|  |  |
| --- | --- |
| TCS logo B&Y_June_2006USE.jpg | Summary Design Document |

NBGM Replayer

**NBGM Team**

PROPRIETARY & CONFIDENTIAL

**TeleCommunication Systems, Inc**

***Enabling Convergent Technologies*** ®

Tianjin Office, China

**Prepared For: Prepared By:**

TCS TCS

*Disclaimer: the information contained in this document is proprietary and confidential and is intended only for the persons to whom it is transmitted by TeleCommunication Systems, Inc.. Any reproduction of this document, in whole or in part, or the divulgence of any of its contents, without prior written consent of TeleCommunication Systems, is prohibited. Receipt or possession of this document does not convey any rights to disclose its contents, in whole or in part, to any third party, or to develop, manufacture, use, or sell anything described herein.*

Table of Contents

[1. System Design 3](#_Toc350242165)

[1.1. System architecture diagram 3](#_Toc350242166)

[1.2. Class diagram 4](#_Toc350242167)

[1.3. Time sequence diagram 5](#_Toc350242167)

[2. Input File Format 5](#_Toc350242169)

[2.1. Invocation Record File 6](#_Toc350242170)

[2.2. Function configuration File 6](#_Toc350242171)

[3. Client Code 8](#_Toc350242172)

**NBGM Task Replay System**

1. **system overview**

NBGM task replay system is used to replay NBGM API invocation in real time by parsing binary log and configuration file for debugging and presentation purposes.

The system comprises of task interface, app framework, NBGM implementation, python and native modules. These modules can be divided into four layers, based on their functions and dependence relations.

* **Base** **layer**

Underlying tech-support modules, including a platform dependent window

* **Core** **layer**

A complete interface layer, which defines a series of core behaviors for task system

* **Framework** **layer**

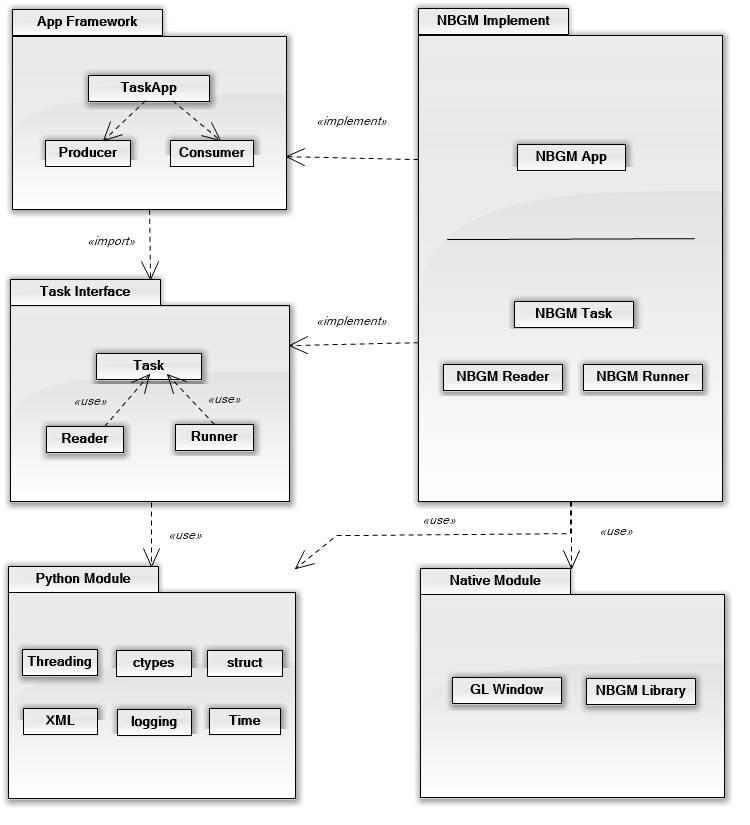
The framework of the system, which takes producer-consumer thread model

