Ventilator

Logs

Software

Design

Specification (SDS)

900 Series

**Table of Contents**

[1 Revision History 5](#_Toc309633459)

[2 Introduction 6](#_Toc309633460)

[2.1 Objective 6](#_Toc309633461)

[2.2 Scope 6](#_Toc309633462)

[2.3 Reference Documents 6](#_Toc309633463)

[2.4 Definitions 7](#_Toc309633464)

[3 Ventilator Logging System 8](#_Toc309633465)

[3.1 Recording Media 8](#_Toc309633466)

[3.1.1 Overview 8](#_Toc309633467)

[3.1.2 Hardware Interface 8](#_Toc309633468)

[3.1.3 File System Interface 8](#_Toc309633469)

[3.1.3.1 Target Considerations 8](#_Toc309633470)

[3.1.3.2 FileX Implementation 9](#_Toc309633471)

[3.1.3.3 FileX Options 9](#_Toc309633472)

[3.1.3.4 Media Interface 11](#_Toc309633473)

[3.1.3.4.1 Media Description 11](#_Toc309633474)

[3.1.3.4.2 Media Format 11](#_Toc309633475)

[3.1.3.4.3 Media Open 11](#_Toc309633476)

[3.1.3.4.4 Media System Date and Time 11](#_Toc309633477)

[3.1.3.4.5 Media Read and Write 12](#_Toc309633478)

[3.1.3.4.6 Media Flush 12](#_Toc309633479)

[3.1.3.4.7 Media Close 12](#_Toc309633480)

[3.1.3.4.8 Media Abort 12](#_Toc309633481)

[3.1.3.4.9 Media Check 12](#_Toc309633482)

[3.1.3.4.10 Media Space Available 12](#_Toc309633483)

[3.1.3.4.11 Media Volume 12](#_Toc309633484)

[3.1.3.4.12 Media Test Environment 13](#_Toc309633485)

[3.1.3.4.13 Media Test Equipment 13](#_Toc309633486)

[3.1.3.5 Directory I/O Functions 13](#_Toc309633487)

[3.1.3.6 File I/O Functions 14](#_Toc309633488)

[3.2 Ventilator File Interface 15](#_Toc309633489)

[3.2.1 Media Access 15](#_Toc309633490)

[3.2.2 Media Bandwidth 15](#_Toc309633491)

[3.3 Ventilator Directories 16](#_Toc309633492)

[3.3.1 Overview 16](#_Toc309633493)

[3.3.2 Directory Structure 16](#_Toc309633494)

[3.3.2.1 General User Accessible Directories 16](#_Toc309633495)

[3.3.2.2 Secure Directories 17](#_Toc309633496)

[3.3.2.2.1 Secure Field Directory 18](#_Toc309633497)

[3.3.2.2.2 Secure Covidien Directory 18](#_Toc309633498)

[3.3.3 File System Access Protection 18](#_Toc309633499)

[3.3.4 Log Database 19](#_Toc309633500)

[3.3.4.3 System Development 19](#_Toc309633501)

[3.3.4.4 Production Usage 20](#_Toc309633502)

[3.4 Ventilator Log Definitions 21](#_Toc309633503)

[3.4.1 Base Classes 21](#_Toc309633504)

[3.4.2 Patient Data Log 22](#_Toc309633505)

[3.4.3 Alarm Log 24](#_Toc309633506)

[3.4.4 System Diagnostic Logs 25](#_Toc309633507)

[3.4.4.1 System Diagnostic Log 26](#_Toc309633508)

[3.4.4.2 System Information Diagnostic Log 26](#_Toc309633509)

[3.4.4.3 EST/SST Diagnostic Log 27](#_Toc309633510)

[3.4.5 Operator Settings Log [P2.2 Schedule] 27](#_Toc309633511)

[3.4.6 Alarm Event Log 31](#_Toc309633512)

[3.4.7 Extended Self Test (EST) Log 34](#_Toc309633513)

[3.4.8 Short Self Test (SST) Log 35](#_Toc309633514)

[3.4.9 Calibration Log [Not on P2.2 Schedule] 36](#_Toc309633515)

[3.4.10 General Event Log [P2.2 Schedule] 38](#_Toc309633516)

[3.4.11 SPI Backplane Event Log [Not on P2.2 Schedule] 44](#_Toc309633517)

[3.4.13 Service Log [P2.2 Schedule] 48](#_Toc309633518)

[3.4.14 Preventative Maintenance Log [Not on P2.2 Schedule] 50](#_Toc309633519)

[3.4.15 Device Alert Log [Not on P2.2 Schedule] 52](#_Toc309633520)

[3.4.16 Data Storage Summaries 55](#_Toc309633521)

[3.4.16.1 Data Storage Summary 55](#_Toc309633522)

[3.4.16.2 Data Storage Bandwidth 56](#_Toc309633523)

[3.4.16.3 Data Retrieval Bandwidth 57](#_Toc309633524)

[3.4.16.4 Data Storage / Retrieval Combined Bandwidth 57](#_Toc309633525)

[3.5 Ventilator Logs Access and Use 58](#_Toc309633526)

[3.5.1 Normal Mode 58](#_Toc309633527)

[3.5.2 Service Mode 58](#_Toc309633528)

[3.5.2.1 Directory Access 58](#_Toc309633529)

[3.5.2.2 File Display 58](#_Toc309633530)

[4 New Details 59](#_Toc309633531)

[5 Flash Implementation 59](#_Toc309633532)

[6 Database Initialization 60](#_Toc309633533)

[6.1 System Startup 60](#_Toc309633534)

[6.1.1 New Logs Database 60](#_Toc309633535)

[6.1.2 Existing Logs Database – Verification 60](#_Toc309633536)

[6.1.3 Existing Logs Database – Repair 60](#_Toc309633537)

[6.2 Database Verification Process 61](#_Toc309633538)

[7 Questions 62](#_Toc309633539)

[8 Performing Unit Tests 63](#_Toc309633540)

[8.1 Enable Unit Tests 63](#_Toc309633541)

[8.2 Create Unit Test 63](#_Toc309633542)

[8.3 Using Omnitool 64](#_Toc309633543)

[9 Final Files 65](#_Toc309633544)

[10 Questions 66](#_Toc309633545)

[10.1 Setting Log 66](#_Toc309633546)

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Author** | **Change Description** |
| 1 | 1/28/2011 | Stephen McClure | Initial revision. |
| 1b5 | 11/21/2011 | Hari Damineni | Updated log record data structure definitions |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Introduction

## Objective

To specify the Software Design Specifications for the 900 Series Ventilator Logging System.

## Scope

This document is applicable to the entire ventilator product.

## Reference Documents

|  |  |  |  |
| --- | --- | --- | --- |
| **Reference** | **Part Number** | **Revision** | **Document Title** |
| 1 | MPC8349EARM | Rev. 1  9/2006 | MPC8349EA PowerQUICC II Pro Integrated Host Processor Family Reference Manual. |
| 2 |  |  | Ventilator Logs  Software Requirement Specification (SRS)  900 Series, Covidien |
| 3 | 000-1010 | Rev. 5.0 | Express Logic FileX User Guide |
| 4 |  |  | Viking Alarm Analysis SRS |
|  |  |  |  |

## Definitions

|  |  |
| --- | --- |
| **Abbreviation** | **Description** |
|  |  |
| BD | Breath Delivery |
|  |  |
| EST | Extended Self Test |
| FileX | Express Logix File System API |
| FPGA | Field Programmable Gate Array |
| GUI | Graphics User Interface |
|  |  |
| LSb | Least Significant bit |
| LSB | Least Significant Byte |
| Mb | Megabit |
| MB | Megabyte |
| MSb | Most Significant bit |
| MSB | Most Significant Byte |
| MPC8347EA | PowerPC Processor used by the BD and GUI boards |
| PCBA | Printed Circuit Board Assembly |
| PRD | Product Requirement Document |
| SDS | Software Design Specification |
| SRS | Software Requirement Specification |
| SST | Short Self Test |

# Ventilator Logging System

## Recording Media

### Overview

The SD Memory Card is the physical recording media which is used to store all ventilator logs. The High Capacity version of the SD Memory Card (SDHC) is utilized which provides a FAT32 storage system.

### Hardware Interface

The SDHC Memory Card resides on the GUI PCBA and is interfaced to the PowerPC via the Bridge/Router FPGA. An interface driver handles all the low level SD Memory Card initialization and communication between the physical card itself (the physical layer) and the File System interface.

### File System Interface

The File System interface is provided by the Express Logic FileX system component. FileX is directly interfaced through the SD Memory Card interface driver to the physical SDHC Memory Card. The high-level application is oblivious of this low-level driver interface and communicates with the SDHC Memory Card via the provided FileX system calls which treat the memory region as a FAT32 file system. The FileX API function calls provide the application with the ability to create files, open files, read and/or write file data blocks and to delete files.

#### Target Considerations

The FileX target considerations are minimal and are as follows:

1. FileX requires up to 30KB of application ROM space.
2. FileX requires 100 bytes of the target’s RAM space.
3. Each opened media requires 1.5KB RAM for the Control Block.
4. Each opened media required 512 bytes for one sector.

**Possible Target Considerations**

The SD Memory Card utilizes internal NAND memory. NAND memory usually has a Block consisting of several pages where each page contains several sectors. A page may be erased which erases all the sectors contained within the page. It is advantageous to make the media sector the same size as a NAND page – failure to do so implies that the writing of a single sector results in the possible erasing of a page followed by the copying of the page sectors which were not changed, followed by the writing of the changed sector, followed by a page programming operation.

