicationCore *myApplicationCore=new MyApplicationCore", where 'MyApplicationCore' is the class name of the application's Core object, which must be derived from the Core base class provided by the libagents library; this will create the application Core object (see "The Core object" and "Initializing the Core module" paragraphs earlier in this document, step (A) in Fig.7)
   - *Note:* it is *not* the responsibility of the startup function to create (and initialize) the sub-component objects of the Core object (i.e. the Task, Thread, Agent, and Core Utility

objects); instead, this functionality must be implemented by the Core object itself (see the "Initializing the Core module" paragraph earlier in this document)

3. after the Core application object has been created (together with any sub-component objects that it may contain), the next step is to create the application's Shell Controller object by invoking e.g. "MyShellController *myShellController=new MyShellController(myApplicationCore)", were 'MyShellController' is the class name of the application's Shell Controller object (see "The Shell Controller object" paragraph earlier in this chapter), and 'myApplicationCore' is a pointer to the application Core object
4. this step creates [the startup configuration of] the Shell objects (see "The Shell objects" paragraph earlier in this chapter)
5. after the Shell objects [in their startup configuration] and the Shell Controller object have been created, the next initialization step is to cross-link the Shell Controller object with each of the Shell objects and with the application's Core object: specifically, the Shell Controller object includes a set of pointers which have to be initialized such that they each point to one of the Shell objects, and each Shell object includes a pointer which has to be initialized to point back to the Shell Controller object; additionally, the Shell Controller object also contains a pointer that has to be initialized to point at the application Core object, and the application Core object contains a pointer that has to be initialized to point at the Shell Controller object (see "The Shell Controller object" paragraph earlier in this chapter)
6. at this point the startup configuration of the libagents application has been completely set up, and the startup function must now *start the application's Core object*: this is achieved by invoking the Core object's 'Start()' method (see the "Initializing the Core object" paragraph earlier in this document, step (B) in Fig.7)
7. now the startup function has to initialize and start the Shell Controller object: this is achieved by invoking the Shell Controller object's 'Start()' method (see "The Shell Controller object" paragraph earlier in this chapter)
8. this step is required by some, but not all, host frameworks as the last step of the application's startup function (e.g. in the case of event-driven host frameworks, this step starts the execution of the application's event loop)
   - for example, the "Host framework RUN" step in a Qt 5-based application will consist of:

     ```
     qApp.exec(); // start the QApplication object's event loop
     ```

An example application that is built "on top of" the Qt framework and which illustrates the sequence of steps that must be performed by a libagents application's startup function is available in the libagents-2.0.x distribution package, inside the "libagents-examples/libagents-example-stopwatch-qt" folder (see the function "int main(int argc, char *argv[])" in file "main.cpp").

Multi-threading in the reference Shell module

As previously described in "The Shell Controller object" paragraph earlier in this chapter, a Shell object that is part of a reference Shell module can send an I/O message to the Core module by invoking the Core module's "ReadIoMessage()" method indirectly, via the Shell Controller object's "SendIoMessage()" relay method (see Fig.14), such that the Core module's "ReadIoMessage()" method will be executed by the same OS thread that executes the Shell object method which sends the message; in this context, and given the fact that the Core module's "ReadIoMessage()" method is allowed to be invoked from any OS thread without restrictions (see the description of the Intercom object in the "I/O messages" paragraph earlier in this document), ***the libagents library imposes no restrictions regarding which OS thread(s) are executing the Shell objects' methods that are sending I/O messages to the Core module***.

Similarly, as previously described in "The Shell Controller object" paragraph earlier in this chapter, the messages sent from the Core module are fetched inside a reference Shell module by the Shell Controller's "ShellExec()" method (which, in turn, invokes the Core module's "DeliverIoMessage()" method, see Fig.14); in this context, and given the fact that the Core module's "DeliverIoMessage()" method is allowed to be invoked from any OS thread without restrictions (see the description of the Intercom object in the "I/O messages" paragraph earlier in this document), ***the libagents library imposes no restrictions regarding which OS thread is executing the Shell Controller's "ShellExec()" method that extracts the I/O messages from the Core object Intercom***.

In conclusion, ***the reference Shell module architecture presented in the "Reference Shell module architecture" paragraph earlier in this chapter does not impose, in and by itself, any restrictions regarding which OS threads are executing the Shell objects' methods that send, and respectively extract and process, I/O messages to/from the application's Core module.***

- IMPORTANT: various multi-threading restrictions may be imposed on the reference Shell module objects *by the characteristics of the application's underlying host framework*; this is discussed in the "Host framework integration" paragraph below, the "Thread-safety in an event-driven reference Shell module" and "Thread-safety in a procedural reference Shell module" sub-paragraphs

Host framework integration

As previously explained in "The Shell module" paragraph earlier in this document, the Shell module of a libagents application serves the role of interfacing the application core logic with the application's operating environment, i.e. all the system integration functions required by a libagents application must be provided by the application's Shell module. In this context, this chapter describes the detailed architecture of the Shell objects and of the Shell Controller object of a reference Shell module architecture (see Fig.14), such that the internal architecture of said objects makes use of the specific functionalities provided by the target host framework *all while complying with the generic architectures presented in* "The Shell objects" *and* "The Shell Controller object" *paragraphs above.*

Based on how the characteristics of a libagents application's host framework impact the implementation details of a reference Shell module's components, we shall consider a simple host framework taxonomy consisting of two categories, namely *"**Event-driven host frameworks**"* and *"**Procedural host frameworks**";* the essential characteristics which differentiate these two classes of host frameworks, together with the way said characteristics impact the architectural details of the Shell objects and of the Shell Controller object of a reference Shell module architecture, are discussed in the following paragraphs.

Event-driven host frameworks

The essential characteristic of an event-driven host framework is that the services it provides to a hosted application come in the form of a collection of *predefined "host framework-native objects"* which each implement a specific set of *control and query methods* that can be invoked by the hosted application, and which are each capable of *detecting and handling a specific set of "events"* that may occur during the execution of the application.

With respect to the event detection mechanism, each event-driven host framework-native object can be programmed to execute *a specified "event handler method" whenever a specific event occurs* during application execution; then, whenever an event will occur during application execution, *the internal mechanisms of the host framework* will automatically execute the event handler method associated with said event.

- for example, the Qt framework is an event-driven host framework which provides a 'QTimer' object class which can be programmed (via its setup methods) to automatically execute an event handler method at a specified time, or after a specified delay, or at a specified interval, etc (e.g. an object 'myTimer' that implements the 'QTimer' class can be programmed to periodically execute an event handler method which checks if a specific file has been modified on disk, or it can be programmed to automatically shut down the application at a specific point in time, etc)

In terms of internal implementation of the event handling mechanism, any application that is built "on top of" a typical event-driven host framework is *automatically set up by the host framework to continuously run an internal code loop,* commonly referred to as *the host framework's "event loop"*, and said event loop continuously monitors all the conditions/events for which the application has programmed an event handler method [of a host framework native object]; then, whenever one of the monitored conditions/events occurs, the event loop *automatically* invokes (read: calls) the event handler method associated with said condition/event.

