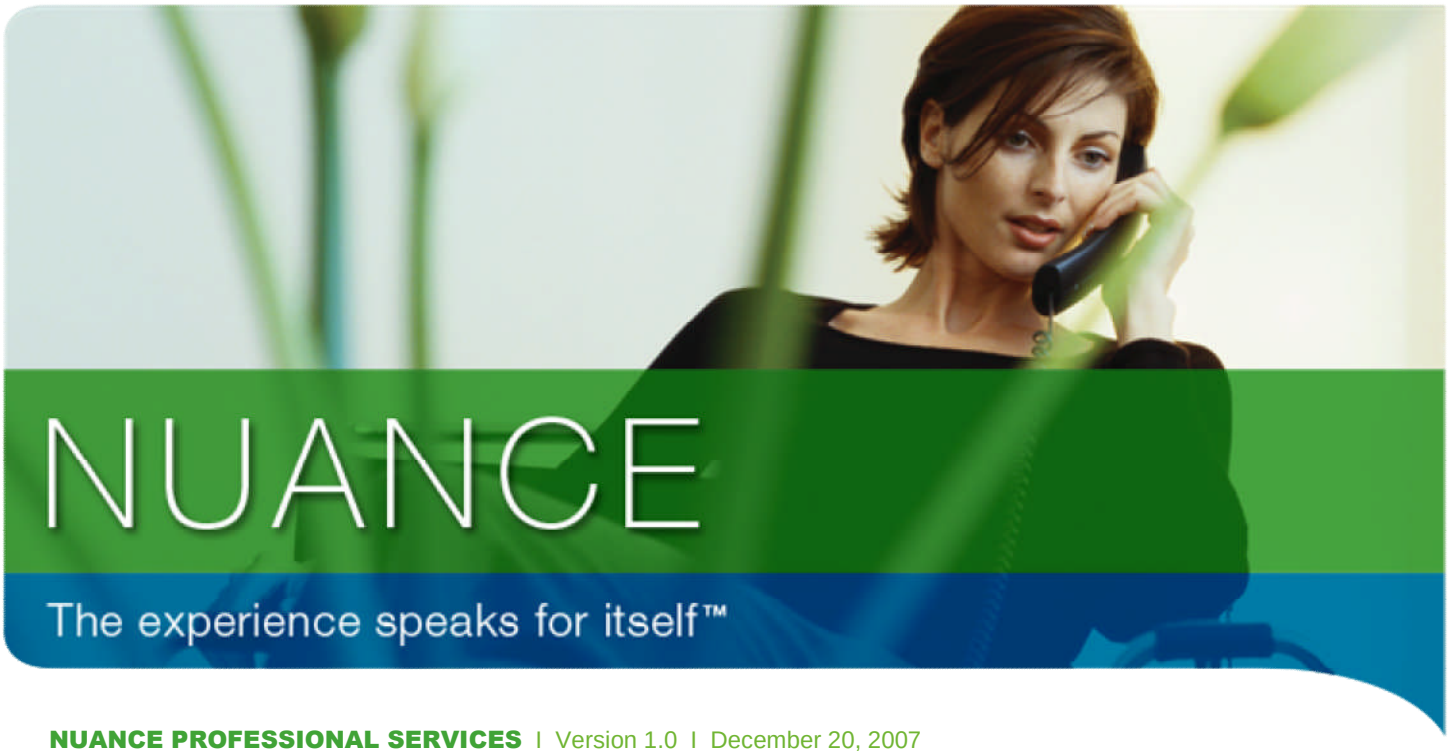


NUANCE PROFESSIONAL SERVICES I Version 1.0 I December 20, 2007

Unified API Design

for SREC RC-1 for Android

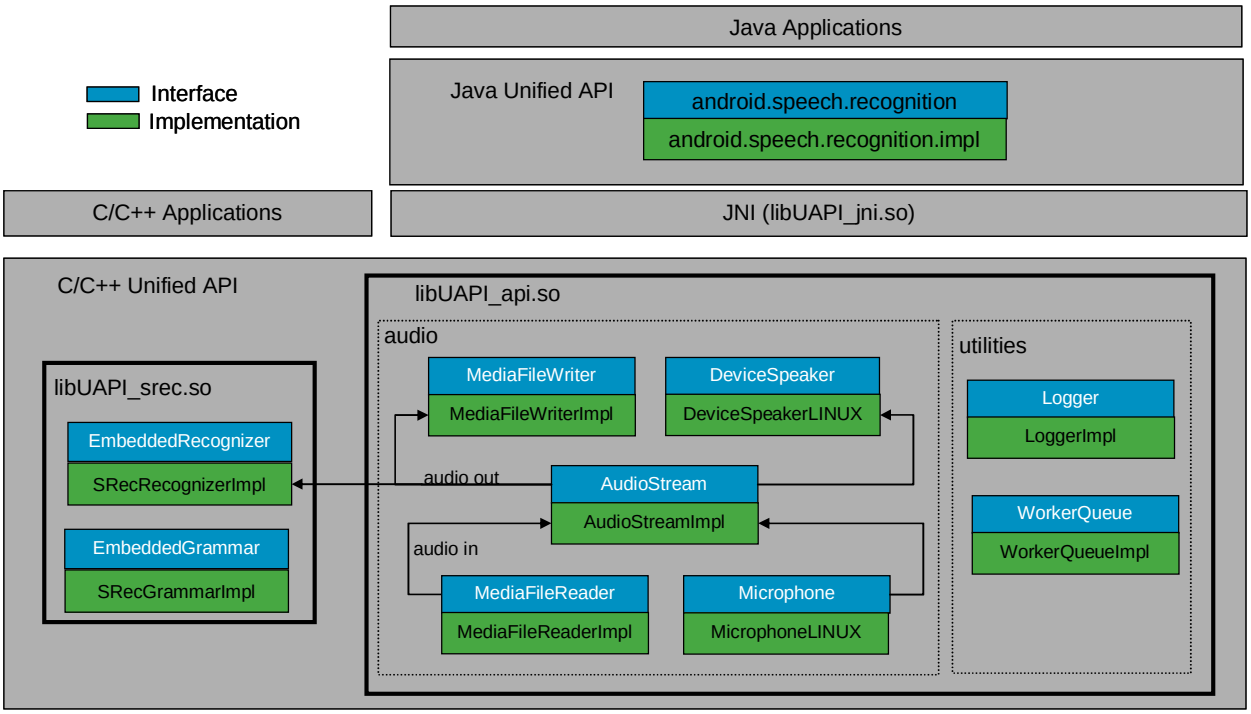
NUANCE

DOCUMENT HISTORY

Date	Revised by	Approved by	Version	Summary of Changes
12/19/2007	Marc Robitaille	Andy Wyatt	1.0	Initial internal version.

TABLE OF CONTENTS

1 OVERVIEW



The Unified Application Programming Interface [Unified API, or UAPI], is a flexible and versatile API for accessing speech software from a mobile device. It can be used by Java or C/C++ (native) applications. This first release of the UAPI includes implementations for the SREC embedded recognizer and code ported for the Android mobile OS, as well as developer test builds for Ubuntu Linux and Windows XP.

The Unified API is a C++ or Java API. When looking at the source tree, the public interfaces can be found:

- cpp\api\include
- java\api\src\android\speech\recognition

The functionality of the Unified API is grouped into 5 modules (resources). These are:

- EmbeddedRecognizer
- Microphone
- DeviceSpeaker
- MediaFileReader
- MediaFileWriter

Typically an application would only have a Recognizer and Microphone during recognition. The others are available for convenience.

The Unified API is multi-threaded.

- Each resource has a WorkerQueue.
- There is a concept of “Task” objects that represent work to be executed via the WorkerQueue of the resource in question.

The Unified API is mainly asynchronous. The only synchronous functions are getInstance() and create(). To handle the asynchronous operations resources have associated “Listener” objects. They are used to receive progress of asynchronous operations.