Be aware of the SD Memory Card’s suggested erase page size and use that size as the buffer size.

#### FileX Implementation

When using FileX the following must be applied:

1. The application code that use FileX services or data structures must include fx\_api.h
2. The application code must link with fx.a (or fx.lib)
3. The application code must call fx\_system\_initialize() to initialize the FileX system.
4. The application code must call fx\_media\_open() to setup the FileX media.

#### FileX Options

The FileX API provides various configuration options. The following options are defined and utilized as shown below.

**FX\_DISABLE\_ERROR\_CHECKING**

This option will initially be disabled (ie. error checking active) until the logging application has been verified.

**FX\_MAX\_LONG\_NAME\_LEN**

In order to permit long file names, additional directory entries are added prior to the normal file entry. This can slow down the search for a specific directory or file name.

This option is set to 255.

**FX\_MAX\_SECTOR\_CACHE**

The cache size is related to the amount of memory supplied by the fx\_media\_open() API call.

This value is set to 16. (What max value should be used???).

**FX\_FAT\_MAP\_SIZE**

The larger the value, the fewer the updates of the secondary FAT sectors.

This value is set to 8.

**FX\_MAX\_FAT\_CACHE**

This value specifies the number of entries in the internal FATH cache.

This value is set to 8.

**FX\_FAULT\_TOLERANT**

When defined, FileX will update the media system sectors (boot, FAT, directory) whenever they are modified. This feature is not required since the Ventilator has a battery backup capability and knowledge of a power-down condition is well-known in advance.

**FX\_FAULT\_TOLERANT\_DATA**

When defined, FileX will update the media data sectors whenever modified. This feature is not required since the Ventilator has a battery backup capability and knowledge of a power-down condition is well-known in advance.

**FX\_NO\_LOCAL\_PATH**

This feature is not required. Implementation is utilized to reduce the code size but this is unnecessary with the ventilator application.

**FX\_NO\_TIMER**

This feature is not required. FileX system date and time will be utilized.

**FX\_UPDATE\_RATE\_IN\_SECONDS**

This option is set to 1 implying that the FileX system time will be updated every second.

**FX\_UPDATE\_RATE\_IN\_TICKS**

This option is not required.

**FX\_SINGLE\_THREAD**

This option is not required – the FileX API may be used by multiple ThreadX tasks simultaneously.

#### Media Interface

##### Media Description

The SDHC Memory Card provides its internal memory as an array of logical sectors, each of 512 bytes in size. The SD card has built-in defect and error management, (which are completely transparent to the host), along with wear-leveling capabilities.

##### Media Format

Prior to using a blank SDHC Memory Card, the card must be formatted by using the FileX fx\_media\_format() API call.

In order to detect a new SDHC un-formatted card versus a formatted card we could attempt to perform a Media Open function first and if it were to fail, then proceed to format the Memory Card (with the user’s permission)..

**Questions**

Ensure that we can use an SDHC Memory Card which has been formatted on a PC.

##### Media Open

The Media must be opened prior to being accessed. In order to achieve this the FileX fx\_media\_open() API call is used.

##### Media System Date and Time

To have an accurate system date and time, the application must call the following functions upon initialization:

1. fx\_system\_date\_set()
2. fx\_system\_time\_set()

Afterwards, the following functions may be used to obtain the date and time:

1. fx\_system\_date\_get()
2. fx\_system\_time\_get()

##### Media Read and Write

The Media logical sectors may be accessed by the following functions:

fx\_media\_read Read directly from a logical sector

fx\_media\_write Write directly to a logical sector

##### Media Flush

A Media Flush operation may be periodically performed in order to reduce the risk of file corruption and data loss due to a loss of power. This possibility is greatly reduced since the ventilator has a backup-battery and can thereby implement a controlled shut-down procedure.

##### Media Close

When the ventilator is to be powered down, a Media Close operation (API fx\_media\_close() function call) is performed which flushes buffers pertaining to current open files.

##### Media Abort

The Media access may be aborted when an I/O error is detected. In order to achieve this the FileX fx\_media\_abort() API call is used. Afterwards the media must be reopened prior to use.

##### Media Check

The Media Check function (fx\_media\_check() API call) may be used to verify and attempt to correct media structure errors.

##### Media Space Available

The Media Space Available function (fx\_media\_space\_available() API call) may be used in order to determine the number of bytes available in the media.

##### Media Volume

The Media Volume name may be set/read by the following functions:

fx\_media\_volume\_set Set the media volume name

fx\_media\_volume\_get Get the media volume name

##### Media Test Environment

V-PBMon will be extended to provide a suite of tests which will permit the basic FileX operations used by the logging system to be verified .

The Ventilator Log functions may be verified by executing them under the V-PBMon environment (or the GUI environment) using a RAM Disk until the SD Memory Card driver interface has been created. Changing media from a RAM disk to a SD Memory Card would then be a minor issue.

##### Media Test Equipment

The following equipment will be required:

1. GUI PCBA
2. SDHC Memory Card
3. FPGA / SDHC Memory Card Driver
4. SDHC Memory Card Interface to PC to verify FileX FAT32 implementation

#### Directory I/O Functions

The FileX API provides the following principal Directory functions:

fx\_directory\_create Create a new directory (no limit on # sub-dir files)

fx\_directory\_name\_test Test that a supplied name is an actual directory

fx\_directory\_delete Delete an existing (empty) directory

fx\_directory\_default\_set Set this as the default directory

fx\_directory\_default\_get Get last default directory

**We need these local path functions only if FileX is being accessed by multiple threads:**

fx\_directory\_local\_path\_set Set the current local path string

fx\_directory\_local\_path\_get Get the current local path string

fx\_directory\_local\_path\_clear Clear the previously set local path directory

fx\_directory\_local\_path\_restore Restore the previous local path

**If not using local path then must protect task access to the following directory access functions with ThreadX sema or priority:**

fx\_directory\_first\_entry\_find Get first entry name in the default directory

fx\_directory\_first\_full\_entry\_find Get first full entry name in the default directory

fx\_directory\_next\_entry\_find Get next entry name in the default directory

fx\_directory\_next\_full\_entry\_find Get next full entry name in the default directory

fx\_directory\_rename Rename a directory

fx\_directory\_short\_name\_get Get the short name from the specified long name

fx\_directory\_long\_name\_get Get the long name for the specified short name

fx\_directory\_information\_get Get directory entry information

fx\_directory\_attributes\_set Set the directory attributes

fx\_directory\_attributes\_read Read the directory attributes

#### File I/O Functions

The FileX API provides the following principal File I/O functions:

fx\_file\_allocate Extend file by specified size

fx\_file\_best\_effort\_allocate Extend file by specified size (or largest available)

fx\_file\_create Create a file of zero length

fx\_file\_open Open a file for reading or writing

fx\_file\_read Read bytes from a file

fx\_file\_relative\_seek Position file pointer to relative byte offset

fx\_file\_rename Rename a file (file must be closed)

fx\_file\_seek Move file pointer to specified byte offset

fx\_file\_truncate Truncate a file (excess clusters are not released)

fx\_file\_truncate\_release Truncate a file and release file clusters

fx\_file\_write Write bytes to a file

fx\_file\_close Close a file

fx\_file\_delete Delete a file

fx\_file\_date\_time\_set Set file date and time

fx\_file\_attributes\_set Set the file attributes

fx\_file\_attributes\_read Read the file attributes

**Note:**

Problems may arise when a file is opened by two tasks, one for reading and one for writing. The records of the file which has been opened for writing may not be seen by the reader unless the reader closes and then reopens the file for reading.

Care should also be taken with the file which has been opered for writing if the file is to be truncated since readers of the same file could return invalid data.

## Ventilator File Interface

This section discusses the manner in which the FileX API functions will be used to access the SD Memory Card.

### Media Access

When the SD Memory Card media is accessed, the following steps are taken such that it may be accessed via the FileX interface:

fx\_media\_format Must be called prior to opening the media.

fx\_media\_open Must be called to open media for read/write access.

### Media Bandwidth

The SD Card Memory uses a 20MHz clock therefore a maximum 10MByte/sec bandwidth may be achieved.

It must be noted that reading and writing operations may be occurring simultaneously which will lower single direction bandwith expectations.

## Ventilator Directories

### Overview

The ventilator logs are data files which all reside on the GUI SDHC Memory Card under a specific directory structure. The directory structure and file format utilize the FAT32 specification which permits the SD Memory Card to be read by a standard IBM PC computer system.

### Directory Structure

This section discusses the directory structure used within the FAT32 system of the SD Memory Card.

#### General User Accessible Directories

The general user directories are those which may be accessed by the average operator.

The following directories are implemented:

Patient\_Settings

Operator\_Setting

Alarm\_Logs

-- Detailed (This is a sub-directory)

Test\_Logs

-- EST (EST and SST are sub-directories)

-- SST

Calibration\_Logs

Event\_Recorder

-- General

-- SPI

Diagnostic\_Logs

Service\_Logs

Maintenance\_Logs

Communication\_Logs

Device\_Alert

#### Secure Directories

The secure directories are those which contain files which may not be accessed and displayed by the clinician ventilator operator.

