# PB 980 VTS Architecture and Code

10/30/2012

# Purpose

This document provides as-built information about the VTS Architecture within the context of the Covidien Gateway Remote Service. Historical design information is provided to facilitate understanding of the system evolution and code artifacts. PB 980 Ventilator components are also described herein (but should probably be moved into their own document).

# Related Documents

Theory of Operations - Covidien Gateway Remote Service

Interface Control Specification – VTS to RSA

# Overview

The VTS (Vent Test System) lives in the overall context of the Covidien Gateway Remote Service, which is composed of four primary components: PB980 Ventilator, VTS, RSA, Covidien Server. Consequently, the VTS design was built to fit the context in which it must operate: The following are primary constraints / considerations.

1. All PB 980s currently use exactly the same MAC address, have a hard-coded IP address of 192.168.0.10, and use a subnet mask of 255.255.0.0.
2. Typical laptop computers only have a single network interface card (NIC) and the system should be fully functional on such a laptop.

These constraints require the overall system to be able to connect to the server, cache information, disconnect from the server, connect to the ventilator, apply cached server information, gather and cache information from the ventilator, disconnect from the ventilator, connect to the server, and share the cached information back to the server.

To meet these key constraints, the RSA was designed following the iMAP model, for email store-and-forward. This isolates and decouples the complexity of knowing when a server connection was available and caching data while simultaneously providing a reusable component for the development of VTS-like components for other devices. In concert with the chosen store-and-forward mechanism was the choice to use the publish / subscribe model for messaging.

To the extent possible, it is intended that the RSA be an invisible pass-through of messages (e.g., notifications). Communications with the RSA to manage synchronization between the VTS and RSA were to be done through more direct messaging. It is unclear why an RPC-style messaging was not chosen for synchronization, except perhaps to maintain a level of consistency in the messaging. An implementation choice to write the RSA in Java further enforced the messaging behavior.

The messaging is inherently asynchronous in nature. Neither side has found it beneficial to impose flow control Detailed information about the messages may be found in the Interface Control Specification – VTS to RSA.

Additionally, the VTS was designed with the understanding that there would be additional devices to support in the future.

# Original Design

The following is an early system design. It should be noted that the mailbox functionality ascribed to the “PC Agent Interface” (aka VTS) is actually implemented within the “Laptop Agent” (aka RSA).



# Breakdown

## **PB980 (Vent Side) Interface**



### **Webserver**

Express logic has provided a webserver with ThreadX, the operating system on which the PB980 is running.

### **LogWebservice**

The LogWebservice is intended to provide a restful API, allowing for log data to be retrieved in binary and XML format, containing human readable log entries. For each format, the LogWebservice will be required to embed metadata such as serial number and ventilator software version.

Known Design Challenges

* The LogWebservice will not be able to give a full log in one shot because this will require a significant amount of memory. The restful API will need to support the concept of cursers/ iterators.
* The native PB980 logs are in binary format. Each log has its own format.

In a first pass, logs were communicated in XML structures composed of a contextual header, including mime-type style and encoding information, plus a body which was a base64 encoding of the binary log. These were decoded externally based on an XML description of the log file mime-type.

Due to concerns about the one-third size increase of base64 encoding for large logs on the ventilator size, the vent team decided that the logs would be passed plainly (i.e., no contextual information) via an FTP service.

Meanwhile it was discovered that for space efficiency, the ventilator was designed with a string table and the logs included indexed references to the string table. This threw a wrinkle into the external decoding of binary logs. The trivial solution of building the string table on the VTS side was discarded for fears of configuration management. The next “simple” solution of building a DLL of the log presentation code on the VTS side seems reasonable, but it was believed that the ventilator code was too entangled to easily carve out.

The current decision is to perform log decoding on the ventilator, passing it back as an XML structure. The XML structure contains a small context header and a body which is a collection of records. Each record is broken into its component elements. An XSLT file is then capable of converting the XML into a presentable log.

