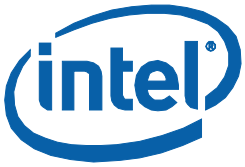


Intel® Quiet System Technology 2.0

Programmer's Reference Manual (PRM)

February 2010

Revision -001

Document Number: 323345

Contents

Figures

Tables

Revision History

Revision Number	Description	Revision Date
-001	• Initial release	February 2010

§

1 Introduction

1.1 Overview

The Intel® Management Engine (ME) hosts a firmware subsystem – *Intel® Quiet System Technology (QST)* – that provides support for the monitoring of temperature, voltage, current and fan speed sensors that are provided within the Chipset, the Processor and other devices on the Motherboard. For each sensor, a Health Status, based upon established thresholds, will be determined at regular intervals. Intel® QST also provides support for acoustically-optimized fan speed control. Based upon readings obtained from the temperature sensors, Intel® QST will determine, over time, the optimal speeds at which to operate the available cooling fans, in order to address existing thermal conditions with the lowest possible acoustic impact. Chapter 2 will present a more detailed overview of the capabilities of Intel® QST.

Support is provided for software applications to monitor and control the operation of Intel® QST. This is facilitated by two software APIs: a Communications API and a Health Monitoring API.

The Communications API, also referred to as the Communications Layer, provides software applications with the ability to:

- Monitor and control the configuration and operation of Intel® QST and the individual monitoring and control services that are contained within it.
- Monitor and control the physical sensor and fan speed control hardware.

The Health Monitoring API, also referred to as the Instrumentation Layer, provides software applications with the ability to:

- Monitor the configuration, readings and health status of the temperature, voltage and fan speed sensors that are monitored by Intel® QST.
- Monitor the configuration, settings and health status of the fan speed controllers managed by Intel® QST.

This document details how software developers may make use of the Intel® QST Communications and Health Monitoring APIs, in order to implement Diagnostics, Tuning and Health Monitoring applications:

- Chapter 3 details how applications utilize the services of the Communications Layer, in order to query and control the operation of Intel® QST.
- Chapter 4 details how applications utilize the services of the Instrumentation Layer, in order to implement hardware-independent health monitoring.
- Chapter 5 details how applications may directly query and control the sensor and fan speed controller hardware.

1.1.1 What's New in Intel® QST 2.0

Intel® QST 2.0 introduces significant changes in the QST command architecture. These changes have been predicated by a number of changes in the Intel® QST architecture and environment:

- Support has been added for the new, DTS-based processor thermal specification. In systems that include BIOS support for this specification, improved acoustics in high-power situations is possible. For more information on this feature, consult the Intel Developer Forum presentations detailed in Section 1.2.
- Support has been added for the architectural changes being made in 2009 platforms. This includes the migration of many Memory Controller Hub (MCH) features into the processor package and the introduction of the Platform Controller Hub (PCH), which combines the former features of the I/O Controller Hub (ICH) and the remainder of the MCH features (including the Intel® Management Engine (ME)).
- Support has been added for the larger Intel® QST configurations that are possible in Workstation/Server designs. Support for as many as 32 temperature sensors, 32 voltage sensors, 32 fan speed sensors and 32 fan speed controllers has been added.
- While firmware-level support for Current sensors will not be introduced until 2010, placeholder support for these sensors was defined in the new Intel® QST command set. Since significant changes in the command set will have far reaching affects on 3rd-party software applications that interface with Intel® QST, the decision was made to include this placeholder support now so that changes in subsequent generations can be minimized.

The changes in the command set being introduced in Intel® QST 2.0, as a result of the architectural and environmental changes, are summarized as follows:

- Previously, both command and entity were identified using a single (byte-wide) field in the command packet header. With the changes in command and entity requirements that are occurring, this single field is being replaced by two (byte-wide) fields that separately specify the command and entity identification.
- In order to allow existing applications to be used with Intel® QST 2.0, a Compatibility Layer is being added. This Compatibility Layer, embedded in the Intel® QST Communications Library (QstComm), allows the Intel® QST 1.x command interface (function QstCommand()) and a subset of the Intel® QST 1.x commands to be utilized on systems equipped with Intel® QST 2.0 firmware. It also allows the Intel® QST 2.0 command interface (function QstCommand2()) and a subset of the Intel® QST 2.0 commands to be utilized on systems equipped with Intel® QST 1.x firmware. Note the following:
  - Those Intel® QST commands that are related to configuration management are not supported by the Compatibility Layer. This includes the following functions:

    GetSubsystemConfiguration
    SetSubsystemConfiguration
    UpdateCPUConfiguration

GetCPUConfigurationUpdate
UpdateFanControllerConfiguration
GetFanControllerConfigurationUpdate

- New commands introduced in Intel® QST 2.0 are also not supported by the Compatibility Layer. This includes the following functions:

  UpdateCPUDTSConfiguration
  GetCPUDTSConfigurationUpdate
  GetCurrentMonitorUpdate
  GetCurrentMonitorConfiguration
  SetCurrentMonitorThresholds

- The configuration payloads supported by Intel® QST 1.x firmware were required to have the same number of entity structures that are supported by the firmware, regardless of how many of these structures are actually enabled in a particular configuration (i.e. 12 Temperature Monitors, 8 Fan Speed Monitors, 8 Voltage Monitors, 12 Temperature Responses and 8 Fan Controllers must be represented). With the significantly larger number of entities that may be supported in an Intel® QST 2.0 configuration payload and because firmware configurations can be built with differing levels of support (based upon environmental requirements – desktop vs. workstation, for example), unused entities are now optional.

  In order to support variable-length payloads, applications that process configuration payloads must either parse the compressed payload to map structure positions or expand it to a fixed size that allows simple indexing. To allow for the latter, support for expanding payload structures to specific limits is being provided. Similarly, to support the preparation of payloads from fixed-size structures, support for payload compression is also included. See Appendix B for more information on this capability.

1.2 Reference Documents

Document	Document No./ Location
A New Processor Temperature Specification: Using the Digital Temperature Sensor (Intel Corporation 2008)	developer.intel.com
Intel® Quiet System Technology (Intel® QST) – Revision 2.0 Updates (Intel Corporation 2008)	developer.intel.com
Fan Specification for 4-wire PWM Controlled Fans (Intel Corporation 2005)	www.formfactors.org
Advanced Technology Extended (ATX) Specification (Intel Corporation, 2004)	www.formfactors.org
Balanced Technology Extended (BTX) Interface Specification (Intel Corporation, 2005)	www.formfactors.org

1.3 Terminology

Term	Description
Intel® QST	Intel® Quiet System Technology – A subsystem contained with the Intel® 965 Express Chipset Family and the Intel® 3 Series, 4 Series and 5 Series Chipset Families, which provides environmental health monitoring and acoustically-optimized fan speed control.
Intel® Management Engine (ME)	A microcontroller built into Intel Chipset. It hosts firmware implementing support for various manageability technologies – Intel® QST, Intel® Active Management Technology (Intel® AMT), etc.
Fan Speed Sensors	Hardware circuits that measure the revolution speeds for fans connected to the motherboard. The chipset provides four of these circuits. How many of them are actually utilized is dependent upon the number of fan connectors provided by the motherboard, the number of fans plugged into these connectors, whether these fans provide support for fan speed monitoring and how Intel® QST is configured.
Temperature Sensors	Hardware circuits that measure temperatures at various monitoring points within the system. Temperature sensors are contained within the chipset (both the I/O Controller Hub (ICH) and Memory Controller Hub (MCH)) and within the Processor. Depending upon what other devices are included in the motherboard design, other temperature sensors may be available as well. Which of these sensors are available for monitoring is dependent upon the configuration of Intel® QST.
Voltage Sensors	Hardware circuits that measure the voltage levels being output from Voltage Regulators included on the motherboard and in the Power Supply. Whether any Voltage Sensors will actually be present is dependent upon the devices included in the motherboard/system design and how Intel® QST is configured.
Fan Speed Controllers	Hardware circuits that provide support for controlling the speed of fans connected to the motherboard. The chipset provides three of these circuits. How many are actually utilized is dependent upon the design of the motherboard and how Intel® QST is configured.
Duty Cycle	Fan Speed Controllers control fan speed using particular duty cycle values. The duty cycle is a specification of the percentage of full speed that fans are to be controlled to. It must be noted that fans do not necessarily react to duty cycle values in a linear fashion.
$T_{CONTROL}$	Upper Limit of a processor's normal operational temperature range. This is measured via the processor's on-die thermal diode, or Digital Thermometer temperature(s). This temperature value, available via Model-Specific Register (MSR), is determined during processor validation and fused into the processor. It is thus specific to each individual desktop processor. *Note:* Operation above a Processor's Tcontrol set point is allowed, but the thermal solution must meet the Processor's thermal profile.

Term	Description
Instrumentation Layer	A layer of software which instruments the capabilities and status of specific software and hardware entities contained within the system. The Intel® QST Instrumentation Layer instruments the temperature, voltage and fan speed sensors that are being monitored, as well as the fan speed controllers that are being managed.
DLL	Dynamic Link Library – A Windows* O/S feature that allows executable code modules to be loaded on demand and linked at run time. This enables the library-code fields to be updated automatically, transparent to applications, and then unloaded when they are no longer needed.
LoadLibrary()	Win32* function that loads the specified DLL into system memory and maps it into the address space of the calling process. Subsequent attempts by the calling process to load/map this DLL will result in its Reference Count being incremented.
FreeLibrary()	Win32 function that decrements the Reference Count for a loaded DLL. When the reference count reaches zero, the module is unmapped from the address space of the calling process. When the DLL is unmapped by all processes, it is unloaded from system memory.
GetLastError()	Win32 function that retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.
GetProcAddress()	Win32 function that retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).
Open Watcom	The open source versions of Sybase's Watcom C/C++ and Fortran compiler products. Use of this software is governed by the Sybase Open Watcom Public License, version 1.0. This package is used to generate 32-bit flat model code that can be used in the DOS environment (supported by DOS/4GW or PMODE/W; included in the package) to communicate with Intel® QST. The package can be obtained from www.openwatcom.org.