* **Business** **layer**

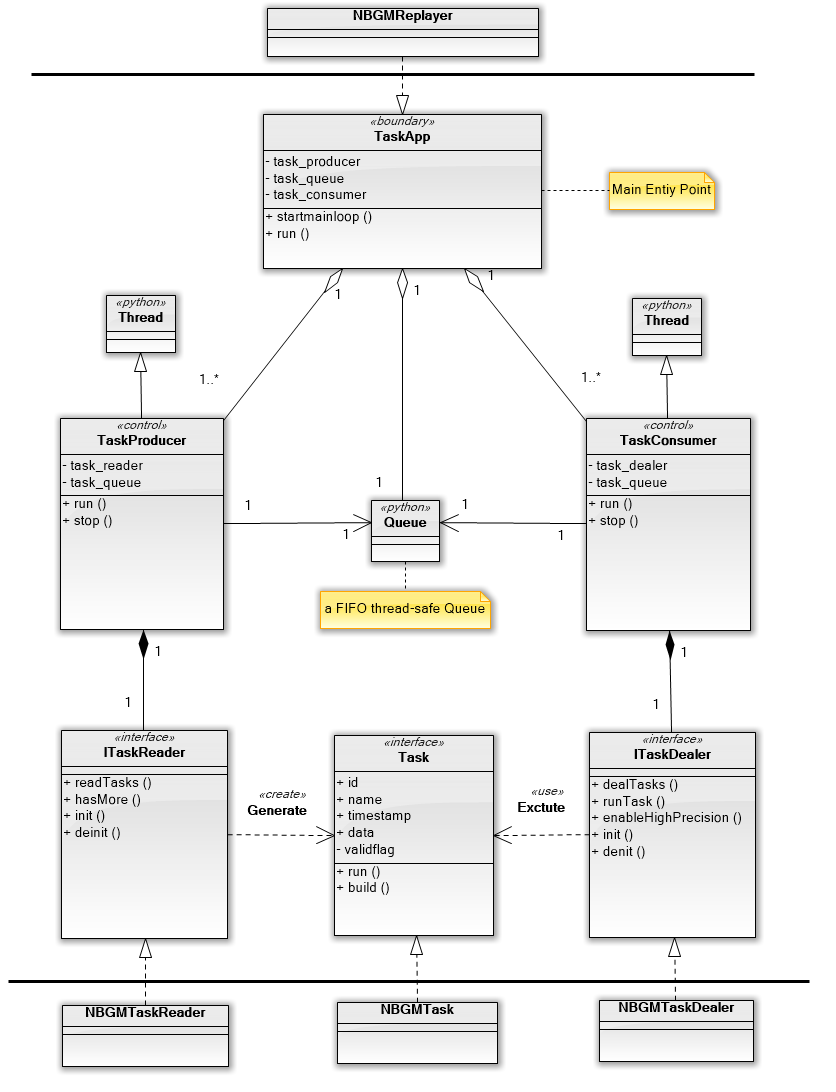
NBGM implementation of Core Layer

If necessary, you can make your own implementation for different requirements.