### **DeviceInfoWebservice**

The DeviceInfoWebservice is intended to provide a restful API, allowing for device details to be retrieved in an XML format. This includes device identity information (type, make, model, serial number, and release), plus a collection of information for each internally managed component (e.g., hardware, software, firmware). The vent specific options and data key information are held in the component called “MasterVent”.

### **Download Trigger Interface**

The Download Trigger interface has already been implemented. There is work required to add more security around how a download is triggered. Currently there are some concerns about the weakness or lack of security around triggering a download via the Ethernet. Additionally, the trigger message should contain a hash of the ventilator serial number to ensure that the correct device is connected. Otherwise, the interface risks being retired.

### **Download Manifest Security**

The download manifest does not add any security to prevent a user from downloading software to the ventilator. It is desirable to require a download key in the manifest prior to allowing a download to proceed.

Known Design Challenges

* The downloader is not able to access the serial number of the vent when in download mode. This is because the serial number is stored on a USB storage device, on the GUI CPU, which is not available to the downloader.
* Even if the serial number makes its way to flash, the corner case, were the GUI/BD CPU is replaced, presents a challenge.

### **Download Status**

Currently the download status reported by the ventilator only gives the percentage complete per phase of installation per file. It is desirable to also have an overall percentage complete for the whole installation.

## **PB980 (PC Side) Interface**



### **Abstract File System Layer**

The Abstract file system layer is intended to allow for a virtual mapping of files based on a URL and a download manifest provided by the laptop agent.

### **Download FS Interface**

The Download FS Interface provides an interface, allowing for the mappings in the Abstract File System Layer to be configured by a download notification.

### **Get Log Http Webclient & Interface**

The Get Log Http Webclient & Interface is intended to provide the facilities needed to get log data. Currently there is no infrastructure in place to get logs from the ventilator.

### **Get DeviceInfo Http Webclient & Interface**

The DeviceInfo Http Webclient & Interface is intended to provide the facilities needed to get device identifying information from the ventilator. Currently there is no infrastructure in place to get device information from the ventilator.

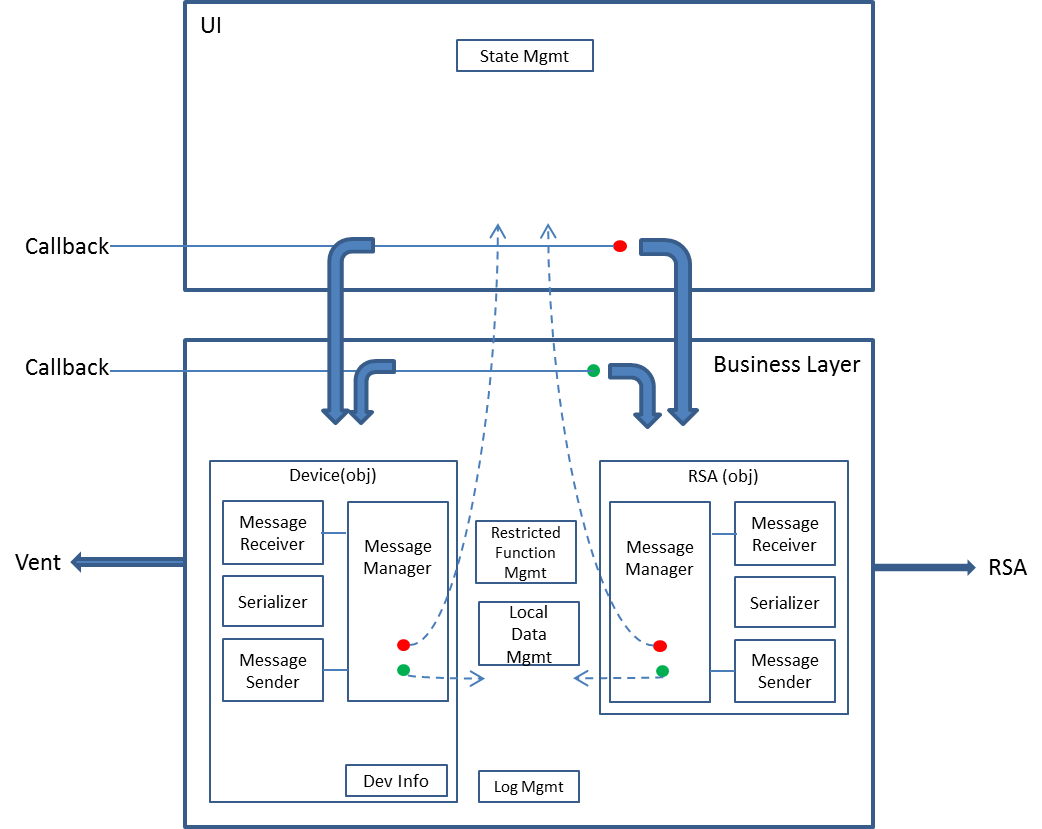
## **RSA**

The RSA is documented elsewhere. It is mentioned here only as a reminder that it is a windows service, to be running all the time. When VTS starts, the first thing it attempts to do is to connect to the RSA. If the RSA service is not running, VTS will so indicate. In such a case, the user will need to terminate VTS, then manually start the RSA and restart VTS.

## **VTS**

The VTS is something of a multi-threaded state machine as it simultaneously handles user, ventilator, and RSA interactions. As mentioned previously, the RSA is intended to be a simple pass-through to the server when the server is connected, which creates a delay in messages and responses to the messages. To handle this, most interactions are initiated by the user (some merely by the state), with an associated callback to react to the asynchronous response.

The following image indicates how the VTS is built. As is indicated by the red and green callback “marbles”, these are passed with a request, typically a query to the RSA or the device. The callbacks are invoked on a corresponding response.



### **User Interface (VTS segment of Project)**

### **Comms**

All the messages and functionality for serializing and de-serializing are contained here. For message details, refer to the Interface Control Specification – VTS to RSA

### **IPI\_CORE**

This segment of the code provides a variety of utility functions from IPI which may not all be used.

### **Business Layer**

At the business layer, an attempt was made to have the Device (Vent) connection and the RSA connection be derived from the same base class to handle the basic communications identically (to the extent possible). These were overlaid with a “BusinessServicesBridge” to allow the UI a single interface point. Ultimately, in the short-time span and high requirements churn environment, the well intentioned plans broke down. The following sections describe the key elements or functionality in the BusinessServicesBridge.

#### Callback Model

Most of the interactions with the RSA are inherently asynchronous, requiring the use of a callback when a response is finally available. As mentioned, we attempted to handle the device interactions in the same way. These interactions follow a basic pattern.

Usually two identically named methods are provided to make the initial call (e.g., CreateDevice). One method takes the basic parameters while the second take the same basic parameters and also includes a parameter for the callback function. Typically the method without a callback function simply turns around and calls the other, passing null for the callback.

The method will usually generate a transaction id, encode that in its request message, hand off the request to a “send buffer” (which will place the message on a socket to the RSA), store the callback, if any in a hash table, indexed by the transaction id, and return the transaction id to the user.

When the response is received, a lookup for the associated request is made, and if a callback was specified, it is invoked. If no callback was specified then the response is stored in a hash table indexed by the transaction id.

A third method is provided, usually named Get<method>Result which takes the transaction id and returns the stored response. This is only useful when the callback is not originally provided. Currently, this third method is not being used.

#### Instance

The BusinessServicesBridge is a singleton, accessed through the static member Instance.

Rather than create a duplicative layer (at this point) to fully isolate the interactions with the ventilator and the RSA, the internal objects are exposed. However, many functions are still only accessed through the bridge as there is often additional business logic involved.

#### RSA (User) Interface Services

Provides access to the RSA

#### Device (User) Interface Services

Provides access to the device functions.

#### VTS Status Messages

The business layer holds four delegates to interact with the UI Status Messages. These are:

AddServerStatusMessage

RemoveServerStatusMessage

AddDeviceStatusMessage

RemoveDeviceStatusMessage

The “server” status messages actually apply to both RSA status and the Server status. These are displayed toward the bottom right of the User Interface. Such messages are generally called when we are going to perform a task known to take some time, such as getting device information from the RSA (or server). When the associated callback is invoked, the status messages are removed (if appropriate).

The “device” status messages apply to the interaction with the ventilator. Because most ventilator interaction is synchronous (e.g., webservice call), the status messages often come and go without being seen. Ultimately it is a race condition between thread swapping that would allow these messages to be seen by the user.

#### Restricted Function Manager

The RestrictedFunctionManager coordinates access rights provided by the device with those reported by the server for the user of the device. The entire concept is not well thought out and should be expected to evolve. Currently, there are two restricted functions: log viewing and software downloads. From the ventilator side, simply having the SERVICE\_KEY (as an “option” in the “MasterVent” component of the device info) is enough to provide the user access to both of these capabilities. If access is denied by the ventilator, it can be overridden by access rights provided by the server, by having the “install” right (as a right in the “MasterVent” component of the device info returned from the server).

A separate accounting is managed for each device, by serial number. To determine if a capability is allowed, the IsFunctionRestricted method is called. If for some reason information specific to a single source is desired, the IsFunctionSourceRestricted is available.

PendingNotifications is a related component which holds on to notification headers for which we choose NOT to get the associated notifications because the user is not fully logged-in. When the user is fully logged in or restrictions have changed, then the method RestrictionsChanged and any pending notifications that are now allowed will be requested.

### **Connectivity**

A attempt was made, as mentioned previously, to make the device and RSA behavior patterns mirror images, at least to the extent that is meaningful. Consequently there is a ConnectToRSA method and a ConnectToDevice method. Both methods expect callbacks as parameters in order to notify the UI when the activity is complete.

Each method follows its own set of internal state changes, attempting as much as possible to hide business layer logic from the UI. For example, the connecting to the RSA also results in creating a session and an anonymous login. Connecting to the device performs serveral additional actions, such as getting device information, checking with the RSA for knowledge about the device, storing the device info, etc.



### **Log Management**

As VTS connects to the device (i.e., vent), several things happen. First, old device connectivity information is cleared. (It is assumed that a bio-med may work on several machines sequentially). Because the interface to the device is webservice based, no permanent connection is made. Instead we simply ask for the device information and treat success as having made a “connection” for purposes of communicating with the user. The device info returned is effectively converted into a log (from the user perspective), using XSLT. Additional activity occurs with respect to interacting with the RSA, and when completed, VTS asks the device for a list of logs.

When the list is returned, VTS schedules (currently at a ¼ second interval), getting each of the logs. As each log is retrieved, it is checked to be either a common diagnostic log or an extra log. The diagnostic logs are automatically stored to the RSA in raw form, decoded, and stored in a decoded form.

When the user views the log, they have the ability to right-click in the embedded browser and print.

### **Log Decoding Functionality**

The original log decoder was a general purpose binary log decoder. Its code is primarily kept in the BinaryLogDecoder segment of the project. This is expected to be deprecated shortly and so is not discussed further.

The current plan for the logs are for them to be decoded at the ventilator, placing all relevant data in an appropriate XML structure, which would look something like:

<?xml version="1.0"?>

<log>

<header>

<version>VIKING\_3.1.3.146</version>

<type>System Diagnostic</type>

<datetime>4/17/2008 22:30:52</datetime>

<serialnumber>35B12P3044</serialnumber>

</header>

<records>

      <record>

<col name="Date" type="DateTime"> 5/1/2008 8:30:52</col>

<col name="Test/Event"    id="2568">Alarm Silence</col>

<col name="Code"                 type="DiagnosticCode" >User</col>

<col name="Type"                  id="2608">User</col>

<col name="Notes"                id="2608">User</col>

      </record>

</records>

</log>

As can be seen, the log is an XML structure. This log is then converted into an HTML presentation to the user through the use of XSLT. The relevant XSLT files are contained under VTS/SourceDefinitions/ directory.

XsltToTable.xslt is a generic xml to table generator.

XsltToTableCC.v2.xslt is an adaptation which supports the header and the records with the col tags.

When this change is ready to be integrated, see the DeviceInfo as a log behavior for how to attach the XSLT.

### **Debug vs Production**

The main challenge with hiding communications behind GUIDs (e.g., where the GUID identifies the function) is that it can be really hard to debug. The config file can set a debug flag and the ModalConstants provides the facility to convert the coded variable into either the debug or runtime variable as appropriate.

From a coding standpoint, most of the code references the human readable variable and invokes the ModalConstants method to ensure it is of the correct debug or runtime flavor.

### **Work List aka Pre-Load**

Dues to the network limitations of the vent, a bio-med user may want to make sure he has all the information he needs for the day’s work when he is connected to a regular network. To do this, he would go to the “work list” button and then type in vent serial numbers one by one until all the associated messages (notifications) have been loaded.

This process is very similar to what happens when we connect directly to a vent, but the results are stored a different location, DeviceDataFromServer. See functions GetDeviceData<xxx>.

### **Software Update**

When the bio-med goes to the software update action, they are given a choice (at the UI) to load a local file or (if a notification has been loaded from the RSA indicating a software download) available software downloads. Such downloads may include documents to view.

When the given download is invoked, the separate DLL, called DownloadShell\_Net, built from Michael’s environment somewhere, is set up with progress update callbacks and is invoked.

### **Session Manager & Manager**

The Session Manager & Interface is intended to provide a means to establishing a session and logging into the Laptop Agent. Also, the session manager will associate a device to the system and initiate the process of downloading notifications for the device.

The session manager interface will provide an API to login, logoff and create a device. The calling process will have the option of synchronous/asynchronous behavior. The asynchronous behavior will require the calling process to provide a delegate (function pointer and object).

### **Event Manager**

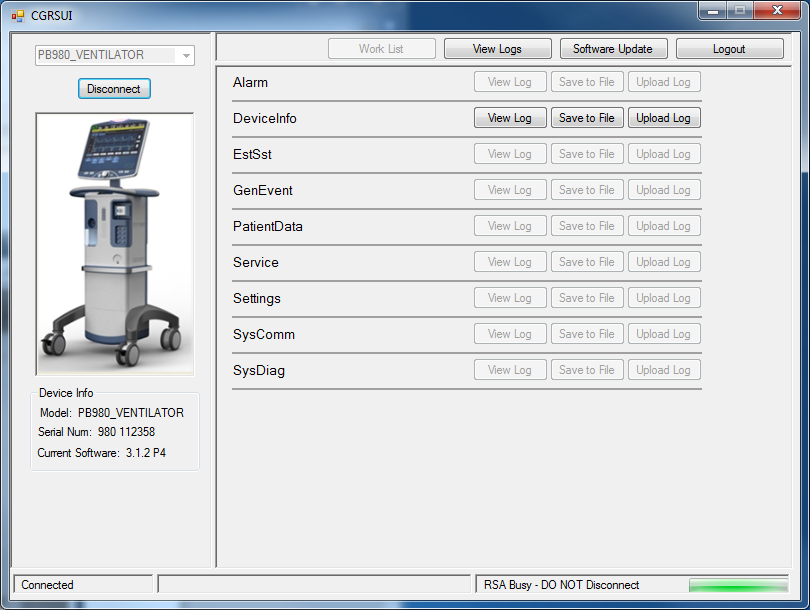
The Eventmanager is intended to track outgoing requests and incoming responses. When a request is made, the calling class registers a callback for the response and sets a timeout. When the response arrives, the event manager calls the callback notifying the object of the event.

### **Message Receiver & Sender**

Both the message sender and receiver are dedicated to the task of communicating with the laptop agent.

## User Interface

### Solutions Center





The Solutions Center will provide the functionality required to login, view notifications, start the software update application, Log Viewer application and read release notes. A user will have the ability to double click a download notification to initiate a download.

### Log Viewer



The Log Viewer Application will provide the functionality required to retrieve and view ventilator logs. Export the logs to a file. Load and view an Exported logfile’s.

### Software Update Application



There is already a UI developed for the downloader. The UI was intended only to provide manufacturing a means of downloading software and is considered a temporary implementation.

Below is the current implementation.