The top-level architecture of an event-driven host framework (as described above), and the way it integrates with a hosted application, are illustrated in Fig.18 below:

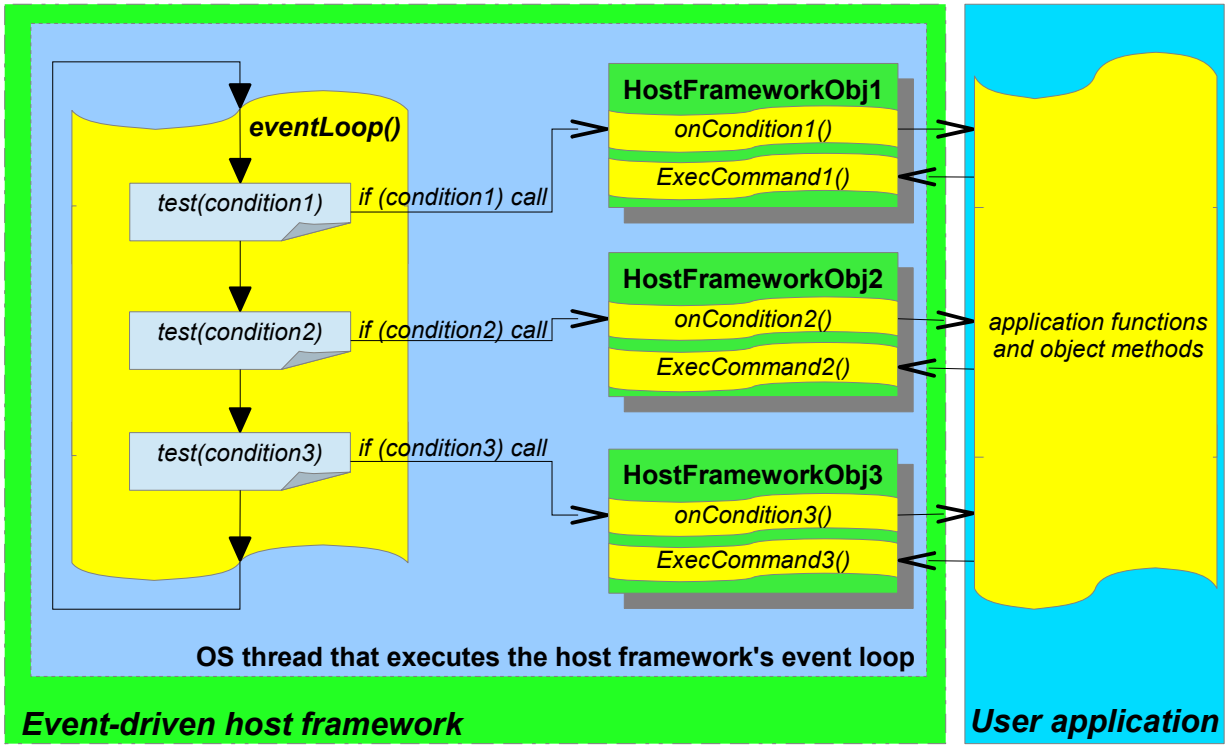

Fig.18: top-level architecture of an application built "on top of" an event-driven host framework

As it can be seen in Fig.18 above, the execution of a user application that is built "on top of" an event-driven host framework essentially consists of:

1. the application's host framework detects various events/conditions that occur in the application's operating environment, then
2. the host framework's event loop [automatically] invokes the event handler methods of the host framework-native objects which have been programmed to respond to the events/conditions that have occurred, then
3. the event handler methods notify the user application about the events/conditions that have occurred by invoking the required application functions and/or object methods, and
4. finally, the user application's functions and/or object methods (may) react upon the application's operating environment by sending commands to the host framework-native

objects (by invoking their control methods)

In terms of execution threads, the OS thread that executes the event loop of an event-driven application is commonly referred to as *the **"event loop thread",*** and *all the host framework objects' event handler methods are executed by the event loop thread* (see Fig.18 above). Consequently, *all the user application's methods which are invoked by the host framework objects' event handlers are also executed by the event loop thread,* and, furthermore, any *host framework-native objects' control methods which are called back by the user application methods are also executed by the event loop thread*. In conclusion, ***an event-driven application which does not create OS threads will have all its methods, may they be methods of the user application itself or methods of the host framework's objects, executed exclusively by the host framework's event loop thread.***

- *Note 1:* some host frameworks implement *multiple event loops which all run in parallel, each in a separate OS thread* (e.g. one event loop may monitor GUI events in one OS thread, another event loop may monitor time-related events in another OS thread, and yet another event loop may monitor all the other kinds of events in yet another OS thread); in this scenario, *multiple event handler methods [of multiple host framework-native objects] may be executed in parallel, in separate OS threads*, if said event handler methods are invoked by different event loops (and thus from different OS threads)
- *Note 2:* some host frameworks do not invoke the event handler associated with an event/condition *directly* from (one of) their event loop(s); instead, when an event loop detects a condition/event, *a new OS thread is started*, then the required event handler method is executed *in the newly started OS thread*, and finally, when the event handler completes execution, the OS thread that executed the event handler is destroyed
- *Note 3:* some host frameworks allow more than one event handler to be associated with a given event/condition: in this case, when an event/condition occurs, all the event handlers associated with said event/condition will be executed either sequentially, in an application-defined order, by a single OS thread if the host framework implements a single event loop, or they may be executed in parallel by multiple OS threads if the host framework implements [some form of] multi-thread event handlers, or a combination thereof

Event-driven Shell objects

We will here-forth use the term *"Event-driven Shell objects"* to designate the Shell objects of a reference Shell module architecture which is built "on top of" an event-driven host framework (see the "Reference Shell module architecture", "The Shell objects" and "Event-driven host frameworks" paragraphs earlier in this chapter).

As previously explained in "The Shell objects" paragraph earlier in this chapter, the reference Shell module architecture specifies a *generic architecture* for the Shell objects (see Fig.15), while the implementation details of the Shell objects depend on the characteristics of the application's host framework; in this context, the following apply to the *event-driven* Shell objects that are part of a reference Shell module architecture

- ➔ ***depending on the restrictions imposed by the host framework,*** an event-driven Shell object can be implemented either as a user-defined ***wrapper class*** which contains (one or more) host framework-native objects and/or invokes functions of the host framework's native API, or as a user-defined class which is ***derived*** from (one or more) host framework-native objects

In terms of execution threads, ***the control methods and query methods of an event-driven Shell object are executed by the OS thread that invokes said methods, while the event handler methods of an event-driven Shell object are executed by the host framework's event loop thread***.

- ✗ for example, the Qt framework is an event-driven host framework which provides a

'QUdpSocket' socket object (read: base class) which implements a 'writeDatagram()' method for sending a datagram at a specified {address:port}, and, additionally, a 'QUdpSocket' object generates an event each time it receives a new datagram. Based on the QUdpSocket class (provided by the Qt framework), a user-defined event-driven Shell object 'ShellUdpSocket' can be defined as a class which contains [as one of its elements] a 'QUdpSocket' object, and *the Qt framework allows* a method of the [user-defined] 'ShellUdpSocket' class, e.g. 'ShellUdpSocket::onDataReceived()', to be *declared as the event handler method of the 'QUdpSocket' native object* (i.e. the 'ShellUdpSocket::onDataReceived()' method will be automatically invoked *by the Qt framework's event loop* each time a new datagram is received by the 'QUdpSocket' native object); the 'ShellUdpSocket' object described above is illustrated in Fig.19 below:

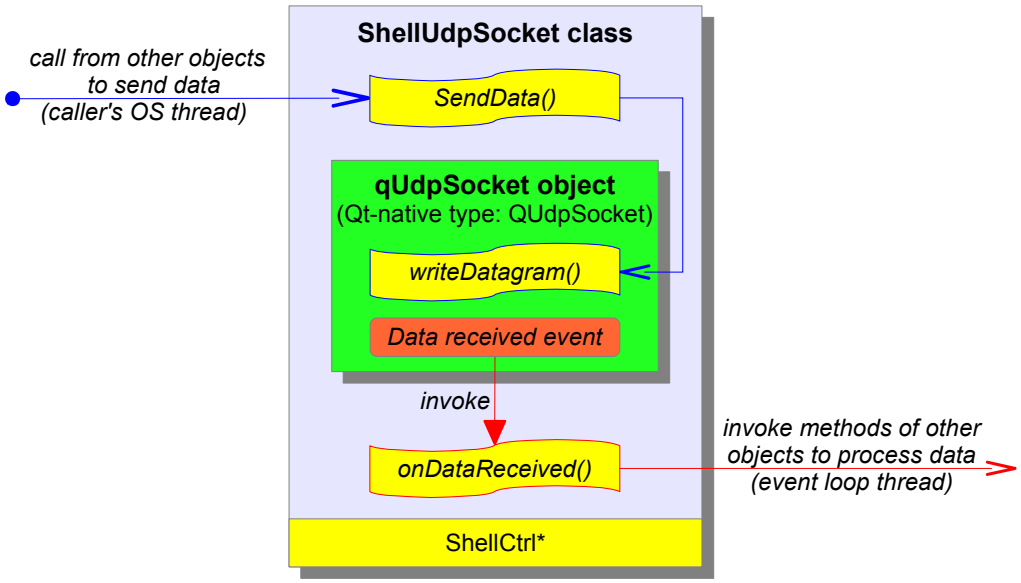

Fig.19: a 'ShellUdpSocket' Shell object implemented as a wrapper class over a Qt-native 'QUdpSocket' object

- *Note:* the qUdpSocket object's 'writeDatagram()' method is executed by the OS thread that invokes the 'ShellUdpSocket::SendData()' method, while the 'ShellUdpSocket::onDataReceived()' method is executed by the Qt framework's event loop thread [which monitors the incoming UDP network traffic]

Event-driven Shell Controller object

We will here-forth use the term *"Event-driven Shell Controller object"* to designate the Shell Controller object of a reference Shell module architecture which is built "on top of" an event-driven host framework (see the "Reference Shell module architecture", "The Shell Controller object" and "Event-driven host frameworks" paragraphs earlier in this chapter).

As previously explained in "The Shell Controller object" paragraph earlier in this chapter, the reference Shell module architecture specifies a *generic architecture* for the Shell Controller object (see Fig.16), while the implementation details of the Shell Controller object depend on the characteristics of the application's host framework; in this context, the following apply to the *event-driven* Shell Controller object that is part of a reference Shell module architecture:

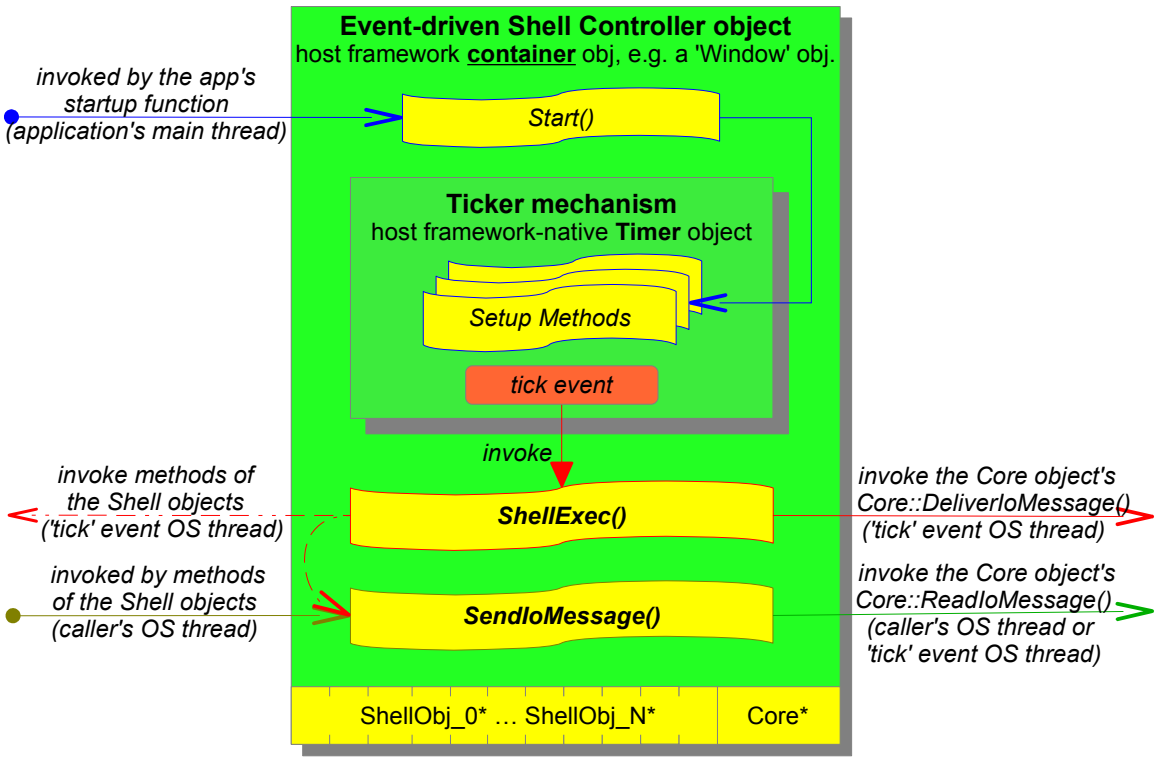

Fig.20: architecture of an event-driven Shell Controller object

➔ ***the "Ticker mechanism" of an event-driven Shell Controller object must be implemented as a host framework-native Timer object*** (e.g. a 'QTimer' object in the Qt framework, or a 'wxTimer' object in the wxWidgets framework, or a 'TTimer' object in the C++ Builder framework, etc), and it must be instantiated as ***a component of the Shell Controller object*** by the Shell Controller's constructor or by its 'Start()' method

- *Note 1:* depending on the restrictions imposed by each particular host framework, the Shell Controller's Timer object can either be *instantiated as a component* (read: "embedded") inside the Shell Controller class, or it can be instantiated on the heap (via the 'new' operator) by the Shell Controller's constructor and then *referenced via a pointer* contained in the Shell Controller class; the latter alternative is supported by most GUI-oriented host frameworks
- *Note 2:* some event-driven host frameworks no not provide a native Timer *object*, and, instead, they provide (one or more) *timer setup function(s)* which can be used to specify a *"timeout event handler"* function which will be executed *by the host framework's event loop* at a specified time, or after a specified delay, etc (e.g. the GTK framework does not provide a native Timer object, and, instead, it provides the 'gtk_timeout_add()' function which can be used to program the execution of a timeout event handler function at a specified time): for such host frameworks, the Shell Controller's "Ticker mechanism" must be implemented as a custom user-defined object which will make use of the host framework-native timer setup functions

➔ ***the event-driven Shell Controller object must be a valid container for a host framework-native Timer object, or for a pointer to a host framework-native Timer object*** (this condition is necessary because the Shell Controller's "Ticker mechanism" is implemented as *a host framework-native Timer object which is instantiated as a component of the Shell Controller object,* see above)

- *Note 1:* most GUI-oriented event-driven host frameworks which provide a native Timer object (e.g. a 'QTimer' object in the Qt framework, or a 'wxTimer' object in the wxWidgets framework, or a 'TTimer' object in the C++ Builder framework, etc) allow

their native “Window” objects (e.g. a 'QWidget' object in the Qt framework, or a 'wxFrame' in the wxWidgets framework, or a 'TForm' object in the C++ Builder framework, etc) to contain a native Timer object or a pointer to a native Timer object, such that the main window of a GUI application can serve as the application's Shell Controller object on most GUI-oriented event-driven frameworks; alternatively, a dedicated hidden “Window” object can also be used for implementing the Shell Controller object

- *Note 2:* some host frameworks provide means for declaring *any user-defined class to be a valid container for host framework-native objects* (e.g. the Qt framework allows any user-defined class which is derived from 'QObject' and which includes a 'Q_OBJECT' macro to be a container for Qt-native objects); on such frameworks, the Shell Controller object can be implemented as a user-defined class which is properly declared as a container for host framework-native objects (the means by which this declaration is achieved is host framework-specific)

➔ ***the 'Start()' method of an event-driven Shell Controller object must control all the functional parameters of the Timer object***, i.e. it must be able to [at least] start, stop, and set the tick rate of the Timer object, based on the call arguments

In terms of execution threads, ***the 'Start()' method of an event-driven Shell Controller object is executed by the application's main thread*** (because the 'Start()' method is invoked by the application's startup function, see “The application startup function” paragraph earlier in this chapter), ***the 'SendIoMessage()' method is executed by the OS thread that invokes the method*** (which may be any OS thread, see “The Shell Controller object” paragraph earlier in this chapter), and ***the 'ShellExec()' method, together with any other object methods and/or functions that it may invoke during its execution, are executed by the host framework's event loop thread*** (because the “Ticker mechanism” is implemented as a host framework-native Timer object, and thus its 'tick' events are generated by the host framework's event loop thread).

Thread-safety in an event-driven reference Shell module

An important characteristic of most event-driven host frameworks that are actively used and/or maintained at the time of writing this document (e.g. Qt, GTK, wxWidgets, C++ Builder, etc) is that *by default they implement a single event loop,* and thus all the event handler methods of all the host framework-native objects are executed by default by a single OS thread, namely by the host framework's event loop thread (see the “Event-driven host frameworks” paragraph earlier in this chapter); additionally, many event-driven host frameworks which implement a single event loop *mandate that their native objects' methods are invoked exclusively from the framework's event loop thread.*

Given the above, the following conditions are ***sufficient*** for a reference Shell module implemented “on top of” an event-driven host framework to be single-threaded, and thus compatible with a very broad range of event-driven host frameworks:

1. ***the application's host framework implements a single event loop*** (and thus a single event loop OS thread, see the “Event-driven host frameworks” paragraph earlier in this chapter)
2. ***the Shell objects and the Shell Controller object are implemented strictly as described in the “Event-driven Shell objects” and “Event-driven Shell Controller object” paragraphs above***
3. ***any and all of the Shell objects' methods are invoked exclusively by other Shell objects' methods, and/or by the Shell Controller's 'ShellExec()' method*** (this also implies that ***none of the Core module objects should invoke any method of any Shell object***)

If the three conditions above are obeyed, then any and all activities (read: call chains) inside the

reference Shell module will always be triggered *exclusively* by a host framework event, and thus ***any and all of the Shell objects' methods will always be executed exclusively by the host framework's event loop thread.***

Procedural host frameworks

The essential characteristic of a procedural host framework is that the services it provides to a hosted application come in the form of a collection of *predefined "host framework-native API functions" and/or "host framework-native objects"*, where said host framework-native API functions and the methods of the host framework-native objects can be invoked by the hosted application. In other words, *a procedural host framework can only execute commands* in response to having its native API functions and/or native objects' methods invoked, but *it cannot monitor, nor process, any events/conditions* that may occur during application execution.

The top-level architecture of a procedural host framework (as described above), and the way it integrates with a hosted application, are illustrated in Fig.21 below:

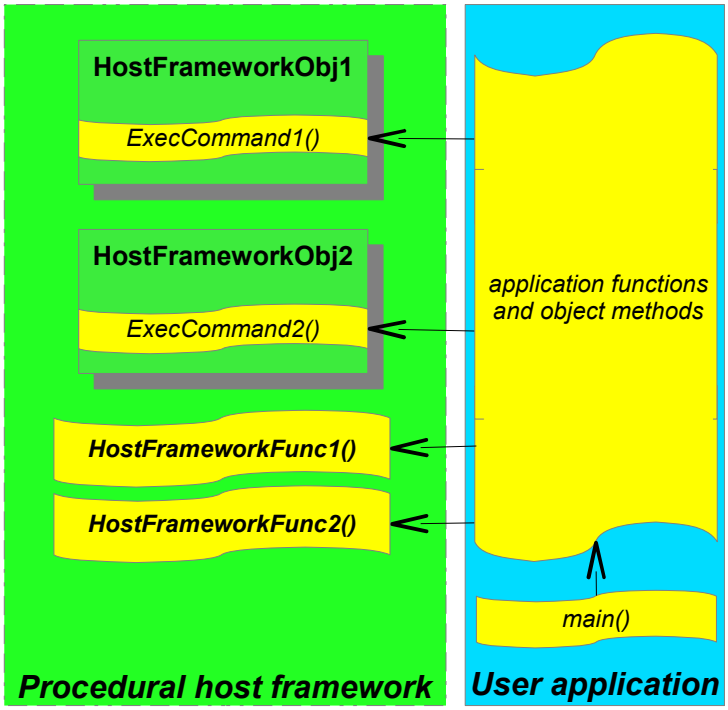

Fig.21: top-level architecture of an application built "on top of" a procedural host framework

As it can be seen in Fig.21 above, the execution of a user application that is built "on top of" a procedural host framework essentially consists starting up the application by running its startup function 'main()', and then the 'main()' function invokes various application functions and methods in order to complete its task; in turn, the application functions and methods can then invoke the methods of the host framework-native objects and/or the host framework's API functions whenever the application needs to interact with its operating environment (e.g. read/print a string from/to the application console, send/receive data to/from a network socket, etc).

In terms of execution threads, the OS thread that executes the user application's startup function (illustrated as "main()" in Fig.21 above) is commonly referred to as the application's ***"main thread"***, and ***an application that is built "on top of" a procedural host framework and which does not create OS threads will have all its methods, may they be methods of the user application itself or methods of the host framework's objects and/or API, executed exclusively by the application's main thread.***

Procedural Shell objects

We will here-forth use the term *"Procedural Shell objects"* to designate the Shell objects of a reference Shell module architecture which is built "on top of" a procedural host framework (see the "Reference Shell module architecture", "The Shell objects" and "Procedural host frameworks" paragraphs earlier in this chapter).

As previously explained in "The Shell objects" paragraph earlier in this chapter, the reference Shell module architecture specifies a *generic architecture* for the Shell objects (see Fig.15), while the implementation details of the Shell objects depend on the characteristics of the application's host framework; in this context, the following apply to the *procedural* Shell objects that are part of a reference Shell module architecture:

- ***depending on the restrictions imposed by the host framework,*** a procedural Shell object can be implemented either as a user-defined ***wrapper class*** which contains (one or more) host framework-native objects and/or invokes functions of the host framework's native API, or as a user-defined class which is ***derived*** from (one or more) host framework-native objects
- ***each procedural Shell object must implement its own event/condition monitoring mechanism*** which must be able to detect, and respond to, the specific set of events/conditions that the Shell object must monitor, if any (this condition is necessary because the procedural host frameworks do not provide a mechanism of their own for detecting, and responding to, the events/conditions that may occur in the application's operating environment - see the "Procedural host frameworks" paragraph earlier in this chapter); specifically:
  - ***all the procedural Shell objects contained in a reference Shell module architecture must be derived from a common base class which must declare a protected virtual method 'monitor()'***
    - *Note 1:* in order to ease the implementation of the procedural Shell objects, the libagents API provides an "empty" base class 'ShellObject' which contains exclusively the declaration of the protected virtual method "virtual void monitor(void) {return;}"; in this way, all the procedural Shell objects can be easily declared as derived classes from the 'ShellObject' base class, and they can each implement their own 'monitor()' method according to their required functionalities
    - *Note 2:* if a procedural Shell object is obtained by derivation from (one or more) host framework-native object(s), then said procedural Shell object must be declared using multiple derivation such that it is also derived from the procedural Shell objects' common base class which declares the 'monitor()' method
  - ***the 'monitor()' method of each procedural Shell object must poll the application's operating environment each time it is invoked, and check for the occurrence of any event/condition that the Shell object must detect (if any); then, whenever a given event/condition is detected, the 'monitor()' method must invoke the appropriate event handler method of the Shell object*** (i.e. the event handler method associated with the detected event/condition)
    - IMPORTANT: the procedural Shell objects' 'monitor()' method must be regularly invoked, at a relatively high rate, by an external "ticker mechanism", such that the events/conditions that it must detect can be [quasi-]continuously monitored by the Shell object. A "ticker mechanism" tick interval of 1x...5x the OS tick is recommendable for most applications (this translates into 1...5ms tick rate for most operating systems)
    - *Note:* as it can be seen from the description above, a procedural Shell object's

'monitor()' method, together with the external “ticker mechanism” which regularly invokes the 'monitor()' method, act as a substitute for (the lack of) a host framework-native event loop, thus enabling the procedural Shell objects to *react* to environmental events

In terms of execution threads, ***the control methods and the query methods of a procedural Shell object are executed by the OS thread that invokes said methods, while the Shell object's event handler methods are executed by the OS thread that executes the Shell object's 'monitor()' method***

- for example, consider a “ShellTimer” procedural Shell object which must provide a “GetTime()” method which returns the current system time, a “Reset()” method which resets the ShellTimer, and, additionally, *the ShellTimer object must be capable of executing an “onTime()” event handler method at a specified set of times* as programmed via two methods “AddTime()” and “DeleteTime()”; the internal architecture of such a ShellTimer procedural Shell object is illustrated in Fig.22 below:

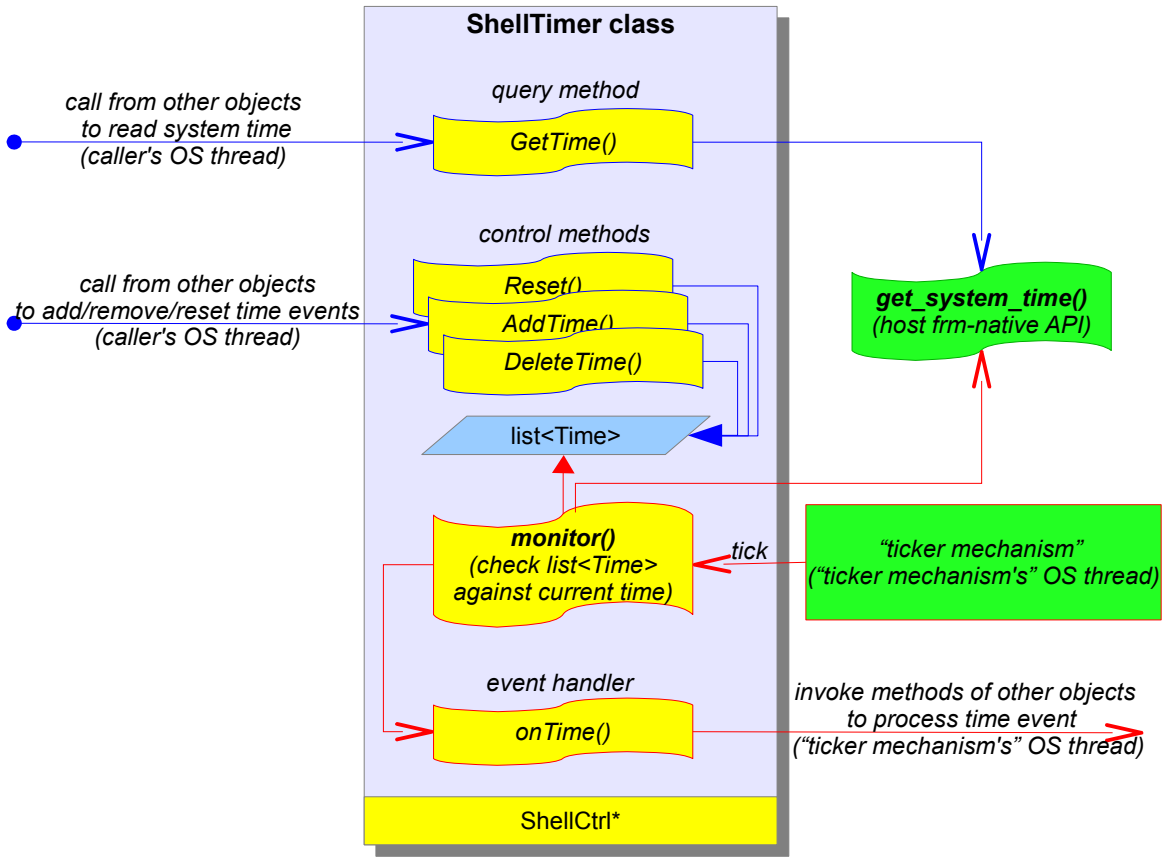

Fig.22: a 'ShellTimer' Shell object implemented over a procedural host framework API must use an external “ticker mechanism” for condition polling

- *Note:* the host framework's native API functions (i.e. 'get_system_time()' in Fig.22 above) are executed by the OS thread that invokes them (i.e. by the caller of the Shell object's 'GetTime()' query method, and by the external ”ticker mechanism” which invokes the 'monitor()' method in Fig.22 above)

Procedural Shell Controller object

We will here-forth use the term *“Procedural Shell Controller object”* to designate the Shell Controller object of a reference Shell module architecture which is built “on top of” a procedural host framework (see the “Reference Shell module architecture”, “The Shell Controller object” and “Procedural host frameworks” paragraphs earlier in this chapter).

As previously explained in "The Shell Controller object" paragraph earlier in this chapter, the reference Shell module architecture specifies a *generic architecture* for the Shell Controller object (see Fig.16), while the implementation details of the Shell Controller object depend on the characteristics of the application's host framework; in this context, the following apply to the *procedural* Shell Controller object that is part of a reference Shell module architecture:

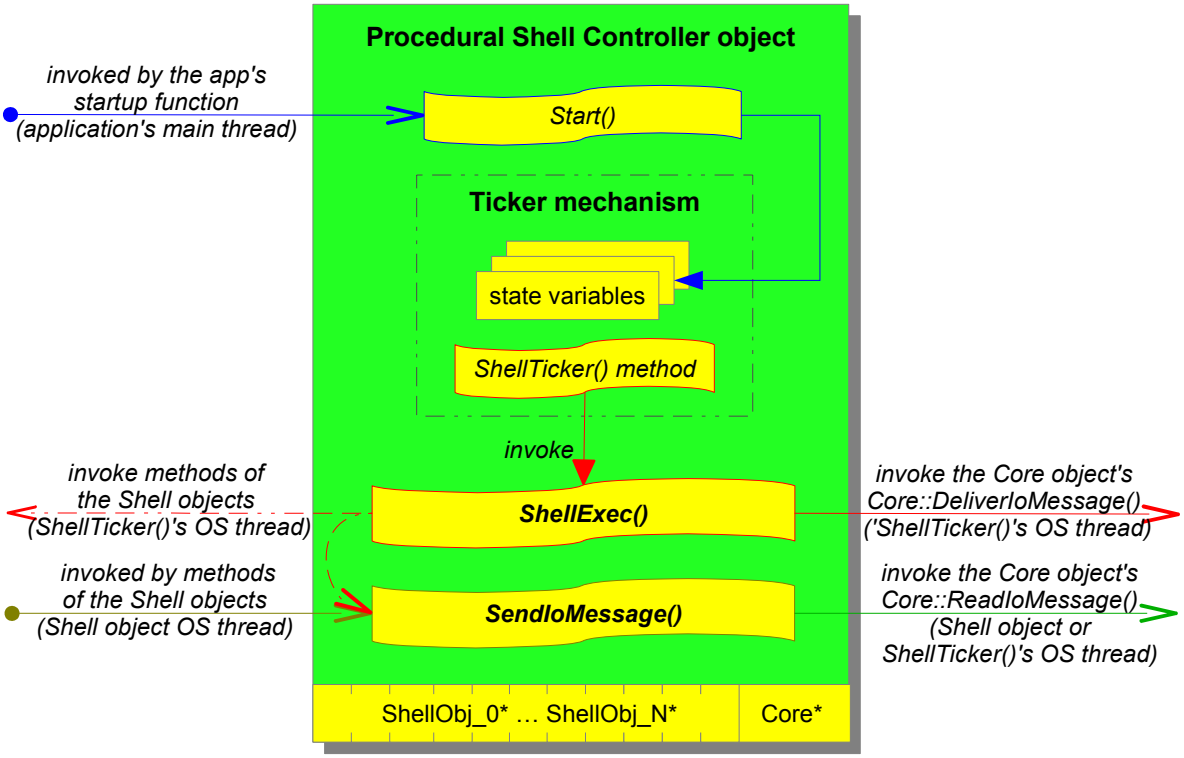

Fig.23: architecture of a procedural Shell Controller object

➔ ***the "Ticker mechanism" of a procedural Shell Controller object must be implemented as a private 'ShellTicker()' method which runs in a dedicated OS thread and which regularly invokes the Shell Controller's 'ShellExec()' method;*** the interval at which the 'ShellTicker()' method invokes the 'ShellExec()' method must be programmable via the Shell Controller's 'Start()' method