There are two types of secure directories:

1. Field (Directories which may be accessed by Field Service and Covidien Engineers)
2. Covidien (Directories which may only be accessed by Covidien Engineers)

**The following secure directories may only be accessed by Field Service and Covidien Engineers.**

Patient\_Settings

Operator\_Setting

Alarm\_Logs

-- Detailed (This is a sub-directory)

Test\_Logs

-- EST (EST and SST are sub-directories)

-- SST

Calibration\_Logs

Event\_Recorder

-- General

-- SPI

Diagnostic\_Logs

Service\_Logs

Maintenance\_Logs

Communication\_Logs

Device\_Alert

**The following secure directories may only be accessed by Covidien Engineers…**

Patient\_Settings

Operator\_Setting

Alarm\_Logs

-- Detailed (This is a sub-directory)

Test\_Logs

-- EST (EST and SST are sub-directories)

-- SST

Calibration\_Logs

Event\_Recorder

-- General

-- SPI

Diagnostic\_Logs

Service\_Logs

Maintenance\_Logs

Communication\_Logs

Device\_Alert

##### Secure Field Directory

The Secure “Field” Directory contains files and/or sub-directories which may only be accessed by a Covidien Field Engineer.

These directories include the following:

Event

##### Secure Covidien Directory

The Secure “Covidien” Directory contains files and/or sub-directories which may only be accessed by Covidien Engineers.

These directories include the following:

Diagnostic

### File System Access Protection

The FileX system utilizes ThreadX semaphores for multiple thread protection and I/O suspension. All FileX file system calls result in sector read/write operations effected by the SD Memory Card itself. The File Access C++ Class will be used to control access to each directory and file.

### Log Database

#### System Development

During system development it is necessary to ensure that the developers that are using the logging system are working with data log files of the correct revision. An incorrect revision implies that the log record data structure has been updated and errors will result when attempting to access such files. To overcome such issues the following procedure has been developed.

**Configuration File**

On startup the logging system verifies the directory database by initially accessing the configuration file. This configuration file is used to identify the revisions for each of the specific logs stored within the log database.

**Configuration File Exists**

**No Old Revisions and No Missing Directories**

If the configuration exists and all the identified file revisions are current and there are no missing log database directories, the database validity has been confirmed and may be used.

**Old revisions**

If the configuration file identifies an old revision for a specific log file type then the files in that specific log directory (and its sub-directories) will be automatically deleted.

**Missing directories**

If a directory is detected to be missing, that directory will be created.

**Missing Configuration File**

If the configuration file is missing a new configuration file will be created. During this process each database log directory will be accessed and a file in each log accessed to obtain its revision ID. If the ID is current then all is well. If the ID is not current then all files in that directory will be deleted. If a directory is missing then it will be created. The new configuration file that is created will contain current revision IDs.

The configuration file is called “config.dat” and resides in the /log directory.

Operational Description

#### Production Usage

Files shall not be simply deleted during production usage. When a file revision is changed a utility should be provided to convert the old file structure to the new current structure definition.

## Ventilator Log Definitions

### Base Classes

At a high-level logging subsystem can be viewed as data records residing in log files which can be accessed by data access API. SysLogsLogRecord contains the common information application to all log record types and serves as the base class of all the record types and is defined below.

**class** SysLogsLogRecord

{

**public**:

SysLogsCommonInfo info;

};

SysLogsCommonInfo contains the following information.

**struct** SysLogsCommonInfo

{

TimeStamp timestamp;

Uint32 patientId;

Uint32 checksum;

};

SysLogsBaseLog serves as the base class for all log files. The following table defines and describes the data members and API provided by the base class which are further re-used and re-defined in the derived classes.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| m\_syslogsMutex | Data | Protects the Log file from simultaneous access and race conditions. |
| m\_SysLogsFilesPtr | Data | Pointer to the File System API class |
| m\_logFullPathFilename | Data | Path name of the log file |
| m\_sysLogsLogType | Data | Type of the log file (Alarm Log, Settings Log etc.) |
| m\_sysLogsLogRecordSize | Data | Size of the log records contained in the log |
| m\_sysLogsLogVersion | Data | Version identification information |
| m\_sysLogsLogId | Data | File identification information |
| m\_sysLogsClassId | Data | Class identification used during software exceptions |
| m\_ramLogHeader | Data | Contains the run-time information (insert, extract positions, entry counts, warning levels etc.) about the log file. |
| m\_logStatus | Data | Maintains the valid or invalid status of the log |
|  |  |  |
| LoadRamLogHeader**()** | API | This function is used to load the log header from the flash file system into RAM memory |
| CreateLog**()** | API | This function is used to build the log header then create the log file in the file system. |
| SetNewPatient() | API | This function is used to increment the Patient ID whenever the operator selects the “New Patient” option |
| ResetLogLevels() | API | This function should be called whenever the operator has copied the log to secondary storage. The log file warning and full pointers are then adjusted for the new warning and full index positions. |
| ReadLog() | API | This function is used to retrieve log data based on a search event time and requested number of records which are passed in as parameters. |
| ReadLogBlock() | API | This function is used to retrieve a specific block of log records. GUI can leverage this API to request records based on display page sizes |
| WriteLog() | API | This function is used to write Log data to the Log file. |

### Patient Data Log

The Patient Data Log contains information which is specific only to the current patient. When the [New Patient] button is pressed the previously stored patient data may be displayed in service more but not in normal mode. When the patient data file size reaches the 90% full mark, the GUI application is notified. When the full mark is reached the data storage region is reused by having the oldest record being deleted and replaced with the latest data. This involves using pointers to identify the oldest and latest data entries. A field also identifies the current patient.

Since data in the header is being modified with every change recorded in the patient data log, the header will be loaded into RAM on startup and modified as required. When the system is being shut down, the header will then be written into the data file.

The Patient Log Data is an overall view of the current patient conditions. For instance, when a specific exhaled tidal volume is specified, it is not the instantaneous tidal volume level but the tidal volume level for the entire exhaled breath.

The number of record entries is set at 72 \* 60.

One entry every minute for 72 hours.

**File Data**

The Patient Data to be stored is defined within the following structure:

**struct** PatientInfo

{

Real32 breathPhaseItem;

Real32 breathTypeItem;

Real32 breathTypeDisplayItem;

Real32 isPeepRecoveryItem;

Real32 staticAirwayResistanceValidItem;

Real32 staticLungComplianceValidItem;

Real32 endExpPausePressureStateItem;

Real32 pavStateItem;

Real32 vtpcvStateItem;

Real32 previousBreathTypeItem;

Real32 isNivInspTooLongItem;

Real32 isLeakDetectedItem;

Real32 endExpPressItem;

Real32 exhTidalVolItem;

Real32 peakCctPressItem;

Real32 meanCctPressItem;

Real32 ieRatioItem;

Real32 exhMandTidalVolItem;

Real32 exhSpontTidalVolItem;

Real32 inspTidalVolItem;

Real32 inspTidalVolAlarmItem;

Real32 exhMinuteVolItem;

Real32 exhSpontMinuteVolItem;

Real32 totalRespRateItem;

Real32 peepIntrinsicItem;

Real32 peepTotalItem;

Real32 staticLungComplianceItem;

Real32 staticAirwayResistanceItem;

Real32 plateauPressureItem;

Real32 percentLeakItem;

Real32 exhLeakRateItem;

Real32 inspLeakVolItem;

Real32 leakVolumeItem;

Real32 endInspPressItem;

Real32 deliveredO2PercentageItem;

Real32 spontRateToVolumeRatioItem;

Real32 spontPercentTiItem;

Real32 spontInspTimeItem;

Real32 pavLungComplianceItem;

Real32 pavPatientResistanceItem;

Real32 totalAirwayResistanceItem;

Real32 pavLungElastanceItem;

Real32 pavPeepIntrinsicItem;

Real32 elastanceWorkOfBreathingItem;

Real32 resistiveWorkOfBreathingItem;

Real32 patientWorkOfBreathingItem;

Real32 totalWorkOfBreathingItem;

Real32 endTidalCo2Item;

Real32 volumePerUnitTime;

Real32 dynamicComplianceItem;

Real32 dynamicResistanceItem;

Real32 peakSpontInspFlowItem;

Real32 c20ToCRatioItem;

Real32 peakExpiratoryFlowItem;

Real32 endExpiratoryFlowItem;

Real32 nifPressureItem;

Real32 p100PressureItem;

Real32 vitalCapacityItem;

Real32 netFlowItem;

Real32 netVolItem;

Real32 circuitPressItem;

Real32 lungPressItem;

Real32 lungFlowItem;

Real32 lungVolumeItem;

Real32 iesyncDeltaInterpleuralP;

Real32 iesyncIeSignal;

Real32 isApneaActiveItem;

Real32 isErrorStateItem;

Real32 modeSettingItem;

Real32 supportTypeSettingItem;

Real32 autozeroActiveItem;

Real32 realTimeClockItem;

Real32 largeFontDisplayPlot1Item;

Real32 largeFontDisplayPlot2Item;

Real32 largeFontDisplayPlot3Item;

UnusedFourWords unusedFourWords; // future expansion

};

**class** SysLogsPatientDataLogRecord : **public** SysLogsLogRecord

{

**public**:

PatientInfo patient\_info;

Uint32 checksum;

};

**File Data Storage**

Overall Storage requirements are:

* 348 Bytes / minute
* 1.47MB / 72 Hours (3 Days)

### Alarm Log

The Alarm Log contains all the details pertaining to each ventilator alarm (high, medium and low) along with the physiological aspects associated with the alarm (eg. information pertaining to the ambient noise floor before, during and after the alarm providing proof that the alarm tone associated with this alarm did in fact sound). Operator actions associated with the alarm event are also recorded (eg. alarm reset, alarm silence,etc.). At least 1000 of the most recent entries must be stored. Only the alarm log for the current patient shall be made available to the clinician. The full alarm log can be viewed in Service Mode.

**struct** AlarmLogData

{

Uint32 alarmID;

Uint32 alarmBaseMessageType;

Uint32 alarmUrgencyType;

Uint32 alarmUpdateType;

Uint32 alarmAnalysisMessageArray[MAX\_ALARM\_ANALYSIS\_MESSAGES];

Uint8 alarmAnalysisMessageCount;

Uint8 ambientNoiseBeforeAlarm;

Uint8 alarmVolumeDuringAlarm;

Uint8 alarmAmbientSetting;

UnusedFourWords unusedFourWords;

};

**class** SysLogsAlarmsLogRecord : **public** SysLogsLogRecord

{

**public**:

AlarmLogData alarm; // Alarm information

Uint32 checksum;

};

**File Data Storage**

The specification states that 1000 of the most recent alarms are to exist in the Alarm Log file.

File size = 1000 x 68 bytes = 66.4 KBytes.

### System Diagnostic Logs

The System Diagnostic Log is comprised of the following three separate diagnostic log files:

1. System Diagnostic Log
2. System Information Diagnostic Log
3. EST/SST Diagnostic Log

Each of these diagnostic logs retain 256 diagnostic records.

Diagnostic logs are not erased when a “New Patient” is selected.

**File Data**

The following data structure contains the data needed and provides a uniform structure that can be used for all the above mentioned diagnostic logs.

**class** SysLogsSystemDiagnosticsLogRecord : **public** SysLogsLogRecord

{

**public**:

DiagnosticCode systemDiagnosticsData;

Uint32 checksum;

};

**File Creation / Deletion**

If no such file exists the file is created.

**File Modification**

Records are added when any associated errors/faults are detected.

**File Size Restriction**

The log will store 256 entries of records.

Records in excess of 256 will be deleted.

**File Data Transfer Rate**

The System Diagnostic entries may occur infrequently, however, there may be a burst of errors detected if a failure was detected.

If, for example, 20 errors were to be detected in one second, the data rate would be:

20 \* 36 = 0.7 KBytes / second.

**File Data Storage**

Assuming 256 entries, each of the Diagnostic Logs would attain the following size:

36 bytes \* 256 = 9 KBytes

#### System Diagnostic Log

The System Diagnostic Log contains:

1. POST diagnostics
2. Background Test diagnostics
3. Soft faults (eg. assertion failures and spurious interrupts)

#### System Information Diagnostic Log

The System Information Diagnostic Log contains diagnostic information that was generated by the communications software as well as other system events such as the resetting of the system clock:

BD System Clock Reset

GUI System Clock Reset

Diagnostics generated by the BD Ethernet communications interface

Diagnostics generated by the BD SPI communications interface

Diagnostics generated by the BD Serial communications interface

Diagnostics generated by the BD USB communications interface

Diagnostics generated by the GUI Ethernet communications interface

Diagnostics generated by the GUI SPI communications interface

Diagnostics generated by the GUI Serial communications interface

Diagnostics generated by the GUI USB communications interface

Etc…

#### EST/SST Diagnostic Log

The EST/SST Diagnostic Log contains diagnostic fault information obtained from:

Extended Self Tests (EST)

Short Self Tests (SST)

EST and SST errors (not test results)

Major and minor Alerts.

### Operator Settings Log [P2.2 Schedule]

The Operator Settings Log records changes made to the ventilator settings by the operator.

It is required that a minimum of 500 operator settings changes will be stored in the log file.

**File Data**

**typedef** **struct** CALENDAR\_t

{

Uint8 day; // Range 1..31

Uint8 month; // Range 1..12

Uint8 year; // Range 00..99 (within 2000 century)

Uint8 hour; // Range 0..23

Uint8 minute; // Range 0..59

Uint8 second; // Range 0..59

Uint16 milliseconds; // Range 0..999

}

CALENDAR; // 8 Bytes

**typedef** **struct** CLINICIAN\_t

{

Uint8 id [16]; // 16 characters, null terminated

Uint8 filler [16] // 16 Bytes – Other possible clinician data???

}

CLINICIAN; // 32 Bytes

**typedef** **struct** OPERATOR\_SETTINGS\_t

{

Uint32 settings\_id;

Uint32 settings\_old\_value;

Uint32 settings\_new\_value; // 12 Bytes

Uint8 filler [12] // 12 Bytes

}

OPERATOR\_SETTINGS; // 24 Bytes (Total)

**typedef struct** OPERATOR\_SETTINGS\_FILE\_RECORD\_t

{

CALENDAR calendar; // 8 Bytes

CLINICIAN clinician; // 32 Bytes

OPERATOR\_SETTINGS settings; // 24 Bytes

}

OPERATOR\_SETTINGS\_FILE\_RECORD // 64 Bytes (Total)

**Overview**

The Patient Setting file will have the name: “setting\_current\_patient.dat”. Only this file shall be available to the clinician.

When a New Patient is selected, the current patient setting file “setting\_current\_patient.dat” will be archived through a process whereby the file is renamed to the following format: setting\_date\_time.dat where the date and time fields relate to the date and time when the settings file was created (ie. the date and time as of when the operator pressed the “New Patient” option). This file (and others with the same naming convention) will be available to all other users excepting the clinician.

If file storage memory availability is limited then the clinician is granted the opportunity to transfer the current settings file to external storage. Subsequent to this, the settings file shall be limited to the latest 500 setting changes and all archived setting files will be deleted.

The setting data is to be retained for view by service personnel following the selection of New Patient during ventilator setup. This implies that we cannot use a fixed size file and cyclically rotate through the records using start and end pointers to records as this would retain the latst 500 records (say) but would drop the old records so they would no longer be available.

**File Naming Convention and Usage**

??? WHAT IS THE LONGEST CONTIGUOUS TIME THAT THE VENTILATOR WILL VENT A PATIENT PRIOR TO THE “NEW PATIENT” BUTTON BEING PRESSED ???

Setting data is stored in files that can contain a maximum of 24 hours of data.

The current setting files use the format: setting\_current.dat

The archived setting files use the format: setting\_archive\_yymmdd\_hhmmss.dat

Where: yy = year tens and units hh = hour tens and units

mm = month tens and units mm = minute tens and units

dd = day tens and units ss = second tens and units

No prior alarm data

[New Patient] Pressed

Directory: /log/setting/

setting\_current New Patient

[New Patient] Pressed

Directory: /log/setting/

setting\_archive\_110608\_053422.dat Prior patient

setting\_current.dat New Patient

[New Patient] Pressed

Directory: /log/setting/

setting\_archive\_110608\_053422.dat Prior patient

setting\_archive\_110611\_123216.dat Prior patient

setting\_current.dat New Patient

etc…

When a limited storage medium condition is detected, all archived data files are immediately deleted.

Current patient data greater than three days of age will be deleted.

The clinician is granted the opportunity in which the old current patient alarm log files may be transferred to a backup flash drive.

In the previous example, the alarm log directory would now look as follows:

Directory: /log/setting/

setting\_current\_110617\_000000.dat Same patient, next day

setting\_current\_110618\_000000.dat Same patient, next day

setting\_current\_110619\_000000.dat Same patient, next day

**File Creation / Deletion**

If operator selects “New Patient” option in GUI Normal Mode

If Setting file does not exist

Create new Setting file for current patient

Else

Rename existing Setting file to Setting\_date\_time format

using the date and time when the file was initially created

Create new Setting file for current patient

Endif

Endif

**File Modification**

Append the setting entry to the end of the current patient Setting file.

**File Size Restriction**

There is a minimum file size restriction of 500 setting changes

There are no maximum file size restrictions.

**Limited Memory Availability**

If limited memory availability

Permit clinician to transfer current patient setting file to external storage

If current patient setting file has more than 500 setting changes

Truncate current patient setting file to latest 500 entries

Endif

Delete all existing archived setting files

Endif

**File Data Transfer Rate**

The file data rate is very low, being based upon the number of operator settings changed in a given time period.

One operator change would be effected in one second giving a transfer rate of 64 Bytes/sec.

If there were 8 operator changes in one minute = 512 Bytes/minute.

If this same rate continued we would have 480 operator changes in one hour: 30KBytes/hour

**File Data Storage**

Overall Storage requirements are:

* 512 Bytes / minute
* 720 KBytes / day

The ventilator is required to have a minimum of 500 operator settings changes for a given patient.

This would result in a file storage of 500 \* 64 bytes = 32,000 Bytes (31.25 KB)

An operator settings log containing 16384 entries would be of size 1.0 MB.

There is an SRS requirement to provide the clinician with the means of transferring the Operator Settings Log to an external storage device when the allocated space is exhausted. This operator request is most unlikely in being triggered due to the minimal size of the file.

**Questions**

1. How do I get access to this data?

2. Why clear this with “New Patient” – one might want to review setting changes if you were looking at the event recorder. Perhaps only allow the current patient log to be viewed by the clinician operator. Logs older than 73+ hours would be deleted.

### Alarm Event Log

The Detailed Alarm Log contains alarm-related events, patient and ventilator data surrounding each alarm event. Please refer to the Ventilator Logs SRS document for full details. This file is not deleted with each “New Patient” selection.

This file is created on the occurrence of an alarm condition.

**File Naming Convention and Usage**

Each alarm eventAlarm data is stored in files that can contain a maximum of 24 hours of data.

The current alarmevent files use the format: alarm\_current\_yymmdd\_hhmmss\_nnn.dat

Where: yy = year tens and units hh = hour tens and units

mm = month tens and units mm = minute tens and units

dd = day tens and units ss = second tens and units

nnn = numeric increment for each alarm event

The archived alarmevent files use the format: alarm\_archive\_yymmdd\_hhmmss.dat

Where: yy = year tens and units hh = hour tens and units

mm = month tens and units mm = minute tens and units

dd = day tens and units ss = second tens and units

nnn = numeric increment for each alarm event

No prior alarm event data

[New Patient] Pressed

Directory: /log/alarm\_event/

alarmevent\_current\_110608\_053422\_001.dat First alarm at this time

alarmevent\_current\_110608\_053422\_002.dat Second alarm at this time

alarmevent\_current\_110608\_053422\_003.dat Third alarm at this time

alarmevent\_current\_110608\_054509\_001.dat First alarm at this time

alarmevent\_current\_110608\_120456\_001.dat First alarm at this time

[New Patient] Pressed

Directory: /log/alarm\_event/

alarmevent\_archive\_110608\_053422\_001.dat First alarm at this time

alarmevent\_archive\_110608\_053422\_002.dat Second alarm at this time

alarmevent\_archive\_110608\_053422\_003.dat Third alarm at this time

alarmevent\_archive\_110608\_054509\_001.dat First alarm at this time

alarmevent\_archive\_110608\_120456\_001.dat First alarm at this time

alarmevent\_current\_110609\_152308\_001.dat First alarm at this time

alarmevent\_current\_110609\_172230\_001.dat First alarm at this time

When a limited storage medium condition is detected, all archived data files are immediately deleted.

The last 20 alarm event logs will be retained and the others will be deleted.

The clinician is granted the opportunity in which the current alarm event log files to be deleted may be transferred to a backup flash drive.

In the previous example, the alarm event log directory would now look as follows:

Directory: /log/alarm\_event/

alarmevent\_current\_110609\_152308\_001.dat First alarm at this time

alarmevent\_current\_110609\_172230\_001.dat First alarm at this time

**File Data**

The Alarm Log entry

Operator Settings (Operator Settings Log entry immediately prior to alarm event)

Patient Data (Patient Date Log entry immediately prior to alarm event)

Patient Data (Patient Date Log entry at alarm event)

Patient Data (Patient Date Log entry immediately after the alarm event)

Command Data (From Event Recorder Log)

Sensor Data (From Event Recorder Log)

Ambient sound levels prior to, during and immediately after the alarm event

**File Creation / Deletion**

If file does not exist, create file

If limited storage memory available

Delete the oldest Alarm Log file records

Endif

**File Modification**

Append the detailed alarm log entries to the end of the file

**File Size Restriction**

There is no file restriction.

The file shall contain at least 1000 entries.

**File Access**

Open the file for reading

Set the file read pointer

Perform the read operation

Close the file

**File Data Transfer Rate**

Low.

**File Data Storage**

The total file data storage required would incorporate:

The Alarm Log entry 272 Bytes

Operator Settings 64 Bytes

Patient Data (Patient Date Log entry immediately prior to alarm event) 256 Bytes

Patient Data (Patient Date Log entry at alarm event) 256 Bytes

Patient Data (Patient Date Log entry immediately after the alarm event) 256 Bytes

Command Data (From Event Recorder Log) 244 Bytes

Sensor Data (From Event Recorder Log) 864 Bytes

Ambient sound levels prior to, during and immediately after the alarm event 12 Bytes

Giving a total of: 2224 Bytes

The Detailed Alarm Log is required to store 1000 entries = 1000 \* 2224 = 2.13 MBytes

**Questions**

How do I get access to this data?

### Extended Self Test (EST) Log

The Extended Self Test (EST) Log contains the results of the previously executed set of Extended Self Tests. This log identifies the date and time when the test was performed, a description of each test which was performed along with the associated test result.

This file is fixed in size and utilizes the following file naming convention:

Directory: /log/est/

est.dat

**File Data**

**typedef** **struct** EST\_TEST\_t

{

Uint16 test\_result\_code // Success / Error code

Uint8 test\_description [116]; // Test description

Uint8 test\_result\_text [116]; // Pass / Fail descriptive text

}

EST\_TEST; // 234 Bytes total

#define MAX\_EST\_TESTS 20

**typedef** **struct** EST\_TEST\_LOG\_t

{

CALENDAR calendar; // Test timestamp [ 8 Bytes]

Uint16 number\_of\_tests; // Number of tests performed [ 2 Bytes]

EST\_TEST tests [MAX\_EST\_TESTS]; // Test details [4680 Bytes]

Uint8 filler [430]; // To fill up to next 1K boundary

}

EST\_TEST\_LOG; // Total [5120 Bytes]

**File Creation / Deletion**

This file is created when the EST test is executed.

**File Modification**

This file is created when the EST test is executed.

**File Size Restriction**

There is no file size restriction.

**File Data Transfer Rate**

When this file is created there is a data transfer of 5KB.

**File Data Storage**

This file is 5KB in size.

**Questions**

How do I get access to this data?

### Short Self Test (SST) Log

The Short Self Test (SST) Log contains the results of the previously executed set of Short Self Tests. This log identifies the date and time when the test was performed, a description of each test which was performed along with the associated test result.

This file is fixed in size and utilizes the following file naming convention:

Directory: /log/sst/

sst.dat

**File Data**

**typedef** **struct** SST\_TEST\_t

{

Uint16 test\_result\_code // Success / Error code

Uint8 test\_description [116]; // Test description

Uint8 test\_result\_text [116]; // Pass / Fail descriptive text

}

SST\_TEST; // 234 Bytes total

#define MAX\_SST\_TESTS 20

**typedef** **struct** SST\_TEST\_LOG\_t

{

CALENDAR calendar; // Test timestamp [ 8 Bytes]

Uint16 number\_of\_tests; // Number of tests performed [ 2 Bytes]

SST\_TEST tests [MAX\_SST\_TESTS]; // Test details [4680 Bytes]

Uint8 filler [430]; // To fill up to next 1K boundary

}

SST\_TEST\_LOG; // Total [5120 Bytes]

**File Creation / Deletion**

This file is created when the SST test is executed.

**File Modification**

This file is created when the SST test is executed.

**File Size Restriction**

**File Data Transfer Rate**

When this file is created there is a data transfer of 5KB.

**File Data Storage**

This file is 5KB in size.

**Questions**

How do I get access to this data?

### Calibration Log [Not on P2.2 Schedule]

The Calibration Log contains the results of the previously executed Calibration Tests. This log identifies the date and time when the calibration was performed, a description of the device that was calibrated along with the associated calibration result.

**File Data**

**typedef** **struct** CALIBRATION\_t

{

CALENDAR calendar; // Test timestamp [ 8 Bytes]

Uint8 description [120]; // Device identification [120 Bytes]

Uint8 calibration [120]; // Calibration text [120 Bytes]

}

CALIBRATION; // 248 Bytes total

#define MAX\_CALIBRATIONS 20

**typedef** **struct** CALIBRATION\_LOG\_t

{

Uint16 number\_of\_devices; // # calibrated devices [ 2 Bytes]

CALIBRATION device [MAX\_NUM\_DEVICES]; // Calibration details [4960 Bytes]

Uint8 filler [158]; // Filler to 1K boundary [ 158 Bytes]

}

CALIBRATION\_LOG; // Total [5120 Bytes]

This file is fixed in size and utilizes the following file naming convention:

Directory: /log/cal/

calibration.dat

**File Creation / Deletion**

If no Calibration Log file exists

Create Calibration Log identifying each device and stating that all are uncalibrated

Endif

**File Modification**

If a device has been calibrated

Enter the Calibration date/time and description for the calibrated device

Endif

**File Size Restriction**

There is no file size restriction.**File Data Transfer Rate**

When this file is created there is a data transfer of 5KB.

**File Data Storage**

This file is 5KB in size.

**Questions**

How do I get access to this data?

### General Event Log [P2.2 Schedule]

The general event log stores up to 72 hours of clinical patient monitoring using 20ms samples. Manual and automatic event markers are also stored, along with current ventilator settings.

**File Data**

The information that is currently being displayed as a waveform is the following:

**struct** WaveformData

{

**inline** **void** **operator=**(**const** WaveformData& waveformData);

Real32 circuitPressure;

Real32 lungPressure;

Real32 netFlow;

Real32 lungFlow;

Real32 netVolume;

Real32 lungVolume;

BreathPhaseType::PhaseType bdPhase; // enum

BreathType breathType; // enum

Boolean isPeepRecovery;

Boolean isApneaActive;

Boolean isErrorState;

Boolean isAutozeroActive;

DiscreteValue modeSetting ; // Int16

DiscreteValue supportTypeSetting ; // Int16

Uint32 timeStamp;

};

**The PRD inferred the following data to be stored:**

**typedef** **struct** CALENDAR\_t

{

Uint8 day; // Range 1..31

Uint8 month; // Range 1..12

Uint8 year; // Range 00..99 (within 2000 century)

Uint8 hour; // Range 0..23

Uint8 minute; // Range 0..59

Uint8 second; // Range 00..59

Uint16 milliseconds; // Range 0..999

}

CALENDAR; // 8 bytes

Typedef Struct EVENT\_PATIENT\_DATA\_t // 12 x 32-bit words = 48 Bytes

{

Real32 Inhalation Flow  
 Real32 Exhalation Flow  
 Real32 Proximal Flow (if Proximal Flow sensor option installed)  
 Real32 Auxiliary Pressure (Pcari, Peso, etc.)  
 Real32 Inhalation Pressure  
 Real32 Exhalation Pressure  
 Real32 Proximal Pressure (if the Proximal Flow sensor option is installed).  
 Real32 CO2 Reading (if the CO2 monitoring option is installed).  
 Real32 SpO2 (if the Nell 1 Module is installed).  
 Real32 Estimated airway resistance  
 Real32 Estimated lung compliance  
 Real32 Estimated lung flow