§

2 Intel® QST Overview

2.1 Introduction

This chapter provides an overview of the design of Intel® QST. It details the implementation for each of the different processes operating within the Subsystem. The discussion is broken down into two sections, Environmental Health Monitoring and Fan Speed Control.

2.2 Environmental Health Monitoring

Intel® QST 2.0 provides support for a number of types of Environmental Health Monitoring processes: Temperature Monitors, Fan Speed Monitors, Voltage Monitors and Current Monitors. Each is responsible for regularly obtaining readings from an associated sensor and then establishing an Environmental Health Status, based upon a comparison of readings obtained against configured Health Thresholds.

For each monitoring process type, the number actually in operation (enabled) is dependent upon the availability of sensors of the appropriate type. The following table provides information regarding the typical and maximum number of processes of each type:

Table 1: Health Monitoring Capabilities

Capability	Minimum Supported	Typically Supported	Maximum Supported
Temperature Sensors	2-3[1]	4-5[2]	32[6]
Voltage Sensors	0	5[3]	32
Current Sensors	0	0[4]	32
Fan Speed Sensors	4	4[5]	32

Notes:

1. Minimum support consists of the sensors in the PCH and the processor. If the MCH component in the processor also provides a sensor, this temperature will be exposed as well (the Monitor for the MCH sensor is automatically disabled if the processor does not include such a sensor).
2. Typical temperature support adds a SST Bus-based device providing two temperature sensors. One of these sensors is used as general motherboard temperature input; the other's usage is design-specific.
3. Typical voltage support consists of sensors for the main Power Supply voltage rails (12V, 5V and 3.3V) and motherboard-generated PCH Vcc and Processor Vcc (Vccp) voltages. These sensors are typically provided within the same SST Bus-based device that provides ambient and processor thermal diode temperature sensors.
4. While support for Current sensors is included in the Intel® QST infrastructure, no support for their exposure is included in the current firmware implementation.

5. Typical Fan Speed monitoring support consists of the four sensors provided by the PCH.
6. This is the absolute maximum supported; a particular build of the firmware could support lesser numbers. The 6.0.0.xxxx desktop firmware, for example, includes support for 12 temperature sensors, 8 voltage sensors, 0 current sensors and 8 fan speed sensors.

2.2.1 Health Monitoring Status

Environmental Health Status values are represented within a byte-wide variable. Its contents are defined as follows:

7	6	5	4	3	2	1	0
Reserved	STASCE	STASRE	STATST		STAMST		STAENA

Table 2: Environmental Health Status Byte Contents

Bit	Description
7	**Reserved by Intel.**
6	**STASCE – Sensor Communication Error.** This bit, if set (1), indicates that a permanent communications error exists with the device containing the associated Sensor.
5	**STASRE – Sensor Reading Error.** This bit, if set (1), indicates that an error (diode fault, etc.) is preventing the associated Sensor from returning valid readings.
4:3	**STATST – Threshold Status.** Indicates the status of the Monitor, with respect to the health thresholds defined: 00 – *Normal* (No Threshold Exceeded) 01 – *Non-Critical* Threshold Exceeded 10 – *Critical* Threshold Exceeded 11 – *Non-Recoverable* Threshold Exceeded Note: The contents of this field are indeterminate while the Monitor Status (STAMST) is outside the *Normal* state.
2:1	**STAMST – Monitor Status.** Indicates the overall status of the Monitor, with respect to operation. 00 – *Normal* (no sensor communication/operational issues) 01 – *Non-Critical* (a sensor communication/operational issue has occurred) 10 – *Critical* (multiple sensor communication/operational issues have occurred) 11 – *Non-Recoverable* (sensor is considered to be permanently in a bad state) For temperature monitors, entry into the non-recoverable state will result in all fans being run at their highest speed, in order to protect the system from undetectable thermal conditions.
0	**STAENA – Monitor Enabled.** If set (1), indicates that this Monitor is enabled. If cleared (0), indicating that the Monitor is disabled, the contents of the other fields will be 0.

Note: A structure definition has been provided in header file QstCmd.h that allows the fields of this byte to be individually addressed:

```
/*****************************************************************************/
/* QST_MON_HEALTH_STATUS - Monitor Health Status structure definition        */
/*****************************************************************************/

typedef struct _QST_MON_HEALTH_STATUS
{
   BIT_FIELD_IN_UINT8         bMonitorEnabled: 1;
   BIT_FIELD_IN_UINT8         uMonitorStatus: 2;
   BIT_FIELD_IN_UINT8         uThresholdStatus: 2;
   BIT_FIELD_IN_UINT8         bSensorReadingError: 1;
   BIT_FIELD_IN_UINT8         bSensorCommError: 1;

} QST_MON_HEALTH_STATUS;

// Health Status (uMonitorStatus, uThresholdStatus field) Values

#define QST_STATUS_NORMAL               0
#define QST_STATUS_NON_CRITICAL         1
#define QST_STATUS_CRITICAL             2
#define QST_STATUS_NON_RECOVERABLE      3
```

2.2.2 Temperature Monitors

The Temperature Monitor processes are responsible for the regular monitoring of specific temperature sensors and the comparison of temperature values obtained against programmed over-temperature thresholds. Two types of Temperature Monitors may be defined: Normal Temperature Monitors, which have a specific temperature sensor directly associated with them, and Virtual Temperature Monitors, which have no physical sensor directly associated with them.

2.2.2.1 Software-Assisted (Virtual) Temperature Support

In many systems, additional temperature sensor devices are present but not directly accessible by Intel® QST. Situations could thus arise wherein specific hardware components begin overheating, yet Intel® QST is unaware of the situation. If software running on the Host Processor was aware of this situation, it could only react by overriding Intel® QST's fan speed control. Doing so, however, would prevent Intel® QST from properly responding to temperature inputs that it *does* have access to.

Specific examples of this type of temperature sensor are the sensors that are present within most Hard Disk Drives and exposed by the Self-Monitoring Analysis and Reporting Technology (S.M.A.R.T.). Temperature values from these sensors are accessible from software, but are not exposed by any hardware access methodology that could be monopolized by Intel® QST.

In order to address this type of situation, the concept of a Virtual Temperature Monitor was introduced into Intel® QST. Virtual Temperature Monitors differ from standard Temperature Monitors on only one aspect: they have no physical temperature sensor associated with them. Instead, software running on the host processor must regularly provide these Monitors with updated temperature readings as changes occur.

Note: A total of 32 Temperature Monitors are available within Intel® QST. Any number of these may be configured for Virtual Temperature Monitoring.

2.2.2.2 Relative Temperature Readings

In addition to traditional temperature sensors, which return temperature readings in absolute form (°C, for example), support is provided for sensors that return readings in relative form. Relative temperature sensors provide temperature readings that are relative to (are a ± offset from) some particular temperature set point.

Beginning with the Intel® Core™ 2 processor family, each processor core contains a Digital Thermal Sensor (DTS), which returns temperature readings that are relative to the processor's Thermal Control Circuit (TCC) activation temperature. In Desktop processors, the TCC activation temperature varies from one individual processor to another. Prior to the Intel® Core™ i7 processor, no process existed for determining each processor's TCC activation temperature. As a result, temperatures could only be represented in relative form. Beginning with the Intel® Core™ i7 processor family, however, support for determining the TCC activation temperature for an individual processor was added; the Intel® QST 2.0 firmware utilizes this information to calculate and expose processor temperatures that are in absolute form.

2.2.3 Fan Speed Monitors

The Fan Speed Monitor processes are responsible for the regular monitoring of specific fan speed sensors and the comparison of fan speed readings with configured under-speed thresholds. Since many fans are speed-controlled, the Fan Speed Monitors are cognizant of the fact that fans can be purposefully stopped. Thus, the Fan Speed Monitors also interrogate the Fan Speed Control processes, in order to make this determination and not report spurious Environmental Health exceptions.

2.2.4 Voltage Monitors

The Voltage Monitor processes are responsible for the regular monitoring of specific voltage sensors and the comparison of the voltage values obtained against programmed over- and under-voltage thresholds, in order to establish their health status.

2.2.5 Current Monitors

As mentioned previously, placeholder support have been added to the Intel® QST infrastructure for Current Sensors. The QST 2.0 firmware does not include support for monitoring any Current Sensors, however.

2.3 Fan Speed Control

Intel® QST provides two types of processes that collectively implement support for acoustically-optimized fan speed control, *Temperature Response Units* and *Fan Controllers*. Based upon readings obtained from the monitored temperature sensors, these processes will collectively determine the optimal speeds at which to operate the available cooling fans, in order to address the current thermal situation with the lowest possible acoustic impact.

For each of these process types, the number actually in operation (enabled) is dependent upon the number of temperature sensor inputs to be analyzed and the number of fan speed controllers available. The following table provides information regarding the typical and maximum number of processes of each type:

Table 3: Fan Speed Control Capabilities

Capability	Minimum Supported	Typically Supported	Maximum Supported
Temperature Response	2-3[1]	4-5[1]	32
Fan Speed Controllers	4[2]	4	32

Notes:
1. Support is provided for the same sets of temperature sensors discussed previously.
2. The PCH provides four fan speed (PWM) controllers.

2.3.1 Temperature Response Units

Temperature Response Units are responsible for analyzing changes in temperature, as measured by the associated Temperature Monitors, and determining the appropriate responses needed from the fan control system. This is accomplished using the Proportional-Integral-Derivative (PID) and Ambient Feedback Limit Control components of Intel® QST's acoustically-optimized fan speed control algorithms. For more information on these algorithm components, please consult the Configuration and Tuning Manual.

2.3.2 Fan Controllers

Fan Controller processes are responsible for managing the delivery of duty cycle updates to the associated fan speed controllers. They apply the contents of the Weighting Matrix component of the Acoustically-Optimized Fan Speed Control algorithms against the temperature responses determined by the Temperature Response Units. The Weighting Matrix balances the use of the available fan speed controllers (and attached fans), to minimize the acoustic cost of addressing the current thermal situation. For more information on this process, please consult the Configuration and Tuning Manual.

Fan Controllers will run in a number of modes, depending upon the configuration, the commands received and the current thermal conditions:

- In Automatic Mode, a Fan Controller automatically determines the appropriate duty cycle response necessary from the associated fan speed controller, based upon measured temperature responses.

- In Manual (Software Control) Mode, a Fan Controller applies the duty cycle values that have been commanded by software applications.
- In Override Mode, which occurs during critical thermal, fan stall or sensor failure situations, associated Fan Controllers are responsible for the application of a 100% duty cycle value.

2.3.3 Fan Controller Health Status

Fan Controllers also maintain a Health Status for their operation. This status is represented in a byte variable, like that produced by the Monitoring processes, but the contents of this variable differ. The content of this byte is defined as follows:

7	6	5	4	3	2	1	0
Reserved	FSCOFS	FSCOTS	FSCOFC	FSCOSW	FSCHST		FSCENA

Table 4: Fan Controller Health Status Byte Contents

Bit	Description
7	**Reserved by Intel.**
6	**FSCOFS – Override Fan Stall.** This bit, if set (1), indicates that the Fan Controller's Duty Cycle output has been temporarily overridden to 100% as a result of a stall condition in one or more of the fan(s) controlled by this Fan Controller. In order to (attempt to) recover the use of this fan, the Duty Cycle is taken to 100% for a period of time to try and overcome any inertia issues. If cleared (0), this bit indicates that no fan stall recovery is in progress.
5	**FSCOTS – Override Temperature Sensor.** This bit, if set (1), indicates that the Fan Controller's Duty Cycle output has been overridden to 100% as a result of a failure in one of the temperature sensors. Without knowledge of temperatures, all fans are run at 100%, in order to avoid (undetectable) thermal overrun. If cleared (0), this bit indicates that no temperature sensor has suffered a failure.
4	**FSCOCE – Override Fan Controller.** This bit, if set (1), indicates that the Fan Controller's Duty Cycle output has been overridden to 100% as a result of a failure in one of the fan speed controllers. If cleared (0), this bit indicates that no fan speed controller has suffered a failure.
3	**FSCOSW – Override Software.** This bit, if set (1), indicates that the Fan Controller is being overridden by Software. That is, software has requested that the Fan Controller be placed under their control and have specified a duty cycle value that the Fan Controller is to output to the associated hardware fan Speed Controller. If cleared (0), this bit indicates that no software override is in progress.
2:1	**FSCHST – Controller Status.** Indicates the health state of the Fan Controller: 00 – Normal – No issues with the Fan Controller. 01 – Non-Critical – A single attempt to program the Controller has failed. 10 – Critical – Two attempts to program the Controller have failed. 11 – Non-Recoverable – Four attempts to program the Controller failed.
0	**FSCENA – Controller Enabled.** This bit, if set (1), indicates that this Fan Controller is enabled. If cleared (0), this bit indicates that this Controller is disabled. In this case, the content of the remaining fields will be 0.

Note: A structure definition has been provided in header file QstCmd.h that allows the fields of the Fan Controller Health Status byte to be individually addressed:

```
/*****************************************************************************/
/* QST_FAN_CTRL_STATUS - Fan Controller Status structure definition          */
/*****************************************************************************/

typedef struct _QST_FAN_CTRL_STATUS
{
   BIT_FIELD_IN_UINT8         bControllerEnabled: 1;
   BIT_FIELD_IN_UINT8         uControllerStatus: 2;
   BIT_FIELD_IN_UINT8         bOverrideSoftware: 1;
   BIT_FIELD_IN_UINT8         bOverrideFanController: 1;
   BIT_FIELD_IN_UINT8         bOverrideTemperatureSensor: 1;
   BIT_FIELD_IN_UINT8         bOverrideFanStall: 1;
   BIT_FIELD_IN_UINT8         uFiller: 1;

} QST_FAN_CTRL_STATUS;
```

2.4 Header File Overview

Throughout this document, definitions from the provided header files will be utilized to demonstrate how to perform specific operations. Included are:

QstComm.h	Provides definitions specific to the use of the Intel® QST Communications Layer. See Chapter 3 for more information.
QstCmd.h	Provides definitions specific to the commands that are used to communicate with Intel® QST. See Chapter 3 and Chapter 5 for more information.
QstCmdLeg.h	Provides definitions specific to the Intel® QST 1.x command set. It is provided to support the development of applications that can interoperate with both Intel® QST 1.x and Intel® QST 2.0 firmware.
QstCfg.h	Provides definitions specific to the configuration of Intel® QST. See Appendix A for more information.
QstCfgLeg.h	Provides definitions specific to the Intel® QST 1.x configuration payload. It is provided to support the development of applications that can interoperate with both Intel® QST 1.x and Intel® QST 2.0 firmware and will process Intel® QST configuration payloads.
QstInst.h	Provides definitions specific to the use of the Intel® QST Instrumentation Layer. See Chapter 4 for more information.

The following subsections provide general information on the use of these header files…

2.4.1 Special Data Types

Unfortunately, the Intel® Management Engine provides no support for floating point math or associated data representations. In order to represent values to specific precision, Intel® QST utilizes two special data types, INT32F and INT32LF.

2.4.1.1 INT32F

Specifying temperature values and thresholds, as well as calculated duty cycle (percentage) values requires accuracy to at least two decimal places. In order to represent values with two decimal places, the INT32F data type was introduced. This data type is basically a 32-bit signed two's-complement Integer value that has two implied decimal places. For example, a value of 25 would be stored as 2500 (0x09C4) and a value of 16.5 would be stored as 1650 (0x0672).

2.4.1.2 INT32LF

The support for the DTS-based thermal specification requires accuracy to at least four decimal places. In order to represent values to four decimal places, the INT32LF data type was introduced. This data type is basically a 32-bit signed two's-complement Integer value that has four implied decimal places. For example, a value of 25 would be stored as 250000 (0x3D090) and a value of 16.125 would be stored as 161250 (0x275E2)

2.4.2 Bit Fields

In order to minimize the size of messages, configuration structures, etc., bit fields are used wherever possible. Utilizing a portable representation for bit fields presents a challenge, due to the quirks – and even bugs – that exist within certain C compilers. It was discovered that these bugs could be avoided by utilizing the fixed field length language extension feature that is supported by many C/C++ Compilers (Microsoft*, Borland*, etc.). Three bit field types have been defined, BIT_FIELD_IN_UINT8, BIT_FIELD_IN_UINT16 and BIT_FIELD_IN_UINT32, which support bit fields in 8-, 16- and 32-bit quantities, respectively. Activation of the fixed field length language extension is accomplished with the following declarations:

```
/*****************************************************************************/
/* Bit field counting is buggy in certain C/C++ compilers. We will handle   */
/* this by maximizing the use of the fixed field length language extension  */
/* that is supported by many compilers, including Borland's and Microsoft's */
/* (16- and 32-bit versions). Note that this language extension is enabled  */
/* by default; if you disable it, compilation errors will abound!!          */
/*****************************************************************************/

#if defined(__WIN32__) || defined(__MSDOS__)

#define BIT_FIELD_IN_UINT8      UINT8
#define BIT_FIELD_IN_UINT16     UINT16
#define BIT_FIELD_IN_UINT32     UINT32

#else

#define BIT_FIELD_IN_UINT8      unsigned
#define BIT_FIELD_IN_UINT16     unsigned
#define BIT_FIELD_IN_UINT32     unsigned

#endif
```

3 Monitoring and Controlling Intel® QST

This Chapter details how Software Applications can monitor and control the operation of Intel® QST. It details how communications is facilitated, how applications use supporting software components in order to communicate, what commands are available and under what circumstances these commands are available.

3.1 Communicating with Intel® QST

3.1.1 Overview

In order to facilitate communications between software executing on the Host Processor and firmware executing on the Intel® Management Engine, bidirectional communications buffers and supporting hardware interfaces are provided to both processors. This hardware is collectively referred to as the Intel® Management Engine Interface (MEI), but is often referred to by its code name: HECI (Host to Embedded Controller Interface).

The Communications Layer (QstComm Library) facilitates software with support for using the Intel® MEI to communicate with the Intel® QST firmware. How this is done is dependent upon the O/S in use:

- For Windows* environments, support is facilitated by two software entities: a device driver, which handles the complexities of the Intel® MEI, and a Dynamic Link Library (QstComm.dll), which handles the complexities of the ME communications protocols and interfacing with the Intel® MEI Device Driver.
- For Linux and Solaris environments, support is facilitated by two software entities: a device driver, which handles the complexities of the Intel® MEI, and a Shared-Object File (QstComm.so), which handles the complexities of the ME communications protocols and interfacing with the Intel® MEI Device Driver.
- For DOS environments, support is facilitated by a software library (QstComm6x.lib), which handles both the complexities of the Intel® MEI and the complexities of the ME communications protocols.

Note: Your motherboard manufacturer will distribute Windows* versions of the Intel® MEI Driver with its products. The Linux/Solaris driver can be downloaded from http://www.openamt.org.

3.1.2 Using Windows* QstComm DLL

The Windows* QstComm DLL provides the API necessary for Windows-based software applications to communicate with Intel® QST. It provides support for sending command packets to and receiving response packets from the Intel® QST command processor.

Note: This API also provides support for direct communications with the Sensor and Controller devices that are being managed by Intel® QST. See Section 3.3 for details regarding this capability.

3.1.2.1 DLL Function Overview

The QstComm DLL automatically handles the connections between calling processes and the Intel® MEI Device Driver. Consequently, only a single function is actually needed to perform communications with Intel® QST. Two versions of this function are now supported, one that supports the Intel® QST 1.x command set (QstCommand()) and one that supports the Intel® QST 2.0 command set (QstCommand2()). These functions are used to send command packets to Intel® QST and receive the associated response packets returned. Command and response packets are documented in sections 3.1.3 and 0.

3.1.2.2 DLL Function Access

Applications may either statically or dynamically link themselves with the QstComm DLL.

3.1.2.2.1 Static Linking

Statically linked DLLs are automatically loaded into the address space of the application when the application is itself loaded and will remain there until the application is terminated. Applications may invoke the functions of the DLL just as they would any other functions.

In order to statically link the QstComm DLL with their applications, developers should include **QstComm.lib** file in their applications' project. This file is created during the build of QstComm.dll with the Intel® QST SDK.

Note: The use of static linking is NOT recommended. If something prevents the DLL from loading, this will also cause the load of the application to fail – and will do so without providing any error indication …

3.1.2.2.2 Dynamic Linking

Dynamically linked DLLs are manually loaded into an application's address space when the application actually requires their services. Manual loading of DLLs is normally used only when applications utilize their services intermittently and wish to minimize their memory footprint where possible.

In order to load the QstComm DLL, applications use WIN32* function LoadLibrary(). Once the DLL has been loaded, pointers to the DLL functions must be built using

WIN32 function GetProcAddress(). The functions may then be invoked indirectly through these pointers.

When the application is finished using the services of the DLL, it should unload it using WIN32 function FreeLibrary(). Alternatively, the O/S will automatically unload the DLL when the application is terminated.

Example

The following code example demonstrates how one can facilitate the dynamic load, use and unload of the QstComm DLL:

```
/*****************************************************************************/
/* Variables                                                                 */
/*****************************************************************************/

static HMODULE            hQstCommDLL = NULL;
static PFN_QST_COMMAND    pfQstCommand;
static PFN_QST_COMMAND    pfQstCommand2;

/*****************************************************************************/
/* QstInitialize() - Initializes I/F for accessing service of the DLL        */
/*****************************************************************************/

BOOL QstInitialize( void )
{
   // Load the DLL

   hQstCommDLL = LoadLibrary( QST_COMM_DLL );

   if( hQstCommDLL )
   {
      // Build pointer to the DLL's function(s)

      pfQstCommand = (PFN_QST_COMMAND)GetProcAddress(
                        hQstCommDLL, MAKEINTRESOURCE( QST_COMM_ORD ) );

      if( pfQstCommand )
      {
         pfQstCommand2 = (PFN_QST_COMMAND)GetProcAddress(
                           hQstCommDLL, MAKEINTRESOURCE( QST_COMM2_ORD ) );

         if( pfQstCommand2 )
            return( TRUE );
      }

      FreeLibrary( hQstCommDLL );
      hQstCommDLL = NULL;
   }

   return( FALSE );
}
```

```
/*****************************************************************************/
/* QstCleanup() - Cleans up I/F to DLL                                       */
/*****************************************************************************/

void QstCleanup( void )
{
   if( hQstCommDLL )
   {
      FreeLibrary( hQstCommDLL );
      hQstCommDLL = NULL;
   }
}

/*****************************************************************************/
/* QstCommand() - Sends 1.x command to QST Subsystem and awaits response     */
/*****************************************************************************/

BOOL APIENTRY QstCommand( void *pvCmdBuf, size_t tCmdSize,
                          void *pvRspBuf, size_t tRspSize  )
{
   if( hQstCommDLL )
      return( pfQstCommand( pvCmdBuf, tCmdSize, pvRspBuf, tRspSize ) );

   SetLastError( ERROR_DLL_NOT_FOUND );
   return( FALSE );
}

/*****************************************************************************/
/* QstCommand2() - Sends 2.0 command to QST Subsystem and awaits response    */
/*****************************************************************************/

BOOL APIENTRY QstCommand2( void *pvCmdBuf, size_t tCmdSize,
                           void *pvRspBuf, size_t tRspSize  )
{
   if( hQstCommDLL )
      return( pfQstCommand2( pvCmdBuf, tCmdSize, pvRspBuf, tRspSize ) );

   SetLastError( ERROR_DLL_NOT_FOUND );
   return( FALSE );
}
```

Note: Prototypes for functions QstInitialize() and QstCleanup() have been included in header file QstComm.h. In order to use these prototypes, define symbol DYNAMIC_DLL_LOADING in your project file and include example source file QstComm.c (which provides the code shown above).

3.1.3 Using Linux*/Solaris* QstComm Shared-Object (SO) Files

The Linux/Solaris QstComm Shared-Object (SO) Files (libQstComm.so) provide the API necessary for Linux-/Solaris-based software applications to communicate with Intel® QST. It provides support for sending command packets to and receiving response packets from the Intel® QST command processor.

Note: This API also provides support for direct communications with the Sensor and Controller devices that are being managed by Intel® QST. See Chapter 3.3 for details regarding this capability.

Note: For more information regarding SO files, consult the *Program Library HOWTO*. Additional information is included in the *C++ dlopen mini HOWTO*. Both can be downloaded from the Linux Documentation Project (http://tldp.org/docs.html).

3.1.3.1 SO File Function Overview

The QstComm SO File automatically handles the connections between calling processes and the Intel® MEI Device Driver. Consequently, only a single function is actually needed to perform communications with Intel® QST. Two versions of this function are now supported, one that supports the Intel® QST 1.x command set (QstCommand()) and one that supports the Intel® QST 2.0 command set (QstCommand2()). These functions are used to send Command Packets to Intel® QST and receive the associated Response Packets returned. These Packets are documented in sections 3.1.5 and 3.1.6.

3.1.3.2 SO File Function Access

Applications may either statically or dynamically link themselves with the QstComm SO.

3.1.3.2.1 Static Linking

Statically linked SO Files are automatically loaded into the address space of the application when the application is itself loaded and will remain there until the application is terminated. Applications may invoke the functions of the SO File just as they would any other functions.

In order to statically link the QstComm DLL with their applications, developers should include the libQstComm.so file into their applications' project. This is accomplished by, for example, adding the "-lQstComm" parameter to their GCC invocation.

3.1.3.2.2 Dynamic Linking

Dynamically linked SO Files are manually loaded into an application's address space when the application actually requires their services. Manual loading of SO Files is normally used only when applications utilize their services intermittently and wish to minimize their memory footprint where possible.

SO Files are dynamically loaded and unloaded using the dlopen API. Function dlopen() is used to load the SO file. Once loaded, function dlsym() is used to obtain a pointer to functions QstCommand() and QstCommand2(). These functions can then be invoked indirectly through the pointers provided. Finally, when the program is finished with the SO file, it unloads it using function dlclose().

Example

The following code example demonstrates how one can facilitate the dynamic load, use and unload of the QstComm SO File:

```
/*****************************************************************************/
/* Variables                                                                 */
/*****************************************************************************/

typedef BOOL (*PFN_QST_COMMAND)( void *pvCmdBuf, size_t tCmdSize,
                                 void *pvRspBuf, size_t tRspSize );

static void *             hQstCommSO = NULL;
static PFN_QST_COMMAND    pfQstCommand, pfQstCommand2;
```

```
/*****************************************************************************/
/* QstCommand() - Sends 1.x command to QST Subsystem and awaits response    */
/*****************************************************************************/

BOOL APIENTRY QstCommand( void *pvCmdBuf, size_t tCmdSize,
                          void *pvRspBuf, size_t tRspSize  )
{
   if( hQstCommSO )
      return( pfQstCommand( pvCmdBuf, tCmdSize, pvRspBuf, tRspSize ) );

   errno = ENOEXEC;
   return( FALSE );
}

/*****************************************************************************/
/* QstCommand() - Sends 2.0 command to QST Subsystem and awaits response    */
/*****************************************************************************/

BOOL APIENTRY QstCommand2( void *pvCmdBuf, size_t tCmdSize,
                           void *pvRspBuf, size_t tRspSize  )
{
   if( hQstCommSO )
      return( pfQstCommand2( pvCmdBuf, tCmdSize, pvRspBuf, tRspSize ) );

   errno = ENOEXEC;
   return( FALSE );
}

/*****************************************************************************/
/* QstCleanup() - Cleans up I/F to the SO File                               */
/*****************************************************************************/

void QstCleanup( void )
{
   if( hQstCommSO )
   {
      dlclose( hQstCommSO );
      hQstCommSO = NULL;
   }
}

/*****************************************************************************/
/* QstInitialize() - Initializes I/F for accessing services of the SO File  */
/*****************************************************************************/

BOOL QstInitialize( void )
{
   // Load the SO File

   hQstCommSO = dlopen( “libQstComm.so.1”, RTLD_LAZY );

   if( hQstCommSO )
   {
      // Build pointer to SO File's function(s)

      dlerror();
      pfQstCommand = (PFN_QST_COMMAND)dlsym( hQstCommSO, “QstCommand” );

      if( dlerror() == NULL )
      {
         pfQstCommand2 = (PFN_QST_COMMAND)dlsym( hQstCommSO, “QstCommand2” );

         if( dlerror() == NULL )
            return( TRUE );
      }

      dlclose( hQstCommSO );
      hQstCommSO = NULL;
      errno = ENOENT;
   }
   else
      errno = ENODEV;

   return( FALSE );
}
```

3.1.4 Using DOS QstComm Library

The DOS QstComm Library provides the API necessary for DOS-based software applications to communicate with QST. It provides support for sending command packets to and receiving response packets from the QST command processor.

In order to support access to the Intel® MEI hardware, software must be built for a 32-bit address space and must be supported by a DOS Extender that eliminates the inherent DOS 640KB memory limit. Support is provided for building applications using Open Watcom, an open source version of Sybase's Watcom C/C++ compiler product. In addition to its support for 32-bit address spaces, this package also includes support for multiple license-free DOS Extenders.

Note: Intel® QST support has been successfully tested using both the DOS4GW and PMode/W. Use of Pmode/W avoids the need to include DOS4GW.exe, which occupies 260KB of disk space and a similar amount of DOS conventional memory. Your ultimate decision regarding which to use will depend upon the design and features of your applications...

Note: Applications will be built using the flat memory model. Versions of the DOS QstComm Library are provided for register-based parameter passing (QstComm6r.lib) and stack-based parameter passing (QstComm6s.lib).

Note: This API also provides support for direct communications with the Sensor and Fan Speed Controller devices that are being managed by QST. See Chapter 5 for details regarding this capability.

3.1.4.1 DOS Library Function Overview

The QstComm Library provides support for a single connection between the calling program and QST. Three functions from a set of four must be invoked to perform communications with QST: QstInitialize(), QstCommand(), QstCommand2() and QstCleanup(). QstCommand() and QstCommand2() are used to send Command Packets to QST and receive the associated Response Packets; QstCommand() supports Intel® QST 1.x Command/Response Packets and QstCommand2()supports the Intel® QST 2.0 Command/Response Packets. Command and Response Packets are documented in sections 3.1.3 and 0.

3.1.4.2 DOS Library Function Access

Applications will statically link themselves with the DOS QstComm Libraries. The following example batch file shows how both debug and release executables may be built for a program:

```
set OLDINCLUDE=%INCLUDE%
set OLDPATH=%PATH%
set STDOPTS=/mf /wx /hw /fp6 /6r /bt=dos /i=..\..\Include /i=..\..\Common

if exist Build.log del Build.log

rem Try the two standard places for Open Watcom to be installed...
rem C:\WATCOM\...
```

```
if exist C:\watcom\setvars.bat call C:\watcom\setvars.bat
rem C:\PROGRAM FILES\WATCOM\...
if exist C:\progra~1\watcom\setvars.bat call C:\progra~1\watcom\setvars.bat

set BUILD=Release
set OPTS=/ox /d0

goto DOBUILD

:DEBUG
set BUILD=Debug
set OPTS=/od /d2 /dTRACE

:DOBUILD
set TEMPDIR=DOS\%BUILD%

if     exist %TEMPDIR%    del /f /q %TEMPDIR%
if not exist %TEMPDIR%    md %TEMPDIR%

rem Compile all source module(s)...

wcc386 Test.c /fo=%TEMPDIR%\Test.obj %OPTS% %STDOPTS% >>Build.log
if errorlevel 1 goto ERROR

rem Link the program...

if “%BUILD%” == “Debug” echo debug dwarf >%TEMPDIR%\Build.lnk

echo disable  108                                  >>%TEMPDIR%\Build.lnk
echo @%WATCOM%\binnt\wlink.lnk                     >>%TEMPDIR%\Build.lnk
echo option   dosseg                               >>%TEMPDIR%\Build.lnk
echo system   pmodew                               >>%TEMPDIR%\Build.lnk
echo file     %TEMPDIR%\Test                       >>%TEMPDIR%\Build.lnk
echo library  ..\..\Libraries\DOS\Release\QstInst6r >>%TEMPDIR%\Build.lnk
echo library  ..\..\Libraries\DOS\Release\QstComm6r >>%TEMPDIR%\Build.lnk
echo name     %TEMPDIR%\Test.exe                   >>%TEMPDIR%\Build.lnk
echo option   map=%TEMPDIR%\Test.map               >>%TEMPDIR%\Build.lnk

wlink @%TEMPDIR%\Build >>Build.log
if errorlevel 1 goto ERROR

if “%BUILD%” == “Release” goto DEBUG

%WATCOM%\binnt\wdis %TEMPDIR%\Test /l=%TEMPDIR%\Test.lst /s=Test.c >>Build.log
if errorlevel 1 goto ERROR

echo Build Successful!!
Goto DONE

:ERROR
echo Build Failed!!

:DONE
set INCLUDE=%OLDINCLUDE%
set PATH=%OLDPATH%
```

3.1.5 Command Packets

Command packets are used to encapsulate commands and any associated command data. Table 5 details the content of command packets.

Note: As mentioned previously, there are differences in the format and content of the command packets for Intel® QST 1.x and Intel® QST 2.0. This document restricts itself to the details for the Intel® QST 2.0 command set. Consult previous versions of this document for information regarding the format of Intel® QST 1.x command packets and the Intel® QST 1.x command set.

Table 5: Command Packet Format

Byte Offset	Description
00h	Command Code
01h	Entity Index
02h-03h	Command Data Size
04h-05h	Response Data Size
06h+	Command Data (optional)

3.1.5.1 Command Code

This field is used to indicate what type of command the application desires processed by Intel® QST. Section 3.2.36 details the available commands and associated Command Codes. Definitions for the available command codes is provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST Subsystem Command Codes                                               */
/*****************************************************************************/

#define QST_GET_SUBSYSTEM_INFO              0x00
#define QST_GET_SUBSYSTEM_STATUS            0x01
#define QST_GET_SUBSYSTEM_CONFIG            0x02
#define QST_GET_SUBSYSTEM_CONFIG_PROFILE    0x03
#define QST_SET_SUBSYSTEM_CONFIG            0x04
#define QST_LOCK_SUBSYSTEM                  0x05
#define QST_UPDATE_CPU_CONFIG               0x06
#define QST_GET_CPU_CONFIG_UPDATE           0x07
#define QST_UPDATE_CPU_DTS_CONFIG           0x08
#define QST_GET_CPU_DTS_CONFIG_UPDATE       0x09
#define QST_UPDATE_FAN_CONFIG               0x0A
#define QST_GET_FAN_CONFIG_UPDATE           0x0B
#define QST_SST_PASS_THROUGH                0x0C

#define QST_GET_TEMP_MON_UPDATE             0x0D
#define QST_GET_TEMP_MON_CONFIG             0x0E
#define QST_SET_TEMP_MON_THRESHOLDS         0x0F
#define QST_SET_TEMP_MON_READING            0x10
#define QST_NO_TEMP_MON_READINGS            0x11

#define QST_GET_FAN_MON_UPDATE              0x12
#define QST_GET_FAN_MON_CONFIG              0x13
#define QST_SET_FAN_MON_THRESHOLDS          0x14
#define QST_ENABLE_FAN_MON                  0x15
#define QST_DISABLE_FAN_MON                 0x16
#define QST_REDETECT_FAN_PRESENCE           0x17

#define QST_GET_VOLT_MON_UPDATE             0x18
#define QST_GET_VOLT_MON_CONFIG             0x19
#define QST_SET_VOLT_MON_THRESHOLDS         0x1A

#define QST_GET_CURR_MON_UPDATE             0x1B
#define QST_GET_CURR_MON_CONFIG             0x1C
#define QST_SET_CURR_MON_THRESHOLDS         0x1D

#define QST_GET_FAN_CTRL_UPDATE             0x1E
#define QST_GET_FAN_CTRL_CONFIG             0x1F
#define QST_SET_FAN_CTRL_DUTY               0x20
#define QST_SET_FAN_CTRL_AUTO               0x21
#define QST_RESET_FAN_CTRL_MIN_DUTY         0x22

#define QST_LAST_CMD_CODE                   0x22
```

3.1.5.2 Entity Index

This field is used to indicate the (0-31) index of the particular entity that the command is specific to. For commands that are not specific to a particular entity, this field must be set to 0 (zero).

3.1.5.3 Command Data Size

This field is used to indicate how many bytes of data are included in the command packet. This does not include the space occupied by the packet header. Section 3.2.36 details the data that is associated with each command type.

3.1.5.4 Response Data Size

This field is used to indicate the size of the expected response packet. Intel® QST 2.0 will restrict its response to the size specified, even if the command is capable of returning more data than requested. Response packets are at least one byte in length, in order to return a Response (status) Code.

Note: Specifying that the response packet will contain zero bytes will result in no response packet being generated; this means that the application will not receive a response code and thus will not know whether the command was accepted or rejected. Control will be returned to the application immediately after the transmission of the command packet.

3.1.5.5 Packet Definition

Definitions for a generic command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_CMD_HEADER - Command header structure definition                      */
/*****************************************************************************/

typedef struct _QST_CMD_HEADER
{
   UINT8                        byCommand;
   UINT8                        byEntity;
   UINT16                       wCommandLength;
   UINT16                       wResponseLength;

} QST_CMD_HEADER, *P_QST_CMD_HEADER;

/*****************************************************************************/
/* QST_GENERIC_CMD - Generic command structure definition                    */
/*****************************************************************************/

typedef struct _QST_GENERIC_CMD
{
   QST_CMD_HEADER               stHeader;

} QST_GENERIC_CMD, *P_QST_GENERIC_CMD;

// Useful Macros

#define QST_CMD_DATA_SIZE(x)    (sizeof(x) - sizeof(QST_CMD_HEADER))
```

The generic command packet is typically used to deliver commands that are requesting that Intel® QST return some specific piece of data; only commands that

deliver some specific piece of data to Intel® QST require a custom packet definition (see section 3.2 for more details).

3.1.6 Response Packets

Response packets are used to encapsulate the Response (status) Code and any data returned by the command. These packets contain the following information:

Table 6: Response Packet Format

Byte Offset	Description
00h	Response Code
01h+	Response Data (optional)

3.1.6.1 Response Code

Responses to all commands received back from the Intel® QST will include, at a minimum, a byte field providing a Response (status) Code. This Response Code indicates whether or not the command was accepted and successfully processed. The following response codes have been defined:

Table 7: Response Codes

Error Code	Description
00h	Command Completed Successfully.
01h	Command Rejected: Unsupported Command Code.
02h	Command Rejected: Configuration Locked.
03h	Command Rejected: Invalid Parameter.
04h	Command Rejected: Invalid Version.
05h	Command Failed: Communications Error.
06h	Command Failed: Sensor Error.
07h	Command Failed: Insufficient Memory.
08h	Command Failed: Insufficient Resources.
09h	Command Rejected: Invalid command data.
10h	Command Rejected: Invalid command data size.
11h	Command Rejected: Invalid response data size.
12h	Command Rejected: Invalid in present context.

The following definitions for the response codes are provided in header file QstCmd.h:

```
/*****************************************************************************/
/* Response Codes                                                            */
```

```
/*****************************************************************************/

#define QST_CMD_SUCCESSFUL                  0x00
#define QST_CMD_REJECTED_UNSUPPORTED        0x01
#define QST_CMD_REJECTED_LOCKED             0x02
#define QST_CMD_REJECTED_PARAMETER          0x03
#define QST_CMD_REJECTED_VERSION            0x04
#define QST_CMD_FAILED_COMM_ERROR           0x05
#define QST_CMD_FAILED_SENSOR_ERROR         0x06
#define QST_CMD_FAILED_NO_MEMORY            0x07
#define QST_CMD_FAILED_NO_RESOURCES         0x08
#define QST_CMD_REJECTED_INVALID            0x09
#define QST_CMD_REJECTED_CMD_SIZE           0x0A
#define QST_CMD_REJECTED_RSP_SIZE           0x0B
#define QST_CMD_REJECTED_CONTEXT            0x0C
```

3.1.6.2 Packet Definition

Definition for a generic response packet is provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GENERIC_RSP - Generic response structure definition                    */
/*****************************************************************************/

typedef struct _QST_GENERIC_RSP
{
   UINT8                        byStatus;

} QST_GENERIC_RSP, *P_QST_GENERIC_RSP;
```

The generic response packet is typically used to return the Response Code for commands that deliver some specific piece of data to Intel® QST. Only those commands that request some specific piece of data from Intel® QST will require a custom response packet definition (see section 3.2 for more details).

3.1.7 Security Considerations (Intel® QST Lock Mask)

In order to prevent rogue software from disaffecting its operation, Intel® QST implements an access-level security feature. A Lock Mask is used to specify what operations runtime software applications may request. The BIOS controls the setting of the Lock Mask. During its Power-on Self-Test (POST) processing, the BIOS will communicate the Lock Mask that is to be enforced to Intel® QST.

Software applications that wish to control Intel® QST must be aware of what commands the BIOS will allow it to perform. The contents of the Lock mask, a 16-bit quantity, are defined as follows:

15	14	13	12	11	10	9	8
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

7	6	5	4	3	2	1	0
Reserved	Reserved	Reserved	LCKCHP	LCKMFC	LCKTHR	LCKSST	LCKCFG

Table 8: Subsystem Lock Mask Contents

Bit	Description
15:4	**Reserved by Intel.**
4	**LCKCHP – Lock Chipset.** This bit, if set (1), locks access to the Chipset. In this case, commands attempting to write to Chipset-based resources (temperature sensors, fan speed sensors and fan speed controllers) will be rejected. If clear (0), this bit leaves the chipset unlocked. In this case, commands attempting to write to Chipset-based resources will be accepted and processed.
3	**LCKMFC – Lock Manual Fan Control.** This bit, if set (1), locks access to the Fan Controller Processes. In this case, commands requesting manual duty cycle control will be rejected. If clear (0), this bit leaves the Fan Controller Processes accessible. In this case, commands requesting manual duty cycle control will be accepted and processed.
2	**LCKTHR – Lock Thresholds.** This bit, if set (1), locks access to the Temperature, Voltage and Fan Speed Monitor's Health Thresholds. In this case, all commands requesting changes to these thresholds will be rejected (health monitoring software can only read thresholds). If clear (0), this bit indicates that the Monitor thresholds are accessible. In this case, all commands requesting changes to thresholds will be accepted and processed.
1	**LCKSST – Lock SST Bus.** This bit, if set (1), locks access to SST Bus-based devices. In this case, all commands attempting to write to these devices will be rejected. If clear (0), this bit indicates that the SST Bus is unlocked. In this case, all commands attempting to write to these devices will be processed.
0	**LCKCFG – Lock Configuration.** This bit, if set (1), locks the FSC Configuration. In this case, all commands requesting configuration changes will be rejected. If clear (0), this bit indicates that the FSC Configuration is unlocked. In this case, commands requesting configuration changes will be accepted and processed.

The typical Lock Mask will set bits LCKCFG, LCKSST, LCKMFC and LCKCHP and will reset bit LCKTHR. This will secure the Subsystem against rogue software:

1. Disabling/stopping fans.
2. Disrupting the operation (configuration) of the fan speed controllers and fan speed sensors.
3. Destroying or modifying the overall configuration and operation of Intel® QST.
4. Disrupting the operation of devices on the SST bus.

At the same time, leaving LCKTHR clear will allow the thresholds for Temperature, Fan Speed and Voltage Monitor health determination to be adjusted by run-time software (manageability) applications, in order to handle the nuances of the system design/usage.

The following definition for the Lock Mask is provided in header file QstCmd.h:

```
/*****************************************************************************/
/* QST_LOCK_MASK - Lock Mask structure definition                            */
/*****************************************************************************/

typedef struct _QST_LOCK_MASK
{
   BIT_FIELD_IN_UINT16          bLockConfiguration: 1;
   BIT_FIELD_IN_UINT16          bLockSSTBusAccess: 1;
   BIT_FIELD_IN_UINT16          bLockThresholds: 1;
   BIT_FIELD_IN_UINT16          bLockManualFanControl: 1;
```

```
    BIT_FIELD_IN_UINT16          bLockChipsetAccess: 1;
    BIT_FIELD_IN_UINT16          uReserved: 11;

} QST_LOCK_MASK;
```

3.1.8 QstComm Library Functions

3.1.8.1 QstInitialize()

This function is used to initialize support for communicating with QST. It should be invoked by all DOS applications, by those Windows* applications that dynamically link to the QstComm DLL and by those Linux*/Solaris* applications that dynamically link to the QstComm SO File. Windows* and Linux/Solaris applications that statically link to the QstComm library (DLL or SO File) do not need to invoke this function.

Invocation:

BOOL QstInitialize(void);

Returns:

If the function succeeds, the return value is 1 (TRUE). If the function fails, the return value is zero (FALSE). From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS and Linux applications, extended error information is provided via the *errno* variable.

Windows* Exception Codes:

All exception codes returned via Win32 function GetLastError() are the result of a failure occurring during the use of Win32 functions LoadLibrary() or GetProcAddress().

Linux*/Solaris* Exception Codes:

The *errno* variable will receive one of the following exception codes:

ENODEV	The QstComm SO File (libQstComm.so.1) could not be located or loaded.
ENOENT	Could not locate the symbol for function QstCommand().

DOS Exception Codes:

The *errno* variable will receive one of the following exception codes:

ENXIO	The Intel® MEI could not be located in PCI address space. An unsupported chipset is present.
EIO	An I/O error occurred while attempting to initialize the INTEL® MEI.
ENOMEM	Buffer space to support communications transaction could not be allocated.

3.1.8.2 QstCommand()

This function is used to transmit an Intel® QST 1.x Command Packet to the Intel® QST firmware and obtain the Response Packet returned. Provided that this is not a command specific to Intel® QST configuration management, if the Intel® QST firmware supports the Intel® QST 2.0 command set, the QstComm library will translate this command into the appropriate Intel® QST 2.0 command, deliver it to the firmware for processing and translate the response packet returned back into the Intel® QST 1.x format.

Invocation:

```
BOOL   QstCommand
(
    IN     void          *pvCmdBuf,
    IN     size_t        tCmdSize,
    OUT    void          *pvRspBuf,
    IN     size_t        tMaxRspSize
);
```

Input Parameters:

pvCmdBuff	Pointer to a buffer that contains the Command Packet that is to be transmitted to Intel® QST for processing.
tCmdSize	Specifies the size, in bytes, of the Command Packet to be transmitted.
tMaxRspSize	Specifies the size, in bytes, of the buffer into which the Response Packet will be stored.

Output Parameters:

pvRspBuf	Pointer to a buffer into which the Response Packet will be stored. This parameter may be NULL if no response is desired.

Returns:

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. WIN32 function GetLastError() can be used to obtain extended error information. From DOS and Linux applications, extended error information is provided via the *errno* variable.

Windows* Exception Codes:

This function performs a cursory verification of the integrity of the command packet before sending it to QST. As a result of this verification, any of the following exception codes may be exposed via Win32 function GetLastError( ):

ERROR_BAD_LENGTH	The Command or Response Packet Size parameters request the transmission of packets that are larger than can be supported by the communications facility, or the size of the Response Packet received is inconsistent with the requested Response Packet Size.
ERROR_BAD_COMMAND	The Command Code specified in the Command Packet is invalid.
ERROR_BAD_FORMAT	The Command/Response Data Size parameters within the Command Packet are inconsistent with the Command/Response Packet sizes specified or, for embedded SST (Pass-Through) commands, the SST Write/Read Length parameters is/are inconsistent with the Command/Response Packet size or Command/Response Data Size parameters.

Any other exception codes that are returned are the result of Win32 functions that are invoked by the library.

Linux*/Solaris* Exception Codes:

EFAULT	Valid buffers were not provided.
EINVAL	The Command or Response Packet Size parameters request the transmission of packets that are larger than can be supported by the communications facility, or the size of the Response Packet received is inconsistent with the requested Response Packet Size.
EPERM	The Command Code specified in the Command Packet is invalid.
ERANGE	The Command/Response Data Size parameters contained within the Command Packet is/are inconsistent with the Command/Response Packet sizes specified or, for embedded SST (Pass-Through) commands, the SST Write/Read Length parameters is/are inconsistent with the Command/Response Packet size or Command/Response Data Size parameters.
ENOSPC	The response packet returned by QST is larger than the buffer supplied by the calling routine.

Other exception codes that are returned are the result of standard library functions that are invoked by the library.

DOS Exception Codes:

This function performs a cursory verification of the integrity of the command packet before sending it to QST. As a result of this verification, any of the following exception codes may be exposed via the *errno* variable:

ENXIO	The INTEL® MEI interface is not initialized or no buffer is available to support communications. The application ignored the return code from QstInitialize();
EFAULT	Valid buffers were not provided.
EINVAL	The Command or Response Packet Size parameters request the transmission of packets that are larger than can be supported by the communications facility, or the size of the Response Packet received is inconsistent with the requested Response Packet Size.
EPERM	The Command Code specified in the Command Packet is invalid.
ERANGE	The Command/Response Data Size parameters contained within the Command Packet is/are inconsistent with the Command/Response Packet sizes specified or, for embedded SST (Pass-Through) commands, the SST Write/Read Length parameters is/are inconsistent with the Command/Response Packet size or Command/Response Data Size parameters.
EIO	An I/O error occurred within the INTEL® MEI interface.
ENOSPC	The response packet returned by QST is larger than the buffer supplied by the calling routine.

Other exception codes that are returned are the result of standard library functions that are invoked by the library.

3.1.8.3 QstCommand2()

This function is used to transmit an Intel® QST 2.0 Command Packet to the Intel® QST firmware and obtain the Response Packet returned. Provided that this is not a command specific to Intel® QST configuration management, if the Intel® QST firmware supports the Intel® QST 1.x command set, the QstComm library will translate this command into the appropriate Intel® QST 1.x command, deliver it to the firmware for processing and translate the response packet returned back into the Intel® QST 2.0 format.

Invocation:

```
BOOL   QstCommand
(
    IN     void          *pvCmdBuf,
    IN     size_t        tCmdSize,
    OUT    void          *pvRspBuf,
    IN     size_t        tMaxRspSize
);
```

Input Parameters:

pvCmdBuff	Pointer to a buffer that contains the Command Packet that is to be transmitted to Intel® QST for processing.
tCmdSize	Specifies the size, in bytes, of the Command Packet to be transmitted.
tMaxRspSize	Specifies the size, in bytes, of the buffer into which the Response Packet will be stored.

Output Parameters:

pvRspBuf	Pointer to a buffer into which the Response Packet will be stored. This parameter may be NULL if no response is desired.

Returns:

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. WIN32 function GetLastError() can be used to obtain extended error information. From DOS and Linux applications, extended error information is provided via the *errno* variable.

Windows* Exception Codes:

This function performs a cursory verification of the integrity of the command packet before sending it to QST. As a result of this verification, any of the following exception codes may be exposed via Win32 function GetLastError():

ERROR_BAD_LENGTH	The Command or Response Packet Size parameters request the transmission of packets that are larger than can be supported by the communications facility, or the size of the Response Packet received is inconsistent with the requested Response Packet Size.

ERROR_BAD_COMMAND	The Command Code specified in the Command Packet is invalid.
ERROR_BAD_FORMAT	The Command/Response Data Size parameters within the Command Packet are inconsistent with the Command/Response Packet sizes specified or, for embedded SST (Pass-Through) commands, the SST Write/Read Length parameters is/are inconsistent with the Command/Response Packet size or Command/Response Data Size parameters.

Any other exception codes that are returned are the result of Win32 functions that are invoked by the library.

Linux*/Solaris* Exception Codes:

EFAULT	Valid buffers were not provided.
EINVAL	The Command or Response Packet Size parameters request the transmission of packets that are larger than can be supported by the communications facility, or the size of the Response Packet received is inconsistent with the requested Response Packet Size.
EPERM	The Command Code specified in the Command Packet is invalid.
ERANGE	The Command/Response Data Size parameters contained within the Command Packet is/are inconsistent with the Command/Response Packet sizes specified or, for embedded SST (Pass-Through) commands, the SST Write/Read Length parameters is/are inconsistent with the Command/Response Packet size or Command/Response Data Size parameters.
ENOSPC	The response packet returned by QST is larger than the buffer supplied by the calling routine.

Other exception codes that are returned are the result of standard library functions that are invoked by the library.

DOS Exception Codes:

This function performs a cursory verification of the integrity of the command packet before sending it to QST. As a result of this verification, any of the following exception codes may be exposed via the *errno* variable:

ENXIO	The INTEL® MEI interface is not initialized or no buffer is available to support communications. The application ignored the return code from QstInitialize();
EFAULT	Valid buffers were not provided.

EINVAL	The Command or Response Packet Size parameters request the transmission of packets that are larger than can be supported by the communications facility, or the size of the Response Packet received is inconsistent with the requested Response Packet Size.
EPERM	The Command Code specified in the Command Packet is invalid.
ERANGE	The Command/Response Data Size parameters contained within the Command Packet is/are inconsistent with the Command/Response Packet sizes specified or, for embedded SST (Pass-Through) commands, the SST Write/Read Length parameters is/are inconsistent with the Command/Response Packet size or Command/Response Data Size parameters.
EIO	An I/O error occurred within the INTEL® MEI interface.
ENOSPC	The response packet returned by QST is larger than the buffer supplied by the calling routine.

Other exception codes that are returned are the result of standard library functions that are invoked by the library.

3.1.8.4 QstCleanup()

This function is used to free the resources that are used to communicate with the Intel® QST firmware. It should be invoked by all DOS applications, by those Windows* applications that dynamically link to the QstComm DLL and by those Linux/Solaris applications that dynamically link to the QstComm SO File. Windows* and Linux/Solaris applications that statically link to the QstComm library (DLL or SO File) do not need to invoke this function.

Invocation:

```
void    QstCleanup( void );
```

3.2 Intel® QST 2.0 Commands

This section documents the Intel® QST 2.0 Commands that BIOS and runtime applications may communicate to the Intel® QST firmware. It details the Command and Response Packets for each command, including the parameters and fields contained therein. Commands are broken down into three categories:

1. Commands specific to the Subsystem's status and configuration
2. Commands specific to the BIOS' role in the configuration and operation of the Subsystem
3. Commands specific to the services operating within the Subsystem.

Note: While certain commands were specifically created to be used by the BIOS, runtime software is not specifically precluded from using these commands (barring security concerns). Further, the BIOS is not in any way precluded from using other commands as well. In fact, it might even be considered *normal* for BIOS to be using other commands. For example, a particular BIOS might chose to present a scene in BIOS Setup that displays readings (and possibly health status) from the various sensors and fan speed controllers. It is not uncommon for BIOS to implement manual fan detection and fan spin validation operations as well…

Note: If you are preparing and sending commands to Intel® QST from DOS (or from within the BIOS) and are not utilizing (or are modifying) the driver code provided in the Intel® QST SDK, then be aware that the length of the packet header imposed by the Intel® Management Engine Interface (Intel MEI / HECI) needs to be taken into account when performing communications. This header is NOT documented within this manual.

3.2.1 GetSubsystemInformation

This command is used to obtain basic information about the Intel® QST firmware subsystem.

Table 9: GetSubsystemInformation Command Packet

Byte Offset	Description	Contents
00h	Command Code	00h
01h	Entity Index	0
02h-03h	Command Data Size	0
04h-05h	Response Data Size	14

Table 10: GetSubsystemInformation Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Major Version Number
02h	Minor Version Number
03h	Revision Number
04-05h	Build Number
06h	Supported Temperature Monitors
07h	Supported Fan Speed Monitors
08h	Supported Voltage Monitors
09h	Supported Current Monitors
0Ah	Supported Temperature Responses
0Bh	Supported Fan Speed Controllers
0Ch-0Dh	Maximum Command Size

3.2.1.1 Firmware Revision

The Intel® QST firmware revision number is provided broken down into its constituent components: Major Version Number, Minor Version Number, Revision Number and Build Number. When displayed, these are typically presented separated by dots. As an example, at the time this document was being updated, the latest firmware revision was 6.0.0.1060.

3.2.1.2 Supported Temperature Monitors

This field specifies the maximum number of Temperature Monitoring entities that are supported by the current firmware build.

3.2.1.3 Supported Fan Speed Monitors

This field specifies the maximum number of Fan Speed Monitoring entities that are supported by the current firmware build.

3.2.1.4 Supported Voltage Monitors

This field specifies the maximum number of Voltage Monitoring entities that are supported by the current firmware build.

3.2.1.5 Supported Current Monitors

This field specifies the maximum number of Current Monitoring entities that are supported by the current firmware build.

3.2.1.6 Supported Temperature Responses

This field specifies the maximum number of Temperature Response entities that are supported by the current firmware build.

3.2.1.7 Supported Fan Speed Controllers

This field specifies the maximum number of Fan Speed Controller entities that are supported by the current firmware build.

3.2.1.8 Maximum Command Size

This field specifies the maximum size for any Intel® QST command or response packet. It is actually the size of the command/response buffers that are supported by the current firmware build.

3.2.1.9 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used to deliver it. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_SUBSYSTEM_INFO_RSP - Command response structure definition.      */
/*****************************************************************************/

typedef struct _QST_GET_SUBSYSTEM_INFO_RSP
{
   UINT8                        byStatus;
   UINT8                        uMajorVersionNumber;
   UINT8                        uMinorVersionNumber;
   UINT8                        uRevisionNumber;
   UINT16                       uBuildNumber;
   UINT8                        uSuppTempMonitors;
   UINT8                        uSuppFanMonitors;
   UINT8                        uSuppVoltMonitors;
   UINT8                        uSuppCurrMonitors;
   UINT8                        uSuppTempResponses;
   UINT8                        uSuppFanControllers;
   UINT16                       uMaxCmdSize;

} QST_GET_SUBSYSTEM_INFO_RSP, *P_QST_GET_SUBSYSTEM_INFO_RSP;
```

3.2.2 GetSubsystemStatus

This command is used to obtain an overall status for Intel® QST. Its Command and Response Packets are defined as follows:

Table 11: GetSubsystemStatus Command Packet

Byte Offset	Description	Contents
00h	Command Code	01h
01h	Entity Index	0
02h-03h	Command Data Size	0
04h-05h	Response Data Size	8

Table 12: GetSubsystemStatus Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Subsystem Status
02h-03h	Subsystem Lock Mask (see Section 3.1.7)
04h-07h	Configuration Status

3.2.2.1 Subsystem Status

This 8-bit field provides the overall status of the Intel® QST Subsystem. The individual bits of this field each provide specific details about the condition of the Subsystem's operation.

7	6	5	4	3	2	1	0
Reserved	Reserved	Reserved	Reserved	FSCOFF	FSCOFC	FSCOFE	FSCCFG

Table 13. Subsystem Status Field Contents

Bit	Description
7:4	**Reserved by Intel.**
3	**FSCOFF – Override Full Failure.** This bit, if set (1), indicates that all configured fans have been overridden to their maximum speeds, due to an inability to control the operation of one or more Fan Speed Controller.
2	**FSCOFC – Override Full Critical.** This bit, if set (1), indicates that all configured fans have been overridden to their maximum speeds, due to a critical temperature situation being detected.
1	**FSCOFE – Override Full Error.** This bit, if set (1), indicates that all fans configured have been overridden to their maximum speeds, due to an inability to obtain temperature readings from one or more Temperature Sensor.

Bit	Description
0	**FSCCFG – Subsystem Configured.** This bit, if set (1), indicates that the subsystem has been configured and is actively monitoring sensor health and controlling fan speed. If reset (0), this bit indicates that the subsystem has not been successfully configured. In this case, all fans will be running at their maximum speed.

Note: Bit **FSCCFG** provides an indication of whether or not Intel® QST has a valid configuration and is executing against this configuration. This is independent of any attempts to configure or reconfigure the Subsystem. If the Subsystem has a valid configuration and a failed attempt is made to deliver a new configuration, the Subsystem will revert to executing against the original valid configuration. As a result, it is possible to see the *Subsystem Status* indicate that the Subsystem has a valid configuration yet also see the *Configuration Status* indicate that a failed configuration attempt has occurred.

3.2.2.2 Configuration Status

This 32-bit field provides information about the most recent attempt to apply a configuration. This could be the processing of a configuration payload included in the flash image or a configuration payload received from one of the configuration tools via the command interface.

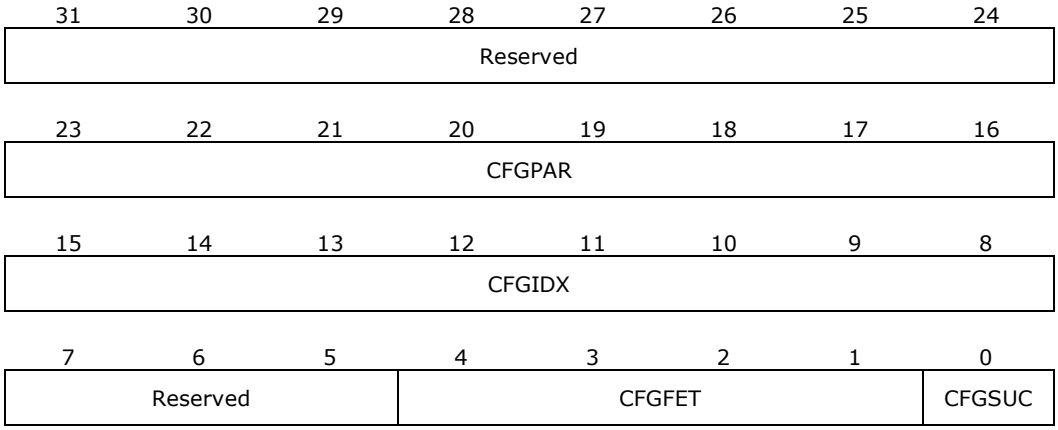


Table 14: Configuration Status Contents

Bit	Description
31:24	**Reserved by Intel.**
23:16	**CFGPAR – Failing Entity Parameter.** If a configuration failure occurred (CFGFET ≠ 7), this field will indicate the index of the entity parameter which was rejected.
15:8	**CFGIDX – Failing Entity Index.** If a configuration failure occurred (CFGFET ≠ 7), this field will indicate the index of the entity whose configuration was unsuccessful.
7:5	**Reserved by Intel**
4:1	**CFGFET – Failing Entity Type.** This field will indicate the type of entity whose configuration was unsuccessful. Values 0 will indicate that an error was detected in the Payload Header Structure. Values 1-6 will indicate that an error was detected in Temperature Monitor, Fan Monitor, Voltage Monitor, Current Monitor, Temperature Response and Fan Controller Structures, respectively. If no errors were detected in the configuration, this field will be set to -1 (01111b).
0	**CFGSUC – Configuration Successful.** If set (1), indicates that a configuration attempt has taken place. In this case, the other fields contain valid data. If clear (0), no configuration attempt has taken place.

Note: Bit **FSCCFG** in the *Subsystem Status* provides an indication of whether or not the Intel® QST Subsystem has a valid configuration and is executing against this configuration. This is independent of any (subsequent) attempts made to configure or reconfigure the Subsystem. If the Subsystem has a valid configuration and a failed attempt is made to deliver a new configuration, the Subsystem will revert to executing against the original valid configuration. As a result, it is possible to see the *Subsystem Status* indicate that the Subsystem has a valid configuration yet also see the *Configuration Status* indicate that a failed configuration attempt has occurred.

3.2.2.3 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used to deliver it. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SUBSYSTEM_STATUS - Subsystem Status byte structure definition         */
/*****************************************************************************/

typedef struct _QST_SUBSYSTEM_STATUS
{
   BIT_FIELD_IN_UINT8           bSubsystemConfigured: 1;
   BIT_FIELD_IN_UINT8           bOverrideFullError: 1;
   BIT_FIELD_IN_UINT8           bOverrideFullCritical: 1;
   BIT_FIELD_IN_UINT8           bOverrideFullFailure: 1;
   BIT_FIELD_IN_UINT8           uReserved: 4;

} QST_SUBSYSTEM_STATUS, *P_QST_SUBSYSTEM_STATUS;

/*****************************************************************************/
/* QST_LOCK_MASK - Lock Mask structure definition                            */
/*****************************************************************************/

typedef struct _QST_LOCK_MASK
{
   BIT_FIELD_IN_UINT16          bLockConfiguration: 1;
   BIT_FIELD_IN_UINT16          bLockSSTBusAccess: 1;
   BIT_FIELD_IN_UINT16          bLockThresholds: 1;
```

```
    BIT_FIELD_IN_UINT16          bLockManualFanControl: 1;
    BIT_FIELD_IN_UINT16          bLockChipsetAccess: 1;
    BIT_FIELD_IN_UINT16          uReserved: 11;

} QST_LOCK_MASK;

/*****************************************************************************/
/* QST_CONFIG_STATUS - