QAL Module Design

1. Revision History

Date	Version	Description	Author
5/22/2016	1	First draft	Brett Gilmer
5/24/2016	2	Review Comments	Brett Gilmer

2. Introduction

2.1 Overview

QAL Modules are classes which, when run, exercise a specific Port/Peripheral on a Device-Under-Test (DUT) in a specific way, as configured at the time of module construction. Each module responds to defined messages (e.g. Start/Stop/Report) which will perform thread control as well as return the current status of the port/peripheral being exercised.

2.3 Definitions

- **Device Under Test (DUT)**

  A Device Under Test is the Thales MPS. The QAL will run on the DUT.

- **QAL Test**

  A specific test pattern to be repeated on a port/peripheral. The test is not pass/fail, but will have some sort of status that is reported. For example, a serial port test might loop a specific string repeatedly, and report the number of errors received.

- **QAL Module**

  An implementation of a single QAL Test, where the Test is run in a background thread and where messages & Control are implemented in the main execution thread.

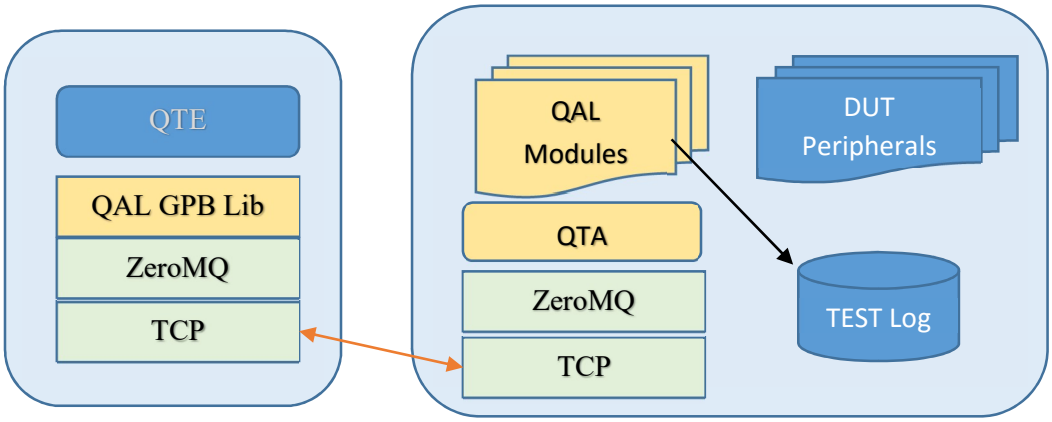
- **QTE**

  The QTE is an application running external to the MPS which originates control messages destined for QAL Modules by sending them to the QTA.

- **QTA**

  The Qual Test Application is the control framework for QAL Testing. QTA will discover all declared QAL modules and route messages from the QTE

3. Modular Overview



3.1 QTE

The design of QTE is out of the scope of this document. Functionally it will originate GPB-based requests and handle GPB-based responses.

3.2 QAL GPB Lib

This shall be a Windows DLL implementing the messages defined in the ICD. Required API is TBD

3.3 ZeroMQ

Overlays the ZMQ/0MQ implementation. Also contains the Thales-specific message wrappers for mapping requests to responses and encoding message types. Design is out of the scope of this document.

3.4 TCP

Standard platform networking. This is an assumption of the underlying ZMQ transport.

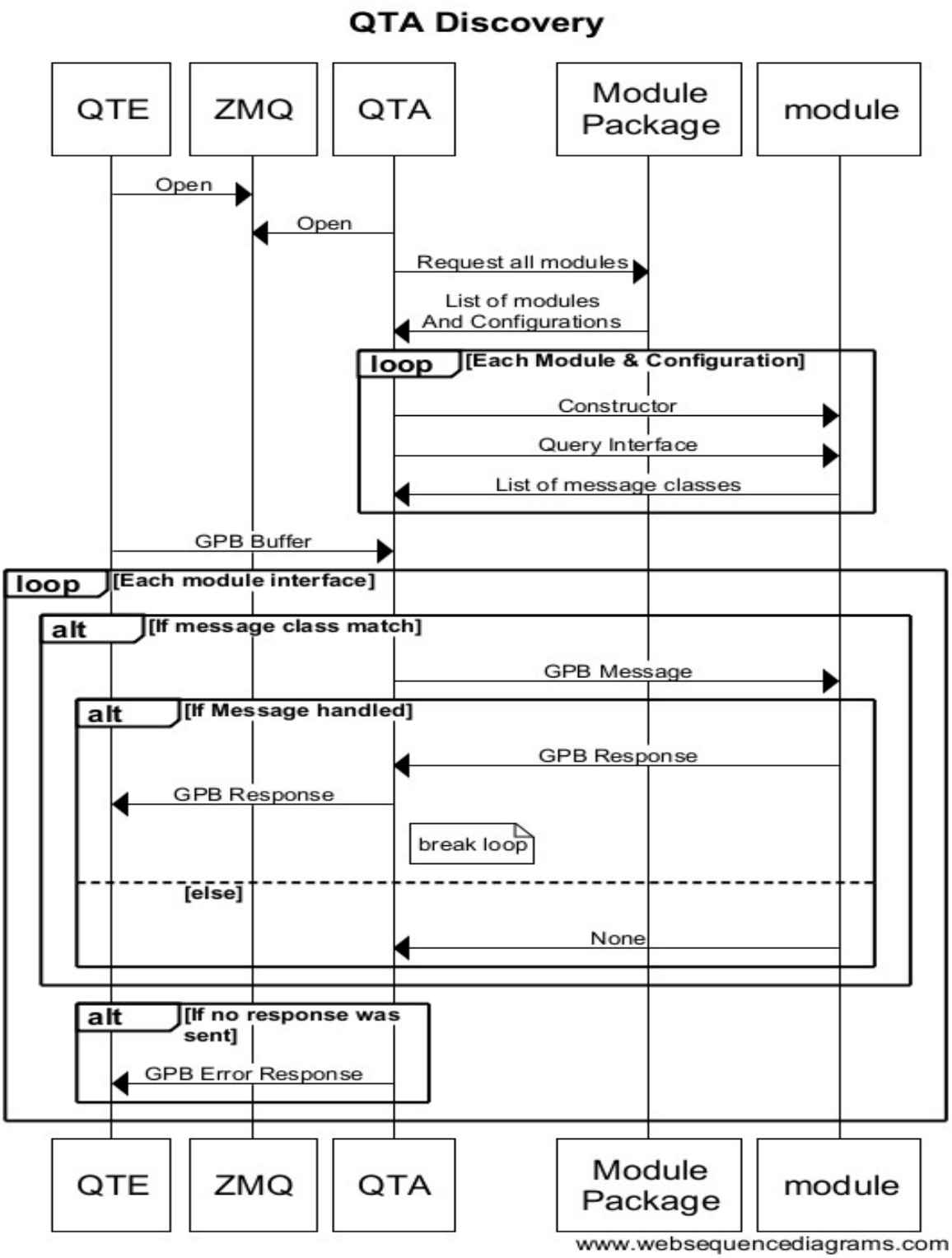
3.5 QTA

QTA will query all available modules and route GPB messages from the ZMQ to the modules that handle each message type for processing and then route the responses back to the ZMQ.

3.6 QAL Modules

The individual Test Modules that exercise specific DUT components and peripherals.

4. Message Overview



5. QAL Modules

Each module is implemented from a base class. Central to the base class is the Thread implementation and the message interface discovery service.