The Java and C++ code handle the errors in a different way.

- C++ returns the status of the request in the “ReturnCode” variable. Ex: recognizer->Recognize(… , UAPI::ReturnCode & rc);
- Java throws exceptions from the JNI layer. Ex: throw new illegalArgumentException(“invalid name”);

2 SMARTPROXY

Many of the methods defined in the Unified API return a SmartProxy and have one or more SmartProxy arguments. Users should treat a SmartProxy as if it is a pointer to the underlying object it wraps. Smart proxies are modelled after the boost shared_ptr design, discussed here: http://www.boost.org/libs/smart_ptr/shared_ptr.htm. The key point about the smart proxies is that they take care of deleting the underlying object (they internally call the delete operator). They use a reference count to do so. When that count goes down to zero, meaning that no more proxies are referencing the underlying object, the object is deleted. For this reason, the users of UAPI MUST NOT call delete on any of the Proxy objects.

The UAPI uses Listeners to notify the application about the progress of asynchronous operations. Those listeners, created by the application, are passed to the UAPI in the setListener function. For example, the Microphone class has the `void setListener(MicrophoneListenerProxy& listener, ReturnCode::Type& returnCode)` method. To set its microphone listener, the application must declare a class that implements the MicrophoneListener interface (e.g. MyMicrophoneListener : public MicrophoneListener). Because the argument of setListener is a MicrophoneListenerProxy, the application must also declare a Proxy to that class (e.g. MyMicrophoneListenerProxy). To do this, these two macros should be used.