➔ ***a procedural Shell Controller object can be implemented as any user-defined class***

➔ ***the 'Start()' method of a procedural Shell Controller object must control all the functional parameters of the "Ticker mechanism"***, i.e. it must be able to [at least] start, stop, and set the tick rate of the "Ticker mechanism", based on the call arguments

✗ for example, a simple "Ticker mechanism" can be implemented for a procedural Shell Controller as follows:

```
// declaration of the Shell Controller class
class MyShellController {
   int tick=0;              // the "current" tick value, initialized with 0
   bool Start(int newTick); // the "Ticker mechanism's" 'Start()' method
   void ShellTicker();      // the "Ticker mechanism's" 'ShellTicker()' method
   [...] // the rest of the objects and methods in the Shell Controller
}
```

```
// definition of the Shell Controller's 'Start()' method
bool MyShellController::Start(int newTick) { // newTick==0 means stop ticker
   if (tick==0 && newTick>0) {               // if ticker stopped then start
      tick=newTick;                          // atomic on any modern system
      new std::thread(&MyShellController::ShellTicker, this);
      usleep(2*tick*1000);                   // make sure ShellTicker started
      return 1;                              // ticker successfully started
   }
   if (tick>0 && newTick==0) {               // if ticker started then stop
      tick=0;
      while(tick!=-1) usleep(1000);          // wait for ShellTicker to stop
      return 1;                              // ticker successfully stopped
   }
   return 0;                                 // didn't perform any action
}

// definition of the Shell Controller's 'ShellTicker()' method
void MyShellController::ShellTicker() {
   while (tick>0) {                          // while ShellTicker is running:
      ShellExec();                           //    call ShellExec()
      usleep(tick*1000);                     //    wait tick milliseconds
   }
   tick=-1;                                  // ShellTicker is now stopped
}
```

In terms of execution threads, the 'Start()' method of a procedural Shell Controller object is executed by the application's main thread (because the 'Start()' method is invoked by the application's startup function, see “The application startup function” paragraph earlier in this chapter), the 'SendIoMessage()' method is executed by the OS thread that invokes the method (which may be any OS thread, see “The Shell Controller object” paragraph earlier in this chapter), and the 'ShellExec()' method, together with any other object methods and/or functions that it may invoke during its execution, are executed by the dedicated OS thread that executes the Shell Controller's 'ShellTicker()' method.

Thread-safety in a procedural reference Shell module

An important characteristic of most procedural host frameworks that are actively used and/or maintained at the time of writing this document (e.g. Win32/64, POSIX-compatibles, etc) is that they do not provide any inbuilt multi-threading protection for many of their their native objects' methods and/or API functions; in this context, the Shell module of a user application that is built “on top of” a typical procedural host framework should either be single-threaded, or it should provide its own multi-threading protection mechanism when invoking the host framework's object methods and/or API functions.

Given the above, the following conditions are *sufficient* for a reference Shell module architecture implemented “on top of” a procedural host framework to be single-threaded, and thus compatible with a very broad range of procedural host frameworks:

1. ***the Shell objects and the Shell Controller object are implemented strictly as described in the “***Procedural Shell objects***” and “***Procedural Shell Controller object***” paragraphs above***
2. ***any and all of the Shell objects' methods are invoked exclusively from methods of other Shell objects, and/or from the Shell Controller's 'ShellExec()' method*** (this also implies that ***none of the Core module objects should invoke any method of any Shell object***)
3. the procedural Shell objects' “ticker mechanism” (see the “Procedural Shell objects” paragraph above) is executed by the same OS thread as the procedural Shell Controller's “ticker mechanism”(see the “Procedural Shell Controller object” paragraph above): a simple implementation of this condition is to have ***the Shell Controller's 'ShellExec()' method successively invoke, at the beginning of its execution, each of the procedural Shell objects' 'monitor()' method***; this is illustrated in Fig.24 below:

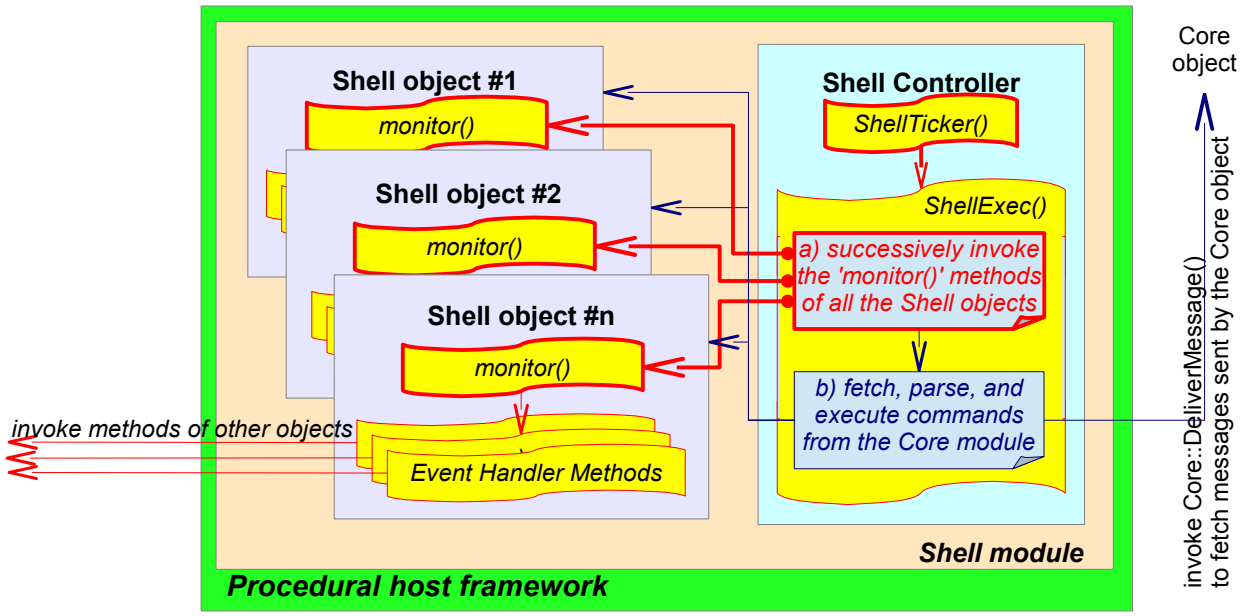

Fig.24: thread-safe architecture for a procedural reference Shell module: all the call chains inside the Shell module start from the Shell Controller's 'ShellTicker()' method, and thus all the methods in the Shell module are executed the OS thread that runs the 'ShellTicker()' method, i.e. the Shell module is single-threaded and thus thread-safe

If the three conditions above are obeyed, then any and all activitides (read: call chains) inside the Shell module will originate exclusively from the Shell Controller's 'ShellTicker()' method, and thus ***any and all of the Shell objects' methods will always be executed exclusively by a single OS thread*** (namely, the OS thread that executes the Shell Controller's 'ShellTicker()' method).

Porting to a new host framework

As previously described in "The architecture of a libagents application" paragraph earlier in this document, *all* the integration functions of a libagents application with its operating environment are performed *exclusively* by the application's Shell module, while the application's core logic is implemented in a platform-independent way by the application's Core module; in this context, and given the reference Shell module architecture as described in the "Reference Shell module architecture" paragraph earlier in this chapter, *the process of porting a libagents application from one host framework to another can be easily formalized*, and it consists of the following steps:

1. ***re-writing the implementation of the Shell objects' methods***: because the signature and functionality of the Shell objects' methods is host framework-independent, only the *implementation*, but not the signature and functionality, of the Shell objects' methods needs to be changed (see "The Shell objects" paragraph earlier in this chapter)
2. ***re-writing the implementation of the Shell Controller object***: this step will require re-writing the implementation of the Shell Controller's "Start()" method, and, in most cases, changing the implementation of the Shell Controller's "Ticker mechanism" and changing the Shell Controller's base class (if any, see "The Shell Controller object" paragraph earlier in this chapter)
3. ***re-writing the implementation of the application's startup function***: as previously described in "The application startup function" paragraph earlier in this chapter, the startup function of a libagents application consists of one or two host framework-specific, application-independent, code block(s), and a contiguous host framework-independent, application-specific, code block; in this context, the process of porting the application's startup function consists of changing only *the host framework-specific code block(s)* in the

startup function according to the new host framework specifications (no changes will [generally] be required to the application-specific code block)

- IMPORTANT: ***apart from the steps (1), (2), (3) listed above, no other changes should be necessary for porting a libagents application from one host framework to another***

The libposif library

The 'libposif' library is a multi-platform implementation of several key system interface objects (e.g. networking objects, file system interface, utility objects, etc) which can be used out-of-the-box as Shell objects in a reference Shell module implementation.

The libposif library is available at http://www.itgroup.ro/libposif and http://libposif.sourceforge.net.

Implementation of cross-referencing objects

A problem which will often be encountered when implementing a libagents applications will be *object cross-referencing*, i.e. when the declarations of two classes A and B both contain references to objects of the other class' type. In this case, the naive C++ class layout where the header file of a class simply #includes the header files of all the classes it depends upon (whether by instantiation of objects, or by object references) cannot be used without special precautions in conjunction with the cross-referencing classes, because said naive solution would lead to 'undefined symbol' errors during compilation.

- an illustrative example of cross-referenced classes involves the Shell Controller object of a reference Shell module architecture (see "The Shell Controller object" paragraph earlier in this document), where the Shell Controller object must contain a set of pointers each pointing to a different Shell object, and, in turn, each Shell object must contain a pointer which points back to the Shell Controller object; in this case, simply #including the header files of all the Shell objects in the header file of the Shell Controller object, and #including the header file of the Shell Controller object in the header files of each Shell object, will lead to 'undefined symbol' errors during compilation, and the compilation will fail

The C++ language allows for two main approaches for solving the class cross-referencing problem:

1) the first approach consists of using the *'class' forward declaration prefix* when declaring a reference to an object inside the referrer class declaration, and #including the header files of the cross-referencing classes *in each others' header files* (note that without using the 'class' forward declaration keyword prefix, an 'undefined symbol' error would be issued at compilation in the *referred* class' header file); this approach is illustrated for two cross-referencing classes Class1 and Class2 below:

   - ```
     // Class1 header file
     #ifndef _class1_h_
     #define _class1_h_
     #include "class2_h" // include Class2's header file in Class1's h file
     class Class1 {
        class Class2 *class2Reference; // use the 'class' prefix keyword
        [...]
     }
     #endif

     // Class1 implementation file
     #include "class1_h"
     [...] // code has access to both Class1 and Class2's header files
     ```

   - ```
     // Class2 header file
     #ifndef _class2_h_
     #define _class2_h_
     #include "class1_h" // include Class1's header file in Class2' h file
     class Class2 {
     ```

```
   class Class1 *class1Reference; // use the 'class' prefix keyword
   [...]
}
#endif

// Class2 implementation file
#include "class2_h"
[...] // code has access to both Class1 and Class2's header files
```

The approach (1) illustrated above is an elegant solution to the class cross-referencing problem in the sense that it uses the naive coding style whereby the header file of a class #includes the header files of all the object types it makes use of, such that when inspecting the #include section of a class' header file it is immediately visible what other classes it depends upon (i.e. including any classes it references). However, ***this solution has a significant drawback***, namely: consider a set of classes {A1...An} whose header files #include the header file of a class B, and also consider that class B is cross-referenced with a class C and thus it #includes class C's header file; in this case, each time the header file of class C changes, both class B *and all the classes {A1...An} need recompilation* when building the project (because the header files of the classes {A1...An} #include [indirectly], via class B's header file, the header file of class C)

2) the second approach consists of using the *'class' forward declaration prefix* when declaring a pointer to a referenced object inside the referrer class declaration (i.e. same as in (1) above), and #including the header file of cross-referencing classes *in each others' definition (cpp) files* (note that without using the 'class' forward declaration keyword, an 'undefined symbol' error would be would be issued at compilation in the *referrer* class' header file); this approach is illustrated for two cross-referencing classes Class1 and Class2 below:

- ```
  // Class1 header file
  #ifndef _class1_h_
  #define _class1_h_
  class Class1 {
     class Class2 *class2Reference; // use the 'class' prefix keyword
     [...]
  }
  #endif

  // Class1 implementation file
  #include "class1_h"
  #include "class2_h" // include Class2's header file in Class1's cpp file
  [...] // code has access to both Class1 and Class2's header files
  ```

- ```
  // Class2 header file
  #ifndef _class2_h_
  #define _class2_h_
  class Class2 {
     class Class1 *class1Reference; // use the 'class' prefix keyword
     [...]
  }
  #endif

  // Class2 implementation file
  #include "class2_h"
  #include "class1_h" // include Class1's header file in Class2's cpp file
  [...] // code has access to both Class1 and Class2's header files
  ```

The approach (2) illustrated above solves the problem of class cross-referencing, but it lacks in elegance in the sense that, in order to identify the classes upon which a given class depends on, one has to inspect both the class' header file (for any header files it may include), *and also the class' definition file* (for the header files of the classes it references). However, ***this approach (2) has a significant advantage over (1)***: specifically, given a set of classes {A1...An} whose header files #include the header file of a class B, and if class B is cross-referenced with a class C and thus *its .cpp definition file* #includes class C's header

file, then a change in class C's header file will require recompilation of only class C and B, while *none of the classes {A1...An} will need recompilation* when building the project (because each of {A1...An} includes only class B's header file, which neither changed, nor does it #include the changed header file of class C)

The two approaches presented above both solve the cross-referencing classes problem in C++, and [one of them] must be used on *all* occasions where class cross-referencing needs to be implemented in a C++ application. In terms of choosing one solution over the other, the relative advantages and disadvantages of the two solutions (as presented above) should be considered.

- *Note:* the 'class' prefix keyword user for declaring a pointer to a class (i.e. the construct 'class Type *ptr) is a new construct introduced by C++11; however, this construct will always be available in a libagents application because any libagents-based application requires a compiler with C++11 support
- IMPORTANT: the C++11 forward declaration construct 'class MyType *prt' can be used directly only if 'MyType' belongs to the top-level namespace; alternatively, if 'MyType' belongs to a user-defined namespace, e.g. 'MyNamespace::MyType', then in order to make a refence to the class 'MyNamespace::MyType' in some referrer class *one needs to explicitly declare 'MyType' as a member of 'MyNamespace' in the referrer class' header file* before using the forward declaration construct 'class MyNamespace::MyType *ptr':

```
// refrerrer.h
namespace Namespace {class Referred;} // declare 'Referred' as part of 'Namespace'

class Referrer {
 class Namespace::Referred *referred; // now 'Namespace::Referred' can be used
 ...
};
```

Cross-referenced 'enum's

None of the two solutions presented in the "Implementation of cross-referencing objects" paragraph above can be used for solving cross-referencing 'enum' values in the .h declaration files of cross-referencing classes; more specifically, two cross-referencing classes can use each others 'enum' definitions (e.g. for initializing a data member, or for dimensioning an array, etc) ***exclusively inside their .cpp definition file*** (e.g. in the constructor definition), but not inside their .h declaration file; else, the compilation process will fail with 'undefined symbol' errors.
The example below illustrates the correct, and respectively incorrect, usages of cross-defined enum values in two cross-referencing classes A and B:

```
// A.h
#ifndef A_h
#define A_h

#include "B.h" // include cross-referenced class B's header file
class A {
 class B *b; // reference to class B
 enum {e=10};
 int i=B::e; // cannot use enum defined in namespace of cross-referenced class B
 int i;      // OK, initialization of 'i' is performed in A.cpp
 A();
}
#endif

// B.h
#ifndef B_h
#define B_h

#include "A.h" // include cross-referenced class A's header file
class B {
 class A *a; // reference to class A
```

```
 enum {e=100};
 int i[A::e];// cannot use enum defined in namespace of cross-referenced class A
 int *i;     // OK, dimensioning of 'i' array is performed in B.cpp
 B();
}
#endif

// A.cpp
#include "A.h"
A::A() {
   i=B::e; // initialize with enum value defined in class B
}

// B.cpp
#incude "B.h"
B::B() {
   i=new int[A::e]; // dimension with enum value defined in class A
}
```

Caveats and limitations

- ***the first access to any Agent/Task/Thread/Core object*** *after said object has been created* ***must be to invoke its corresponding Start method*** *(i.e. Thead::StartAgent()/Task::StartThread()/Core::StartTask/Core::Start());* ***any [other] access to an object's public members before the object has been Start()-ed will result in undefined behavior***
- *the 'onStarted()' methods of Agent/Thread/Task objects are executed by the OS thread that invoked the corresponding 'StartAgent()'/'StartThread()'/'StartTask()' methods*
- ***the 'onMessage()' methods must not invoke any wait operations (e.g sleep(), etc)***
- ***messages can be sent only to existing Agent objects, and they can be delivered only to Agent objects that have been started (via the 'StartAgent()' method) by the time the message is scheduled for delivery.*** *Any message scheduled for delivery to an Agent object that has not [yet] been started at the time when the Agent must received the message, or to an Agent object that has been killed (directly via the 'KillAgent()' method, or by having its container Thread killed via the 'KillThread()' method, or by having its container Task killed via the 'KillTask()' method), will generate a runtime error, and the application will be immediately terminated*
- *because of the multi-threaded nature of a libagents application's Core module, special care has to be taken when killing Core module objects at runtime; in particular, killing an Agent object while it is executing its 'onMessage()' method, or killing a Task/Thread/Agent object while one of its methods is being executed by another OS thread, etc, will cause the application to have undefined behavior (including crash).* ***As a rule of thumb, a Task/Thread/Agent object should only be killed [at runtime] if the application is in a state where it is certain that no method of said object is being executed, by any OS thread, at or after the time when the object is killed***

Messaging performance

The transmission of a message, whether it is a targeted message or a broadcasted message, incurs a cost of ~0.05ms/message for the message sender, and an additional ~0.05ms/message *for each message receiver*, on a typical x86@2GHz *single-core* CPU.

For example:

- the cost of transmitting a *targeted message* (i.e. from one sender Agent to one receiver Agent) on a typical x86@2GHz single-core CPU is ~0.05ms/message for the message

sender, and an additional ~0.05ms/message for the message receiver, such that ~10,000 targeted messages/s can be sent on a typical x86@2GHz single-core CPU (the same cost is also incurred on a multi-core CPU if the sender and receiver Agents are part of the same Thread object, and are thus executed by a single CPU core)

- the cost of transmitting a *broadcasted message* on a typical x86@2GHz single-core CPU is ~0.05ms/message for the message broadcaster, and an additional ~0.05ms/message for *each* message receiver, e.g. ~5,000 messages/s can be broadcasted to 3 subscribers (0.05ms for the broadcaster + 3*0.05ms for each subscriber = 0.2ms => 5,000 messages/s) on a typical x86@2GHz single-core CPU (the same cost is also incurred on a multi-core CPU if the sender and all the receiver Agents are part of the same Thread object, and are thus executed by a single CPU core)
- on a multi-core CPU, the costs incurred by sending and receiving messages are distributed among the Threads that send, and respectively receive, the messages; e.g. on a quad-core typical x86@2GHz CPU with one message sender Agent residing in one Thread object and 3 subscriber Agents residing in 3 other Thread objects, a number of ~20,000 messages/s can be broadcasted to the 3 subscribers (this calculation assumes that the 0.05ms cost incurred by the broadcaster Thread and each of the 3 subscriber Threads are distributed evenly by the OS on the 4 CPU cores, such that the [distributed] total penalty remains 1*0.05ms => 20,000 messages/s)

Profiling tests have been performed on several libagents applications, and they suggest that memory allocations and deallocations take about 80% of the time the application spends with executing libagents v2.0.x library methods. The tests suggest that replacing several key storage objects in the libagents v2.0.x library which are implemented as dynamic objects with (quasi)-statically allocated objects could improve the messaging performance by a factor of up to 4x. Attempting the above-mentioned optimizations is planned for a future revision of the libagents library.

Download

The latest version of the libagents library source code and documentation is available at:

- http://www.itgroup.ro/libagents
  and/or
- http://libagents.sourceforge.net

History

2.0.1

- corrections in the doc: the names of some objects and methods in several diagrams were not updated according to the names that changed in v2.0.0

2.0.0

- removed all the 'LibagentsComponent()=delete' default constructors (they are not necessary because C++11 does not implicitly declare/define them if any constructor is defined)
- *the size of the internal message buffers can be specified separately for the Core object and for each Thread object* (used to be specified only globally via the Core object constructor)
- renamed Agent::onMessageReceived() to Agent::onMessage()
- renamed Core::SendIntercomMessage(), Core::onIntercomMessageReceived() to *Core::SendIoMessage()* and *Core::onIoMessage()*
- removed the 'Core::Intercom' object from the public API:
  - replaced Core::Intercom::PutMessage() and Core::Intercom::GetMessage() with Core object methods *Core::ReadIoMessage()* and *Core::DeliverIoMessage()*
  - renamed "intercom messages" to *"I/O messages"* throughout the document
- added Task::parentCore() method to the Task class for the sake of consistency with Agent and Thread classes (not really needed, the 'Task::core' pointer can be used instead)
- replaced the Core *core() methods in 'Agent', 'Thread', 'Task' classes with pointer variables *Core *core*
- changes in the Core module API to allow future implementation to use an underlying data type for 'message_t' which requires that the object is not changed (e.g. std::vector can be reallocated when inserting new elements, which invalidates any pointers and/or iterators):
  - Core::onMessage(message_t &msg) changed to Core::onMessage(*const* message_t &msg)
- renamed alphanum_t and its methods to *data_t, data_t::is_number(), data_t::is_string(), data_t::is_pointer()*, *data_t::number(), data_t::string(), data_t::pointer()*
  - added convenience methods *data_t::integer()* and *data::is_integer(),* useful e.g. when using enum values in 'case' statements
- renamed id_t to *compid_t* in order to eliminate conflict with gcc libraries' 'id_t' ('compid_t' stands for "component ID")
- replaced naming convention for Shell Controller method ShellController::NotifyCore() with *ShellController::SendIoMessage()* (this affects only this document, not the 'libagents' code)
- added 'Sys' class and moved Core::ticker() and Core::hwthreads() to *Sys::ticker()* and *Sys::threads()*
- other minor changes

1.0.3

- added to "The alphanum_t data type" paragraph: *"it can be safely assumed that a 16-bit integer value, either signed or unsigned (i.e. of type int16_t or uint16_t), can be accurately stored inside an 'alphanum_t' data cell ..."*
- added to "The message_t data type" paragraph: *"Note: the 'message_t::to_string()' method*

can serialize messages with at most 255 data cells ...”

- changed in “Caveats and limitations” paragraph to: “*messages can be sent only to existing Agent objects, and they can be delivered only to Agent objects that have been started ...”*
- added to “Caveats and limitations” paragraph: “*because of the multi-threaded nature of a libagents application's Core module, special care has to be given when killing Core module objects at runtime ...”*
- other minor changes

1.0.2

- added to “The alphanum_t data type” paragraph: “*In any case, if an attempt is made at runtime to load an alphanum_t data cell with an integer value which is too large ...*”
- other minor changes

1.0.1

- minor changes

1.0.0

- initial version of this document

ToDo

- kill a task (implement the Core::KillTask() method)
- let actors crash gracefully (w/o crashing anything else, but destroying whatever objects they have created, etc)
  - maybe each actor should notify a supervisor agent when they create and object, and then the supervizor will delete the objects that were created by a crashed agent, or
  - maybe use some sort of exceptions-based mechanism
- implement debug levels