5.1 QTA Discovery Methods

5.1.1 GetConfigurations(cls)

Class method. It returns a List of Dictionaries, each dictionary containing the configuration details to be used in the constructor for class objects. For example, if a module tests serial port traffic, and there are 2 serial ports, this function might return a list of 2 dictionaries, each defining the port # and configuration for the ports

```
def getConfigurations(cls):
    return [
              {'port' : '/dev/com0', 'baud':19200, parity:'8N1'},
              {'port' : '/dev/com1', 'baud':9600, parity:'8N1'},
       ]
```

5.1.2 __init__(self, config)

Constructor. Passed a dictionary for configuration values, specific to the class (see GetConfigurations). In the constructor is a good place to add message handlers, connecting message classes to handlers.

```
def __init__(self, config={}):
    super(Example, self).__init__(config)
    self.addMsgHandler(StopMessage, self.stop)
    self.addMsgHandler(StartMessage, self.start)
    self.addMsgHandler(RequestReportMessage, self.report)
    self.addThread(self.runCounter)
```

5.1.3 selfName(self, name)

Sets the module name, visible from debug logs. Defaults to the class name. May be called after the superclass init.

```
def __init__(self, config={}):
    super(Example, self).__init__(config)
    self.setName('Foo')
```

5.1.4 terminate(self)

Terminates the module. Any child threads must be terminated, and any system cleanup performed (basically this is a destructor). Must be overwritten if any child threads are created outside of the build in threading mechanism.

5.2 Messaging Methods

5.2.1 addMsgHandler(msgClass, handler)

This function Registers a message Class and a handler that is able to process it. See <handler> below.

```
def __init__(self, config={}):
  super(Example, self).__init__(config)
  self.addMsgHandler(StopMessage, self.stop)
```

5.2.2 <handler>(self, msg)

Message handlers are defined in the module class, and registered to handle specific message classes in the __init__. A message handler receives a GPB message class as a parameter and must return a new GPB buffer as a response (if the message is handled) or None to indicate that this message was not handled by this instance of the module.

```
def stop(self, msg):
    status = StatusRequestMessage(self.counter)
    self.stopThread()
    return status
```

5.3 Logging Methods

5.3.1 log(self, text)

Logs a text string of data into the debug logs. These logs are local to the QAL system and used in debugging the test framework (not for reporting status to the QTE).

5.5 Thread Methods

Each module implements a child thread where the test execution shall take place. This thread shall update class member variables, using semaphore protection as needed, so that the registered message handlers will have the most recent status snapshot for GPB responses.

Advanced Usage ☺ : In The event that the module does not lend itself to the usage of the threading methods, it is acceptable for the Module to implement its own system. It is a requirement to terminate unused threads, and all threads on the self.terminate() function.

5.5.1 addThread(self, threadMain)

Registers an execution function for a thread that will be managed by startThread, stopThread. As many threads as are needed may be added with this method.

5.5.2 startThread(self)

The startThread() function is callable by the derived class and will start the execution threads (if not already started).

Usage in message handler

```
def start(self, msg):
    self.counter = 0
    self.interval = msg.interval
    self.startThread()
    status = StatusRequestMessage(self.counter)
    return status
```

5.5.3 stopThread(self)

The stopThread() function is callable by the derived class and will end the execution threads.

Usage in message handler

```
def stop(self, msg):
    status = StatusRequestMessage(self.counter)
    self.stopThread()
    return status
```

5.5.4 <runThread>(self)

Run is called in the main loop of the execution thread. It should perform module functions and return quickly so that the thread may terminate in a timely manner upon request and so that status messages will have updated information. (Target a return from Run every second?).

Run must call the superclass run() function during its execution.

If design is simplified, the run() may loop internally, but must monitor and return if the member variable self.__running is ever False. When self.__running is set to false, the thread will have a period of time (e.g. 2 seconds) to return gracefully, at which time an Alarm will be raised and an exception will abort the thread.

```
#Sample test increments a monotonic counter over an interval
def runCounter(self):
```

```
self.counter += 1
sleep(self.interval/1000.0)
self.log('counter now %d' % (self.counter))
super(Example, self).run()
Return
```