- DECLARE_SMARTPROXY(ExportSymbol, Proxy, SuperProxy, Interface)
  - ExportSymbol: (optional) used only if this method has to be exported from a library.
  - Proxy: proxy class that you are defining.
  - SuperProxy: interface that the Proxy implements (we have class Proxy : public SuperProxy).
  - Interface: Real class that is wrapped by this Proxy.

- DEFINE_SMARTPROXY(Namespace, Proxy, SuperProxy, Interface)
  - Namespace: (optional) used only if your Proxy has to be defined inside a namespace.
  - Proxy: proxy class that you are defining.
  - SuperProxy: interface that the Proxy implements (we have class Proxy : public SuperProxy).
  - Interface: Real class that is wrapped by this Proxy.

2.1 SmartProxy Example

We will look at the Microphone class to show how to use the SmartProxy. We will also show how a user can define their own SmartProxy.

```
//put in MyMicrophoneListener.h
MyMicrophoneListener : public MicrophoneListener
{
  public:
    virtual void onStarted() {}
}
DECLARE_SMARTPROXY(,MyMicrophoneListenerProxy, MicrophoneListenerProxy,
MyMicrophoneListener)

//put in MyMicrophoneListner.cpp
DEFINE_SMARTPROXY(,MyMicrophoneListenerProxy, MicrophoneListenerProxy,
MyMicrophoneListener)

void main()
{
  MicrophoneProxy mic = Microphone::getInstace();
```

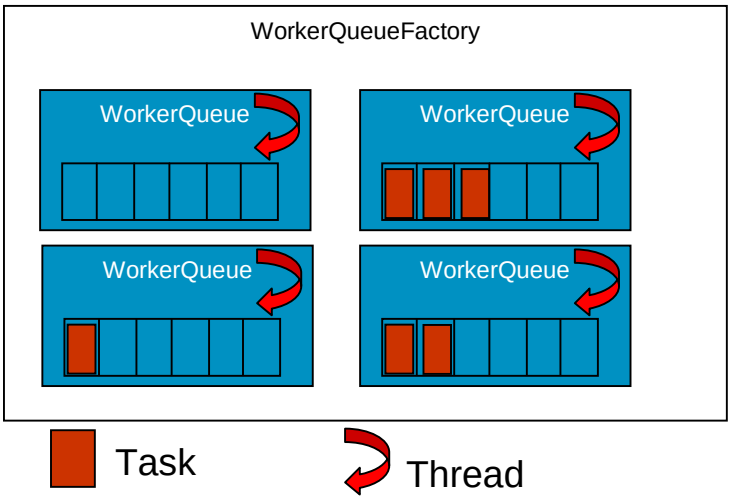
```
    //use « new » but don’t use delete
    MyMicrophoneListenerProxy listener(new MyMicrophoneListener());

    mic->setListener(listner); //use mic like a pointer

    mic->start(); //use mic like a pointer
  }
```

In this example, we see that the application declares its own MicrophoneListener. `DECLARE_SMARTPROXY` and `DEFINE_SMARTPROXY` are also used to define a Proxy to MyMicrophoneListener. This is required to match the parameter type of the setListener method. We see that the operator new is used to create an instance of MyMicrophoneListener. This new instance is stored in the member variable “listener”. The memory allocated will not be freed until 1) the listener variable goes out of scope (when we exit the main) and 2) the Microphone object no longer needs to reference that listener object.

2.2 SmartProxy Guidelines

- Retain a copy of (as opposed to a reference to) a proxy to ensure that the underlying object does not get destroyed until the proxy gets destroyed.
- Pass a proxy around by reference (as opposed to pass-by-value) for efficiency.
- There are two kinds of proxy objects: stack-based or heap-based. The most common type of proxy is a stack-based local variable declared inside a function body. This proxy is ideal because it is thread-safe (no other thread can access function local variables) and gets de-allocated automatically at the end of the method, along with all other local variables. Heap-based proxies, on the other hand, must be explicitly deleted.
- Individual proxy objects are not thread-safe so if you absolutely must access the same proxy from multiple threads (as opposed to giving each thread its own proxy) then you must synchronize access to it manually.
- Factory methods should return proxies by value.
- Methods should accept proxies by reference.
- Never construct more than one root proxy per object. The first SmartProxy ever constructed around an object is called the "root proxy". Root proxies are constructed using SmartProxy`(void* object, ReturnCode::Type& returnCode)`. A root proxy contains the reference count of the underlying object. This means that if you construct multiple root proxies per underlying object, each root will see a difference reference count and one root will destroy the object before the other roots are done using it.
- If you absolutely must wrap the same pointer multiple times and you do not have access to existing proxies that wrap it, simply store <code>`SmartProxy.getCounter()` alongside the pointer and construct all future proxies using `SmartProxy(RefCounter*)` instead.