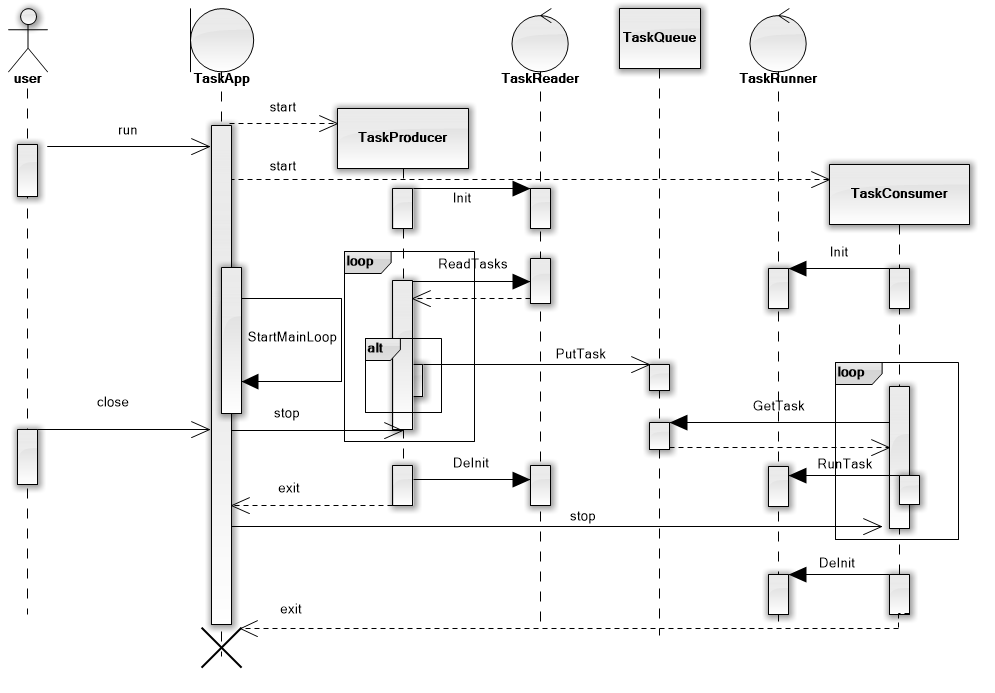
**1.1 System architecture diagram**



For a better understand of the structure of the system, class and time sequence diagram are given as follows.

**1.2 Class diagram**

**1.3 Time sequence diagram**



1. **Input File Format**

NBGM replay system needs three different input files to recreate a rendering scenario, they are

**.log** file - NBGM API invocation record, generated by NBGM app

**.xml** file - NBGM API argument configuration, maintained by user

**.dll** (so) file - NBGM dynamic library, depending on different platforms

* 1. **Invocation Record File**

The record file is made up of dynamic sized blocks which represent NBGM API invocations. The following table describes the format of each binary block.

|  |  |  |
| --- | --- | --- |
|  | **Size** | **Description** |
| Function Id | unsigned int | a unique identification of function |
| Time stamp | float | time of function invocation |
| Argument bin data length | unsigned int | size of argument bin data (in bytes) |
| Argument bin data | unsigned char[n] | contains all the arguments |

* Function Id

Instead of name, function id requires an extra map [id, func\_name or func\_ptr] to be maintained

* Time stamp

A timestamp represents seconds that have elapsed from some time, which is accurate to seconds or millisecond, depending on platform implement. Usually, we can take POSIX timestamp in the situation of demand on low precision.

* Argument data

A kind of packed data structure, containing binary content of each argument in sequence .

Little-endian order is supported for now, and there is no alignment.

* 1. **Function configuration File**

The configuration file take the form of XML, which needs update by the user if NBGM API changes. The update procedure could be auto generated by python script in future. The following table describes the format of each xml node.

|  |  |
| --- | --- |
| **id** | integer, should be unique |
| **name** | used to get function address from dynamic library |
| **args\_num** | the count of arguments, |
| **args\_format** | describes the data type of each argument |

* args\_format

| **Format** | **C Type** | **Python** | **Notes** |
| --- | --- | --- | --- |
| x | pad byte | no value |  |
| c | char | string of length 1 |  |
| b | signed char | integer |  |
| B | unsigned char | integer |  |
| ? | \_Bool | bool | (1) |
| h | short | integer |  |
| H | unsigned short | integer |  |
| i | int | integer |  |
| I | unsigned int | integer or long |  |
| l | long | integer |  |
| L | unsigned long | long |  |
| q | long long | long | (2) |
| Q | unsigned long long | long | (2) |
| f | float | float |  |
| d | double | float |  |
| s | char[] | string |  |
| p | char[] | string |  |
| P | void \* | long |  |

1. The '?' conversion code corresponds to the \_Bool type defined by C99. If this type is not available, it is simulated using a char.
2. The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, \_\_int64.
3. **Client Code**

When using this replay system, you must implement a series of interface first based on different business requirements, such as Task, ITaskReader and ITaskRunner.

One more class you need to pay attention, TaskApp. The TaskApp class is the base for creating task replay applications. Think of it as your main entry point into the run loop. In most cases, you create an instance of this class and implement your main loop, when you are ready to start the application's lifecycle, call your instance's run method.