}

EVENT\_PATIENT\_DATA;

Typedef Struct EVENT\_WAVEFORM\_DATA\_t // 10 x 32-bit words = 40 Bytes

{

Real32 Flow Waveform  
 Real32 Proximal (if proximal flow sensor is attached)  
 Real32 Estimated wye flow   
 Real32 Pressure Waveform  
 Real32 Volume Waveform  
 Real32 Pressure Volume Loops  
 Real32 Flow Volume Loops  
 Real32 SpO2 Plethysmograph  
 Real32 Et CO2 Real32 Volumetric CO2

}

EVENT\_WAVEFORM\_DATA;

Typedef Struct EVENT\_VENTILATOR\_DATA\_t // 51 32-bit words = 204 Bytes

{

Real32 Measured SpO2, if pulse ox attached.  
 Real32 Measured CO2 (if capnography installed).  
 Real32 Measured supply air flow  
 Real32 Measured supply O2 flow  
 Real32 Measured supply Heliox flow  
 Real32 Measured supply air pressure  
 Real32 Measured supply O2 pressure  
 Real32 Measured supply Heliox pressure  
 Real32 Measured accumulator pressure  
 Real32 Measured BD PSOL current  
 Real32 Commanded BD PSOL current  
 Real32 Measured EV current  
 Real32 Commanded EV current  
 Real32 Measured EV position  
 Real32 Commanded EV position  
 Real32 Measured supply air PSOL current  
 Real32 Commanded supply air PSOL current  
 Real32 Measured supply O2 PSOL current  
 Real32 Commanded supply O2 PSOL current  
 Real32 Measured supply Heliox PSOL current  
 Real32 Commanded supply Heliox PSOL current  
 Real32 Commanded EV damping voltage  
 Real32 Commanded mix percentage  
 Real32 Estimated mix percentage  
 Real32 Measured O2 sensor reading  
 Real32 Measured temperature of inhaled gas  
 Real32 Measured temperature of exhaled gas  
 Real32 State of all solenoids in the system  
 Real32 Measured Inhalation Flow  
 Real32 Measured Exhalation Flow  
 Real32 Measured Proximal Flow (if Proximal Flow sensor option installed)  
 Real32 Measured Inhalation Pressure  
 Real32 Measured Exhalation Pressure  
 Real32 Measured Proximal Pressure (if Proximal Flow sensor option nstalled)  
 Real32 Measured Air Flow into the Accumulator.  
 Real32 Measured Oxygen Flow into the Accumulator.  
 Real32 Measured Heliox Flow into the Accumulator.  
 Real32 Measured Air Inlet Pressure  
 Real32 Measured Oxygen Inlet Pressure  
 Real32 Measured Heliox Inlet Pressure  
 Real32 Commanded Inhalation Valve Position  
 Real32 Commanded Accumulator Air Flow Valve Position  
 Real32 Commanded Accumulator Oxygen Flow Valve Position  
 Real32 Commanded Accumulator Heliox Flow Valve Position  
 Real32 Commanded position of each solenoid valve in the system  
 Real32 Measured AC inlet voltage  
 Real32 Measured Battery Voltage for each battery  
 Real32 Current state for each off screen control  
 Real32 Current state for the touch screen  
 Real32 Commanded Audible Alarm status  
 Real32 Measured Audible Alarm status

}

EVENT\_VENTILATOR\_DATA;

Typedef Struct EVENTS\_t

{

CALENDAR calendar; // 8 Bytes

EVENT\_PATIENT\_DATA patient; // 48 Bytes

EVENT\_WAVEFORM\_DATA waveform; // 40 Bytes

EVENT\_VENTILATOR\_DATA ventilator; // 204 Bytes = 300 Bytes

}

EVENTS;

**File Naming Convention and Usage**

Event data is stored in files that can contain a maximum of 1 hour of data.

The current event files use the format: event\_current\_yymmdd\_hhmmss.dat

Where: yy = year tens and units hh = hour tens and units

mm = month tens and units mm = minute tens and units

dd = day tens and units ss = second tens and units

The archived event files use the format: event\_archive\_yymmdd\_hhmmss.dat

Where: yy = year tens and units hh = hour tens and units

mm = month tens and units mm = minute tens and units

dd = day tens and units ss = second tens and units

No prior alarm data

[New Patient] Pressed

Directory: /log/event/

event\_current\_110608\_053422.dat New Patient

event\_current\_110608\_060000.dat Same patient, next hour

event\_current\_110608\_070000.dat Same patient, next hour

event\_current\_110608\_0800000.dat Same patient, next hour

[New Patient] Pressed

Directory: /log/event/

event\_archive\_110608\_053422.dat Prior patient

event\_archive\_110608\_060000.dat Prior patient, next hour

event\_archive\_110608\_070000.dat Prior patient, next hour

event\_archive\_110608\_0800000.dat Prior patient, next hour

event\_current\_110608\_153408.dat New Patient

event\_current\_110608\_160000.dat Same patient, next hour

event\_current\_110608\_170000.dat Same patient, next hour

event\_current\_110608\_180000.dat Same patient, next hour

event\_current\_110608\_190000.dat Same patient, next hour

event\_current\_110608\_200000.dat Same patient, next hour

event\_current\_110608\_210000.dat Same patient, next hour

event\_current\_110608\_220000.dat Same patient, next hour

event\_current\_110608\_230000.dat Same patient, next hour

event\_current\_110609\_000000.dat Same patient, next hour of next day

event\_current\_110609\_010000.dat Same patient, next hour

event\_current\_110609\_020000.dat Same patient, next hour

event\_current\_110609\_030000.dat Same patient, next hour

etc…

The system must retain the last 72 hours of patient data during patient ventilation.

When the patient is being ventilated, any event data files (current or archive) with a filename timestamp greater than 72 hours in the past will be automatically deleted.

**File Creation / Deletion**

The files is created when the ventilator is first turned on.

File data older than 72 hours will be erased.

**File Modification**

Data records are continually appended to the Event Log.

**File Size Restriction**

A maximum of 72 hours worth of events are to be recorded.

Data records older than 72 hours are deleted.

**File Data Transfer Rate [Moderate]**

Data record entries are normally written every 20ms.

50 data record entries are then stored every second.

300 Bytes x 50 = 15000 Bytes = 14.65KB/sec

If the data entries were stored every 10ms the data transfer rate is 29.30 KB/sec.

If the data entries were stored every 5ms the data transfer rate is 58.60 KB/sec.

If the data entries were stored every 2ms the data transfer rate is 146.50 KB/sec.

If the data entries were stored every 1ms the data transfer rate is 293.00 KB/sec.

The 1ms sample rate would correspond to the NeoNatal mode.

The SD Class 10 Memory Cards can currently operate at a bandwidth of 10MB/second so the above data transfer rates would not be a concern.

**File Data Storage**

Overall Storage requirements at 20ms for 72 hours:

15000 bytes/sec x 3600 secs/hour x 72 hours = 3888000000 Bytes = 3.63 GBytes

Overall Storage requirements at 10ms for 72 hours:

29200 bytes/sec x 3600 secs/hour x 72 hours = 7.26 GBytes

Overall Storage requirements at 5ms for 72 hours:

58400 bytes/sec x 3600 secs/hour x 72 hours = 14.52 GBytes

Overall Storage requirements at 2ms for 72 hours:

146000 bytes/sec x 3600 secs/hour x 72 hours = 36.30 GBytes

Overall Storage requirements at 1ms for 72 hours:

292000 bytes/sec x 3600 secs/hour x 72 hours = 72.60 GBytes

The current maximum SD Memory Card envisioned being used is 32GB in size.

At present 64GB SDXC memory cards are available and the 128GB memory card should be available to consumers some time in 2011.

**Questions**

See Waveforms.

### SPI Backplane Event Log [Not on P2.2 Schedule]

The general event log stores up to 72 hours of clinical patient monitoring using 20ms samples. Manual and automatic event markers are also stored, along with current ventilator settings. The SPI event log contains the actual data packets communicated over the SPI backplane. These data packets are in RAW data form and are required to be converted into engineering units form. The required parameters (Barometric pressure, Flow Sensor Tables, etc.) must be used to perform this conversion. This implies that these parameters are stored along with the raw data in order to perform the conversion to engineering units.

**File Data**

SPI Event Data includes all the SPI communication messages which traverse the SPI backplane.

With the current scenario, the protocol loading is as follows:

**BD Transmitted Messages (Total of 52 32-bit words per control cycle)**

PI Insp Control 10 words

PI Exh Control 10 words

Mux Control 8 words

Power Vent Control 10 words

Power Comp Control 14 words

**BD Received Messages (Total of 208 32-bit words per control cycle)**

PI Insp Response 32 words

PI Exh Response 28 words

Mux Response 38 words

Power Vent Response 47 words

Power Comp Response 63 words

If these BD SPI messages are to be stored on the GUI SD Memory Card then this information is required to be transferred across to the GUI system.

This may be achieved by the BD sending this data across the Ethernet channel to the GUI system.

The required data loading would be 260 \* 4 \* 50 = 50.78 KB/sec.

**GUI Transmitted Messages (Total of 9 32-bit words per control cycle)**

User Interface Control 9 words

**GUI Received Messages (Total of 8 32-bit words per control cycle)**

User Interface Response 8 words

**File Creation / Deletion**

The files is created when the ventilator is first turned on.

File data older than 72 hours will be erased.

**File Modification**

Data records are continually appended to the Event Log.

**File Size Restriction**

A maximum of 72 hours worth of events are to be recorded.

Data records older than 72 hours are deleted.

**File Data Transfer Rate [Moderate]**

The data record entry is written every 20ms.

The total amount of data to be recorded on a 20ms control cycle basis is:

BD Control Messages 52 Words

BD Response Maessages 208 Words

GUI Control Messages 9 Words

GUI Response Messages 8 Words

Total number of words = 277 Words (277 x 32-bit words) / 20ms

= 1108 Bytes / 20ms

= 55400 Bytes / sec

= 54.11 KBytes / sec

**In summary:**

If the data entries were stored every 20ms the data transfer rate is 54.11 KB / sec.

If the data entries were stored every 10ms the data transfer rate is 108.22 KB/sec.

If the data entries were stored every 5ms the data transfer rate is 216.44 KB/sec.

If the data entries were stored every 2ms the data transfer rate is 541.10 KB/sec.

If the data entries were stored every 1ms the data transfer rate is 1.06 MB/sec.

**File Data Storage [High]**

Note that the data stored utilizes variable sized records (ie. each protocol message does not use a fixed buffer size).

The storage requirements = 54.11 KB / second

= 3.17 MB / minute

= 190.20 MB / hour

= 13.38 GB / 72 hour period @ 20ms resolution

**In summary:**

If the data entries were stored every 20ms the data storage for 72 hours is 13.38 GB

If the data entries were stored every 10ms the data storage for 72 hours is 26.76 GB

If the data entries were stored every 5ms the data storage for 72 hours is 53.52 GB

If the data entries were stored every 2ms the data storage for 72 hours is 133.80 GB

If the data entries were stored every 1ms the data storage for 72 hours is 267.60 GB

**Questions**

Should any form of Data Compression be utilized???

### Service Log [P2.2 Schedule]

The Service Log contains information pertaining to the actual service that was performed on the ventilator. The data is entered via a GUI screen.

**File Data**

Calendar Date / Time when the service was performed

Maintenance Bio-Tech Engineer Name

Nature and type of service performed [10 entries]

* + - including Service Reference numbers
    - sensor and actuator ID numbers

Preventative Maintenance Information

Miscellaneous notes [100 characters]

**typedef** **struct** CALENDAR\_t

{

Uint8 day; // Range 1..31

Uint8 month; // Range 1..12

Uint8 year; // Range 00..99 (within 2000 century)

Uint8 hour; // Range 0..23

Uint8 minute; // Range 0..59

Uint8 second; // Range 00..59

Uint16 milliseconds; // Range 0..999

}

CALENDAR; // 8 bytes

**typedef** **struct** SERVICE\_TYPE\_t

{

Uint8 reference\_number [21]; // 20 character, null terminated string

Uint8 part\_id\_number [21]; // 20 character, null terminated string

Uint8 filler [214]; // Filler up to 256 bytes

}

SERVICE\_TYPE

**typedef** **struct** SERVICE\_RECORD\_t

{

CALENDAR calendar; // 8 Bytes

Uint8 engineer\_name [21]; // 20 character, null terminated string

Uint8 number\_of\_serviced\_parts; // Specify number of parts changed

SERVICE\_TYPE type [10]; // Identify up to 10 parts changed

Uint8 miscellaneous\_notes [255]; // Max 254 chars, may be limited to 100

Uint8 filler [1251]; // Fill record up to 4096 bytes in size

} // to accomodate later expansion.

SERVICE\_RECORD;

Total memory required = 4096 Bytes

**Note:** This log is augmented on an infrequent basis therefore providing large filler sections to the record templatewill not adversely affect the file system storage limits.

**File Naming Convention and Usage**

The filename uses the format: service.dat

Directory: /log/service/

service.dat

**File Creation / Deletion**

If the file does not exist an empty service log file is created.

The Service Log file is never deleted.

The Service Log entries are never deleted.

**File Modification**

Service log entries are appended with each ventilator service event.

**File Size Restriction**

There are no file size restrictions.

**File Data Transfer Rate**

Very Low.

**File Data Storage**

Assuming 1000 maintenance service operations, the overall file size would be 4MB.

### Preventative Maintenance Log [Not on P2.2 Schedule]

The Preventative Maintenance Log contains information pertaining to when recommended servicing of the ventilator should occur. For example, if a part has been replaced and that part is to be periodically serviced then an entry should be entered in the Preventative Maintenance Log identifying the part and the date(s) when it is to be serviced.

**File Data**

Types of services to be performed at specific intervals based on calendar and/or usage hours.

**typedef** **struct** CALENDAR\_t

{

Uint8 day; // Range 1..31

Uint8 month; // Range 1..12

Uint8 year; // Range 00..99 (within 2000 century)

Uint8 hour; // Range 0..23

Uint8 minute; // Range 0..59

Uint8 second; // Range 00..59

Uint16 milliseconds; // Range 0..999

}

CALENDAR; // 8 bytes

**typedef** **struct** PM\_TYPE\_t

{

Uint8 reference\_number [21]; // 20 character, null terminated string

Uint8 part\_id\_number [21]; // 20 character, null terminated string

Uint8 filler [214]; // Filler up to 256 bytes

}

PM\_TYPE;

**typedef** **struct** PM\_RECORD\_t

{

CALENDAR calendar; // When the maintenance is due (8 Bytes)

Uint8 engineer\_name [21]; // Engineer who entered PM information

Uint8 number\_of\_service\_items; // Specify number of service items

PM\_TYPE type [10]; // Identify up to 10 service items

Uint8 miscellaneous\_notes [255]; // Max 254 chars, may be limited to 100

Uint8 filler [1251]; // Fill record up to 4096 bytes in size

} // for later expansion.

PM\_RECORD;

Total memory required = 4096 Bytes

**File Naming Convention and Usage**

The filename uses the format: pm.dat

Directory: /log/pm/

pm.dat

**File Creation / Deletion**

If the file does not exist an empty Preventative Maintenance Log file is created.

The Preventative Maintenance Log file is never deleted.

The Preventative Maintenance Log entries are never deleted.

**File Modification**

New entries are appended by the service engineer.

**File Size Restriction**

No file size restrictions.

**File Access**

File read/write from Service Mode

File read only from Normal Mode

**File Data Transfer Rate**

Very Low.

**File Data Storage**

Assuming 1000 preventative maintenance entries, the overall file size would be 4MB.

### Device Alert Log [Not on P2.2 Schedule]

A device alert occurs when an error places the Ventilator into the Vent-Inop mode.

**File Naming Convention and Usage**

Each alarm eventAlarm data is stored in files that can contain a maximum of 24 hours of data.

No prior alarm event data

The filename uses the format: alert\_yymmdd\_hhmmss.dat

Where: yy = year tens and units hh = hour tens and units

mm = month tens and units mm = minute tens and units

dd = day tens and units ss = second tens and units

Directory: /log/alert/

alert\_110118\_011243.dat

alert\_110123\_053129.dat

alert\_110205\_045614.dat

alert\_110523\_124332.dat

alert\_110530\_194152.dat

A maximum of 60 device alert log files must be retained.

**File Data**

The Device Alert entry shall contain a minimum of 30 minutes of log entries from the following log files:

Patient Data Log

Operator Settings Log  
 Diagnostic Log  
 Alarm Event Log  
 Event Recorder Log  
 Service Log

Preventative Maintenance Log

**File Creation / Deletion**

If the file does not exist an empty Device Alert Log file is created.

**File Modification**

New entries are appended with every Device Alert condition.

**File Size Restriction**

A minimum of 60 Device Alert log entries are to be stored.

**File Access**

File read only from Service Mode by Covidien engineers.

There shall be a means to secure and preserve the data stored within the Device Alert Log such that it may not be inadvertently erased.

There shall be a means for an operator to manually create a Device Alert event in order to create a Device Alert Log entry.

**File Data Transfer Rate**

Low – The required data may be written to the SD Memory Card at a slow rate.

**File Data Storage**

Each Device Alert Log entry would comprise of:

**Patient Data Log:**

256 Byte record \* 30 minutes = 7680 Bytes = 7.5 KBytes

**Operator Settings Log:**

64 Byte record \* 240 changes in 30 minutes = 7200 Bytes = 7.04 KBytes

**Diagnostic Log:**   
 **System Diagnostic Log** (POST, Background, Soft Faults)

Assuming 500 errors in last 30 minutes = 500 \* 128 Bytes = 62.5 KBytes

**System Information Diagnostic Log** (Communication Errors, EST/SST Failures)

Assuming 500 errors in last 30 minutes = 500 \* 128 Bytes = 62.5 KBytes

**EST/SST Diagnostic Log** (EST/SST tests, EST/SST failures (not test results), Alerts)

Assuming 500 errors in last 30 minutes = 500 \* 128 Bytes = 62.5 KBytes

Providing a combined total = 187.5 KBytes

**Alarm Event Log:**

Assuming 500 alarms in last 30 minutes = 500 \* 272 Bytes = 132.82 KBytes

**Event Recorder Log:**

Record size = 300 bytes stored every 20ms.

Last 30 minutes = 300 \* 50 \* 60 \* 30 = 25.75 MByte

**Service Log**

Service log record = 4 KBytes

**Preventative Maintenance Log**

Preventative Maintenance log record = 4 KBytes

**In Summary**

Patient Data Log 7.50 KB

Operator Settings Log 7.04 KB  
 Diagnostic Log 187.50 KB  
 Alarm Event Log 132.82 KB  
 Event Recorder Log 26,368.00 KB  
 Service Log 4.00 KB

Preventative Maintenance Log 4.00 KB

--------------------------------------------------------------

Total = 26,710.86 KB = 26.09 MBytes / Device Alert Entry

**Overall Device Alert Log File Size**

If 60 Device Alert log entries were to be stored, the file size would be:

26.09 Mbytes \* 60 = 1.53 GBytes

### Data Storage Summaries

This section provides a summary of the total amount of data storage which is required along with the maximum expected data transfer bandwith.

#### Data Storage Summary

The Viking Data Storage requirements are as follows:

**Log File Name** **Minimum Storage Requirements**

Patient Data Log 1.47 MB (3 days data)

Alarm Log 66.4 KB (1000 alarms)

System Diagnostic Log 9 KB (256 entries)

System Information Log 9 KB (256 entries)

EST/ SST Diagnostic Log 9 KB (256 entries)

Operator Settings Log 31.25 KB (500 settings)

Detailed Alarm Log 2.13 MB (1000 alarms)

EST Log 5.00 KB

SST Log 5.00 KB

Calibration Log 5.00 KB

General Event Log 3.63 GB (72 Hours at 20ms samples)

SPI Backplane Event Log 13.38 GB (72 hours at 20ms samples)

Service Log 4.00 MB (1000 entries)

Preventative Maintenance Log 4.00 MB (1000 entries)

Device Alert Log 1.53 GB (60 entries)

**Total Data Storage Required:** **18.552 GBytes**

#### Data Storage Bandwidth

The Viking Data Storage Bandwidth requirements are as follows:

**Log File Name** **Bandwidth Requirements**

Patient Data Log 4.26 Bytes/Sec

Operator Settings Log 64.00 Bytes/Log Update

Alarm Log 272.00 Bytes/Log Update

Detailed Alarm Log 2224.00 Bytes/Log Update

EST Log 5.00KBytes/Log Update

SST Log 5.00Kbytes/Log Update

Calibration Log 5.00KBytes/Log Update

General Event Log **16.65KBytes/Sec**

SPI Backplane Event Log **54.11KBytes/Sec**

System Diagnostic Logs 2.50KBytes/Log Update

Service Log 4.00KBytes/Log Update

Preventative Maintenance Log 4.00KBytes/Log Update

Device Alert Log 26.09MBytes/Device Alert

**The total amount of simultaneous data storage bandwidth is: 69.48 Kbytes/sec.**

**Please Note:**

The above specified file bandwidths have been computed only considering the amount of data throughput to be transferred in one direction, namely to the SD Memory Card. When a file is modified, one or more records may be read from the SD Memory Card into a RAM buffer, one or more records may be modified in the RAM buffer and then the RAM buffer is written into the SD Memory Card. Such transactions will increase the above bandwidth requirements.

In addition, trending may be requesting large amounts of data for graphical display.

#### Data Retrieval Bandwidth

The Viking Data Retrieval Bandwidth requirements are as follows and are based upon the display of 100 records of data which may be displayed on Waveforms or Covidio:

**Log File Name** **Bandwidth Requirements**

Patient Data Log 2560 Bytes ( 2.500 KB/Sec)

Operator Settings Log 640 Bytes ( 0.625 KB/Sec)

General Event Log 30,000 Bytes ( 29.300 KB/Sec)

SPI Backplane Event Log 110,800 Bytes (108.204 KB/Sec)

The “Bandwidth” column assumes that this data is to be retrieve for display within a one second interval.

#### Data Storage / Retrieval Combined Bandwidth

The Data Storage / Retrieval operations which may occur simulataneously thereby creating bandwidth issues are as follows:

**Worst Case Simultaneous data storage bandwidth:**

**Log File Name** **Bandwidth Requirements**

General Event Log 16.65KBytes/Sec

SPI Backplane Event Log 54.11KBytes/Sec

**Worst Case Simultaneous data retrieval bandwidth (assuming 100 records read/second):**

**Log File Name** **Bandwidth Requirements**

General Event Log 30,000 Bytes ( 29.300 KB/Sec)

or

SPI Backplane Event Log 110,800 Bytes (108.204 KB/Sec)

The worst case total storage / retrieval combined bandwidth is **179 KB/Second.**

## Ventilator Logs Access and Use

### Normal Mode

### Service Mode

#### Directory Access

The User Interface provides a means whereby the directory structure may be displayed along with the files contained within a specific directory. The operator is also provided a means which allows a specific file to be selected and displayed.

#### File Display

The file display routine permits a file to be displayed in binary hex/ascii format.

This feature is useful during testing and the development of the ventilator logs.

# New Details

Testing requires a method whereby log data can be provided, such that it is appended to a log and afterwhich the data may be retrieved from the log thereby illustrating that the log was functional.

# Flash Implementation

In BD\_GUI\_SRC/FileSystem/groupsrc/FlashMediaDriver.cc

FlashMediaDriver::Initialize()

This function calls fx\_media\_open to open the Flash media.

To access the media we need the media\_ptr parameter.

To get this use

See for log info…

Bd\_gui\_src\Persistent-Objects\groupsrs\

AlarmLogEntry

AlarmHistoryLog

DiagCodeLog

Etc…………………….

# Database Initialization

## System Startup

When the GUI ventilator application software is started, the Logs Database must be verified.

The following situations may arise:

1. This is a new ventilator system and a new Logs Dababase must be created.
2. This is an existing ventilator system which contains a Logs Database which must be verified
3. This is an existing ventilator system which contains a damaged Logs Database which must be repaired.

### New Logs Database

When a new ventilator system has been created (or a new File System Media has been installed) a new Logs Database must be created. This process involves the creation of the necessary database directories and any initial required files.

### Existing Logs Database – Verification

When an existing ventilator system has been powered up, the application software proceeds to verify the Logs Database. This process involves checking that the database contains the correct directory structure and that the Log files have the correct revision. The usage of a configuration file provides the file Ids and associated versions currently used by the database file system.

New software revisions may occur at any time which may also require additional database directories. During the verification process any additional directories will be created. If one or more file revisions have taken place, the software will proceed to convert the older file formats to the required new file formats prior to allowing user access to the Log Database. This process would occur under the guidance of the Bio-tech engineer, immediately after the ventilator had undergone a software upgrade and would be successfully completed prior to the ventilator being delivered to a ward and used on a patient.

### Existing Logs Database – Repair

Since the Logs Database utilizes a Flash-based storage medium there is always the possibility that the database file system may need to be repaired. TBD.

## Database Verification Process

The Logs Database verification process is as follows:

If database base directory is missing

No Logs Database exists – Build new Logs Database

Verification process completed

Endif

Verify existence of all required Log Database directories (create new directories as required)

If log configuration file exists

Read configuration file

If any configuration file IDs use an old version

… Do the following in the Final System

Convert these specific files to the new format

**… Do the following while developing the Logs Database**

Delete the current Logs Database

Build new Logs Database

Verification process completed

Endf

Else (log configuration file does not exist)

**… Do the following while developing the Logs Database**

Delete the current Logs Database

Build new Logs Database

Verification process completed

For each Log Database Directory

For each file type in this directory

Read header and verify file ID and version

If old version found

**… Do the following in the Final System**

Convert these specific files to the new format

Endif

Enddo

Enddo

Create new configuration file

Endif

# Questions

Do we need checksums?

Do we need retries?

# Performing Unit Tests

## Enable Unit Tests

Enable Unit Tests by accessing the following file:

viking\_gui\UICommon\groupsrc\GuiEventMgr.hh

Go to line #32 and remove the comments “//”

#define ENABLE\_UNIT\_TESTS

## Create Unit Test

Create a Unit Test – see Control\_UT.cc as an example.

This module resides in: Viking\_gui\UnitTests\groupsrc\Control\_UT.cc

The Unit Test code may reside in the syslogs directory and consist of one or more functions.

Create a Unit Test control module, for example: Syslogs\_UT.cc

Within Syslogs\_UT.cc we would have the following code utilizing the TEST macro.

The CHECK macro calls the unit test function and checks for the required response value.

For example

TEST(Syslogs\_Test)

{

CHECK (syslogs\_UT\_verify\_database() == SYSLOGS\_SUCCESS)

… etc…

**return** UnitTest::DONE;

}

**NOTE**: Linker removes dead code – To avoid this place unit test call in UnitTests.cc

**void** **UnitTests::PreventDeadCodeRemoval**()

## Using Omnitool

If you need some comment to be displayed on the Omnitool screen

then use the following call within the Unit Test function:

check\_print()

The Omnitool may be found in the following directory:

viking\_gui\GuiIOServer\testsrc\Omnitool\Release

Copy FunkyLibrary.dll and Omnitool.exe into DOS directory.

Execute the Omnitool.exe utility.

# Final Files

This section identifies the files that are finalized and may be used.

**syslogs\_system.cc**

Delete syslogs database from system:

destroy\_database()

Create a new syslogs database system:

create\_database()

To verify that the existing syslogs database is valid:

verify\_database()

To ensure that an existing database is valid:

restore\_database()

printf ("verify\_database[1] access\_root\_directory() FAILED\n");

printf ("[%s #%d] Test\n", \_\_FILE\_\_, \_\_LINE\_\_);

# Questions

## Setting Log

Using a single file

Each time a setting is entered in the log the Insert and Extract pointers are adjusted.

If this is a new patient the patient ID is incremented

This implies that the file header is being modified each time.

This implies that a Flash block is being erased and programmed.

**Inserting data**

Read header

Write data record

Adjust pointers

Write header (increment patient ID if new patient)

**Extracting data**

Patient Data