3 WORKERQUEUE - THREADING



3.1 WorkerQueueFactory

The WorkerQueueFactory can be viewed as a pool of WorkerQueues. By default the WorkerQueueFactory has 5 WorkerQueues.

```
class WorkerQueueFactory
{
  public:
    virtual WorkerQueue* getWorkerQueue(ReturnCode::Type& returnCode);
}
```

The main method is getWorkerQueue(). This method is called by the resources (Microphone, MediaFileWriter, EmbeddedRecognizer, etc) when they are initialized. For example, the first time someone calls Microphone::getInstace(), the microphone's constructor calls the WorkerQueueFactory::getWorkerQueue method. This method returns a WorkerQueue. Because the WorkerQueue is taken from a pool, the returned WorkerQueue could be shared with other resources. For this reason, a resource is not allowed to block when running a Task (when Task::run() is called by the WorkerQueue thread).

The WorkerQueueFactory singleton instance is created the first time someone calls WorkerQueueFactory::getInstace(). In a normal system, the singleton instance is never deleted. When we are running tests, the WorkerQueueFactory singleton instance is the last thing deleted by System::dispose(). This ensures us that the WorkerQueue threads will never get deleted while a resource is using it.

3.2 WorkerQueue

A WorkerQueue is a queue of Tasks that are processed in a dedicated thread. When started, each resource is given a WorkerQueue.

The main methods are these two:

```
class WorkerQueueImpl: public WorkerQueue, public Runnable
{
    public:
      virtual void enqueue(Task* task, ReturnCode::Type& returnCode);

    protected:
      virtual ReturnCode::Type runThread();
};
```

enqueue adds a new Tasks to the queue and runThread is called when the Runnable (thread) is started. Each WorkerQueue has a dedicated thread. The runThread method simply runs in a loop that pops from the queue and runs the tasks.

3.3 Task

A Task represents some work that has to be run in the WorkerQueue thread. Task can be of two types:

Task	This represents work to be executed as soon as possible. A new Task is always added to the end of the queue. It will be executed when the WorkerQueue thread dequeues it.
ScheduledTask.	This represents work to be executed later. When the specified timeout occurs the Task will be run.

```
class Task
{
  public:
    virtual void run() = 0;
}

class ScheduledTask: public Task
{
  public:
    ScheduledTask(UINT32 timeout_msec);

    virtual void run() = 0;
}
```

When a Task is de-queued, the thread in the WorkerQueue calls the run method. Right after the Task is run, the Task object is deleted (i.e. it calls delete on the object). This means that the argument passed to enqueue method MUST be a Task object that was created using the “new” operator.

```
    {
        Task * task = queue.pop();
        task->run();
        delete task;
    }
```

3.4 How is this used

The asynchronous functions of the UAPI always run in a WorkerQueue thread. When a function is called, the implementation simply adds a new Task to the WorkerQueue. The application is also guaranteed that the methods in

the Listeners will be invoked from that thread. Because of this, the application is allowed to make another call to the UAPI from within the callback.

Example:

```
MyMicrophoneListener : public MicrophoneListener
{
  public:
    virtual void onStarted()
    {
      printf("we are in the WorkerQueue thread\n");
    }
}
DECLARE_SMARTPROXY(,MyMicrophoneListenerProxy, MicrophoneListenerProxy,
MyMicrophoneListener)


void main()
{
  MicrophoneProxy mic = Microphone::getInstace();

  MyMicrophoneListenerProxy listener(new MyMicrophoneListener());
  mic->setListener(listner);

  mic->start();

  //wait here
}

class StartMicrophoneTask : public Task
{
  public:
   virtual void run() { proxy->runStartMicrophoneTask(); }

  private:
   MicrophoneLINUXProxy proxy;
}

void MicrophoneLINUX::start()
{
  //We are in the application's thread.

  StartMicrophoneTask* task = new StartMicrophoneTask(proxy);
  m_pWorkerQueue->enqueue(task, returnCode);
}

void MicrophoneLINUX::runStartMicrophoneTask()
{
  //we are in the WorkerQueue thread

  lhs_audioinOpenCallback(); //start the audio driver

  listener->onStarted();
}
```

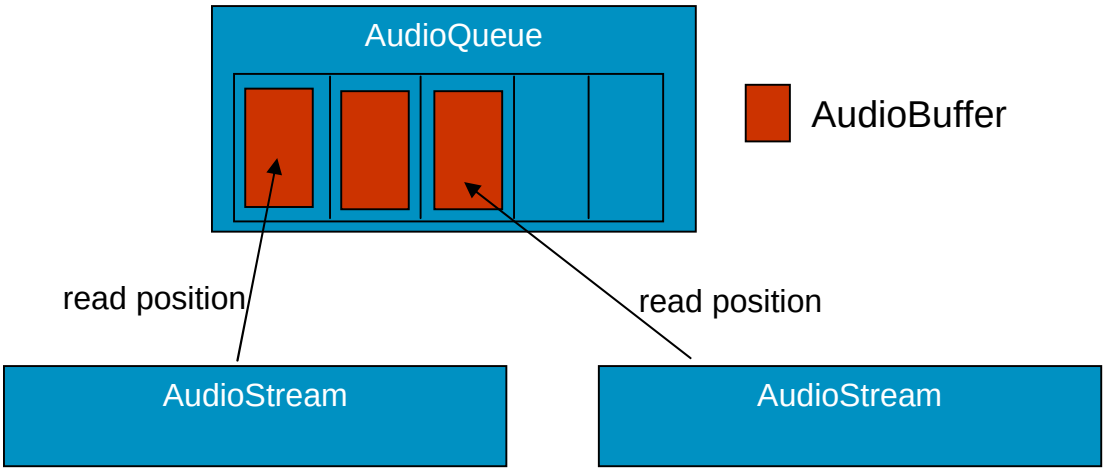
This example is pretty straight forward. The application invokes start. The implementation of `MicrophoneLINUX::start` simply queues a `StartMicrophoneTask`. When the WorkerQueue de-queues that Task, it calls the run method, which in turns calls `MicrophoneLINUX::runStartMicrophoneTask`. Now that we are in the WorkerQueue thread it is safe to call the audio driver methods and to invoke the listener callbacks onStarted().

It is also worth nothing that the StartMicrophoneTask class has a MicrophoneLINUXProxy member variable and not a MicrophoneLINUX pointer. By doing this, the StartMicrophoneTask holds a reference on the MicrophoneLINUX object (increasing the count by 1) and we are sure that the MicrophoneLINUX will not be deleted while the StartMicrophoneTask is queued.

4 AUDIOSTREAM

The UAPI uses AudioStream(s) objects to represent a collection of audio samples. When the application wants to use resources that require audio samples (e.g. DeviceSpeaker, Recognizer MediaFileWriter), it has to create a new AudioStream. This is done on an AudioSource (MediaFileReader, Microphone). The AudioStream is made such that the application never has to worry about how samples are inserted into the AudioStream object. For this reason the AudioStream interface does not have any exposed methods. All the application has to do is create a stream when it wants to start collecting the audio samples and use that stream later to perform an operation like recognition.

How those it work internally?



AudioBuffer

The first object is the AudioBuffer. The AudioBuffer contains the actual audio samples and the encoding type for those samples. Every AudioBuffer derives from a RefCounted base class object. Because each AudioBuffer has a reference count, the audio can be shared by multiple streams. An AudioBuffer does not get deleted until all the active AudioStream(s) have read it. Every time an AudioBuffer was read by a resource (e.g. a Recognizer), it has to be released. When released, the count on the AudioBuffer is decreased by 1. When the count goes down to 0, the buffer is deleted and removed from the Q. This is all done to make sure that the audio is never copied more than once.

AudioQueue

The AudioQueue is mainly a collection of AudioBuffers. The AudioQueue also allows AudioStream(s) to attach to it. When a new AudioBuffer is added by an AudioSource (Microphone, MediaFileReader), its initial reference count is equal to the number of AudioStream(s) attached (number of times AudioStream::create() was called by the application).

AudioStream

The AudioStream public interface does not allow the application to access the audio samples that are stored into AudioBuffer objects. To access it, the UAPI resources implementation will cast this AudioStream object into an AudioStreamImpl object. By doing this the resource implementation have access to these method:

```
    class AudioStreamImpl: public AudioStream
    {
          public:
            static AudioStreamImplProxy create(utilities::AudioQueue* queue,
ReturnCode::Type& returnCode);

            utilities::AudioBuffer* read(ReturnCode::Type& returnCode);

            void release(utilities::AudioBuffer* audioBuffer, ReturnCode::Type&
returnCode);

         private:
           utilities::AudioQueue*        audioQueue;
           utilities::SinglyLinkedNode*  readPosition;
    }
```

When the application calls `create` on an AudioSource, it causes this new stream to attach to the AudioQueue. Note that this AudioQueue is owned by an AudioSource (e.g. Microphone). At that time, the `readPosition` member variable will point to the first available AudioBuffer in the AudioQueue. When the application uses this AudioStream (e.g. when they start recognition) the recognizer will start reading from this stream by calling the `read` function. It will read from the readPosition.

Application:

```
   {
      AudioStreamProxy audio = microphone->createAudio();
      recognizer->recognize(audio, ...);
   }
```

SrecRecognizerImpl:

```
   void recognize(AudioStreamProxy audio, ...)
   {
      AudioStreamImplProxy audioImpl = (AudioStreamImplProxy) audio;

      ReturnCode::Type returnCode;
      AudioBuffer * buffer = audioImpl->read(returnCode);
      if( returnCode == ReturnCode::SUCCESS )
      {
         //we have a valid buffer
         DoInternalRecognition(buffer);

         audioImpl->realease(buffer);
      }
      else if( returnCode == ReturnCode::PENDING_DATA )
      {
          //no audio ready at this time, try reading later
          QueueReadLaterScheduledTask();
      }
      else if( returnCode == ReturnCode::END_OF_STREAM )
      {
          //done, no more audio
      }
   }
```

We can see form that pseudo code that `read` can return 3 different ReturnCode. If SUCCESS is returned, it means that we have a valid AudioBuffer and we can use its samples to perform recognition. Note that once the audio has been used the code has to call the `release` function. This will tell the AudioQueue that this buffer is no longer used by this AudioStream. The reference count on this AudioBuffer will be decreased by 1 at that time. Another possible return code is PENDING_DATA. This means that all the currently available AudioBuffer(s) were read on that stream but there are more to come. In those cases, a ScheduleTask is queue in the WorkerQueue and we will attempt to

read again later. The last possible return code is END_OF_STREAM. This means that this AudioStream does not contain any more AudioBuffer(s), probably because the AudioSource that was filling in AudioBuffer(s) was stopped by the application.

What happen if no AudioStream was created by the application and an AudioSource was started? For example, the application calls microphone.start() and does not call AudioStream::create(). In this case the samples are simply discarded and not stored in the AudioQueue. The audio samples will be saved once someone creates an AudioStream.

5 COMPONENTINITIALIZER

The singletons of the UAPI use a ComponentInitializer. The MicrophoneLINUX is one of those.

```
class MicrophoneLINUX
{
  private:
    class ComponentInitializer
    {
      public:
        ComponentInitializer();
        ~ComponentInitializer();

        ReturnCode::Type returnCode;
    };
    static ComponentInitializer componentInitializer;
};
```

This code is used to initialize static objects from a single thread. Because the componentInitializer is a static member variable of the MicrophoneLINUX class, the ComponentInitializer constructor will be invoked when the process starts up (during static variables initialization). Because this is done from a single thread, it is safe to create the static Mutex used to protect the “instance” variable (the singleton).

```
MicrophoneLINUX::ComponentInitializer::ComponentInitializer()
{
  //safe to create the mutex here.
  mutex = Mutex::create(false, returnCode);
}
```

6 LOGGER

The Logger is a singleton just like the Microphone, the DeviceSpeaker and the EmbeddedRecognizer. All the classes of UAPI use the Logger to report error, warning, information and trace. Because of this, the Logger instance has to be the last object to be deleted. To ensure that this is the case, we use a static LoggerProxy variable **(`makeSureLoggerIsNeverDestroyedLoggerProxy`).**

```
class LoggerImpl
{
  private:
    class ComponentInitializer
    {
      public:
        ComponentInitializer();
        ~ComponentInitializer();

        ReturnCode::Type returnCode;
        LoggerProxy      makeSureLoggerIsNeverDestroyedLoggerProxy;
    };
    static ComponentInitializer componentInitializer;
};
```

The `makeSureLoggerIsNeverDestroyedLoggerProxy` variable is assigned in the `ComponentInitializer` constructor. It goes out of scope in the `ComponentInitializer` destructor. This means that the Logger is valid until the `componentInitializer` variable is destroyed. This happens at process shutdown time.

7 SYSTEM CLASS

The System class is used primarily for unit testing. The Singleton design pattern is used in some of the objects of the Unified API (e.g. Microphone, DeviceSpeaker, EmbeddedRecognizer).

Because our test harness runs multiple tests in a single process, we need a way to recreate the singleton objects every time we start a new test. This is where the System::dispose comes into play. System::dispose is responsible for two main things: 1) cleaning up the Singletons and 2) shutting down the WorkerQueueFactory.

How does the System class work?

7.1 Step 1 – Singletons are added to the System class (`System::add`)

The singletons all derive from the "Singleton" base class.

```
class Microphone : pubic Singleton
class DeviceSpeaker : pubic Singleton
class EmbeddedRecognizerImpl : public Singleton
```

When a singleton is created, it is registered with the System class.

```
instance = new impl::MicrophoneWIN(returnCode);
System* system = System::getInstance(returnCode);
system->add(instance, returnCode);

//1   void System::add(Singleton* singleton, ReturnCode::Type&
returnCode)
//2   {
//3     {
//4       //make sure System::dispose was not called.
//5       LockScope ls(stateMutex, returnCode);
//6       if (shutdownRequested)
//7       {
//8         returnCode = ReturnCode::INVALID_STATE;
//9         return;
//10      }
//11    }
//12    //protect the list of singletons
//13    LockScope ls(singletonsMutex, returnCode);
//14
//15    SingletonInfo* singletonInfo = new SingletonInfo;
//16    singletonInfo->isWaiting = false;
//17    singletonInfo->singleton = singleton;
//18    singletonInfo->condVar =
ConditionVariable::create(singletonsMutex, returnCode);
//19
//20    singletons->push(singletonInfo, returnCode);
//21
//22    singleton->getRoot()->registeredWithSystem = true;
//23  }
```

System::add first checks to make sure System::dispose was not already called (line 6). Then it protects the list of singletons to add the new singleton to its list (line 13). It also does two more things. It creates a ConditionVariable (line 18). This could be used if someone calls Sytems::dispose while the new added singleton is still in the singleton list. It also set the `registeredWithSystem` flag to true. This is used by SmartProxy::onDestruction to know if it has to call System::remove on this object (line 32). More about this in Step 2…

7.2 Step 2 – A singleton is removed from the System class (`System::remove`)

When the SmartProxy reference count on an object goes down to zero, that object can be cleaned up. At that time, the System::remove function will get invoked. See the SmartProxy section for more information.

Let's look at what happens when this code is written.

```
{
   //Step 1 is executed
   MicrophoneProxy mic = Microphone::getInstance(returnCode);
}
//Step 2 is executed
```

Step 1 is executed when the Microphone is created (Microphone's constructor calls system->add).

Step 2 happens when the variable mic goes out of scope. At this point the count in the MicrophoneProxy class is decreases by 1 and goes down to 0. Because the count is 0, the Microphone destructor can now be called (line 39). Before this is done, the Microphone singleton is removed from the System class (line 37).

```
//24  SmartProxy::onDestruction()
//25  {
//26     LockScope ls(root->getMutex(), returnCode);
//27
//28     ARRAY_LIMIT count = root->decrement(returnCode);
//29     if (count == 0)
//30     {
//31       ls.cancel(returnCode);
//32       if (root->registeredWithSystem)
//33       {
//34         System* system = System::getInstance(returnCode);
//36
//37         system->remove(root); //Microphone's root pointer
//38       }
//39       deleteObject(root->getObject()); //Microphone::~Microphone()
//40                                        //is called
//41       delete root;
//42     }
//43  }
```

Note that only the classes that derive from the `Singleton` base class will have the "`registeredWithSystem`" variable set to true. This is the case of the Microphone, the DeviceSpeaker and the EmbeddedRecognizerImpl.

```
//44  void System::remove(SmartProxy::Root* root)
//45  {
//46    //Lock the System mutex to protect the queue of singleton(s)
//47    LockScope ls(singletonsMutex, returnCode);
//48
```

```
//49    //find which entry in the queue contains the root.
//50    FwdIterator it(singletons->begin(), singletons->end());
//51    while (it.hasNext())
//52    {
//53      SingletonInfo* singletonInfo = (SingletonInfo*) it.next();
//54      if (singletonInfo->singleton->getRoot() == root)
//55      {
//56        //we have a match.
//57        if (singletonInfo->isWaiting)
//58        {
//59          //System::dispose was called and it is waiting on the
condition variable (condVar->wait()).
//60          singletonInfo->condVar->signal(returnCode);
//61          return;
//62        }
//63        else
//64        {
//65          //System::dispose is not waiting. We can simply remove
that entry from the list.
//66          singletons->remove(singletonInfo);
//67          delete singletonInfo->condVar;
//68          delete singletonInfo;
//69          return;
//70        }
//71      }
//72    }//while
//73  }
```

System::remove first protects the list of singletons (line 47) and finds the element that must be removed (line 54). Then it checks if System::dispose was called (line 57). If it was NOT called, we simply remove the singleton from the list (line 66). If Sytem::dispose was called, then we have to signal the ConditionVariable (line 60). When this happens, System::dispose is waiting for the singleton to be removed before it returns (line 105). System::dispose will unblock when we signal the ConditionVariable.

7.3 Step 3 – We are shutting down the sytem (`System::shutdown`)

We can now describe how the dispose method works. To understand the need for such a method, let's look at this example:

```
void test1()
{
   MicrophoneProxy mic = Microphone::getInstance(returnCode);
   mic->start();
}

void test2()
{
   MicrophoneProxy mic = Microphone::getInstance(returnCode);
   mic->start();
   sleep(1000);
   mic->stop();
}
```

```
void main()
{
      test1();
      System::getInstance()->dispose();
      test2();
      System::getInstance()->dispose();
}
```

The goal is to have an unused Microphone object when we start “test2”. To do this, we call system->dispose() between the two tests. Because of the nature of test1 (mic->stop() is intentionally not called), System::dispose has to more than simply delete the singleton instance. System::dispose has to be able to do two things. (1) Make sure the Microphone stops recording gracefully (this is to prevent audio driver errors in subsequent tests) and (2) make sure the Microphone “instance” variable is deleted and set back to NULL.

```
//74   void System::dispose(ReturnCode::Type& returnCode)
//75   {
//76     {
//77       LockScope ls(stateMutex, returnCode);
//78       if (shutdownRequested)
//79         return;
//80       shutdownRequested = true;
//81     }
//82
//83     LockScope ls(singletonsMutex, returnCode);
//84     while (!singletons->empty())
//85     {
//86       SingletonInfo* singletonInfo = (SingletonInfo*) singletons-
>front();
//87
//88       SmartProxy::Root* root = singletonInfo->singleton-
>getRoot();
//89       {
//90         // protect the call to getCount()
//91         LockScope rootls(root->getMutex(), returnCode);
//92         if (root->getCount() == 0)
//93         {
//94           // We're too late, the proxy is already being destroyed
by another
//95           // thread.  SmartProxy::onDestruction() will block on
system->remove()
//96           // until we release singletonsMutex. In such a case, we
don't want to
//97           // call shutdown.
//98         }
//99         else
//100        {
//101          singletonInfo->singleton->shutdown(returnCode);
//102        }
//103        singletonInfo->isWaiting = true;
//104      }
//105      singletonInfo->condVar->wait(returnCode);
//106
//107      singletons->remove(singletonInfo);
//108
//109      delete singletonInfo->condVar;
//110      delete singletonInfo;
```

```
//111    }
//112    delete singletons;
//113
//114    // shut down the WorkerQueues
//115    WorkerQueueFactory* workerQueueFactory =
WorkerQueueFactory::getInstance(returnCode);
//116    delete workerQueueFactory;
//117
//118    disposed = true;
//119  }
```

We are in the APP THREAD:
The first thing that we do (line 78) is to check if the method was already called. It is also used to prevent System::add() to add items to the list (line 6).

On line 84, we go through the list of singletons. We will make sure they are all inactive and deleted before we continue to line 115.

On line 91, we lock the SmartProxy::Root mutex. This is needed to protect the getCount() call we are making on line 92. Most of the time the value returned by getCount() is non 0, which means that we execute the code on line 101. This makes perfect sense, System::remove was not called on that object and this is why this singleton object is still in the list. But there is a race condition and this is why we have to do this check on line 92. How can we get this race condition? APP THREAD gets the lock on line 83. It executes until line 88. Then we switch to WORKER QUEUE THREAD. This thread executes line 26 to 30 on the same object that was assign on line 86. At this point, the object has a count of 0, meaning that it is about to be removed from the system class and about to be deleted. In those rare cases, we must not call shutdown on the object since it is no longer used. Instead we wait for remove to be called.

On line 101, we call shutdown on the Singleton object. If we go back to our test1 example, Microphone::shutdown will be called and this will internally queue a StopMicrophoneTask, causing the Microphone audio driver to stop recording. At this point, the dispose method goes into a waiting state (line 105). We are waiting for System::remove to signal the condition variable (line 60).

We are in the WORKER QUEUE THREAD:
When the recording ends, the “count” of the MicrophoneProxy goes down to 0 in SmartProxy::onDestruction (line 29). Then, because this object has `registeredWithSystem` set to true, system->remove is called (line 37). This will signal the condition variable (line 60).

We are in APP THREAD:
The dispose now comes back from its waiting state on line 105. It can then remove this singleton from its list (line 107) and repeat the same operation for the other singletons.

The last thing dispose does is destroy the WorkerQueueFactory (line 116). By doing this, all the WorkerQueue threads will first be joined and then destroyed. This ensures us that nothing is running in the system when line 116 returns.

7.4 Race conditions, deadlocks

The first thing that we can notice is that the actual singleton destructor has not been called (line 39) when the condition variable is signaled (line 60). This means that system dispose could return (running in the APP THREAD) before the singleton destructor (e.g. Microphone::~Microphone()) was called (running in the WORKER QUEUE THREAD). This could be a problem because the application thread could execute a new test which would use an object that is just about to be deleted in another thread. This race condition does not exist because of the following. Before system dispose returns, it destroys the WorkerQueueFactory (line 116). This will not return until all the WorkerQueue threads are joined. Because the singleton is destroyed inside the WorkerQueue thread, we are sure that the singleton will be destroyed (line 39) before system dispose returns.

This code uses 3 mutexes. To avoid dead lock, we have to make sure these mutexes are always locked in the same order. The order is this one:

1	stateMutex	This is used to make sure only one thread creates System::instance. It is also used to protect the shutdownRequested variable used to know if system->dispose was called.
2	singletonsMutex	This is used to protect the list of singletons. It is also used to synchronize the ConditionVariable.
3	SmartProxy::Root::getMutex()	This is used to protect the “count” variable inside the SmarProxy::Root class.

As long as we always lock in this order: 1, 2, 3 we will be ok. This is why on line 31 we unlock the SmartProxy::Root mutex before we call remove on line 37. If we didn’t do this, we would end up locking 3, followed by 1, followed by 2. This would for sure cause a dead lock.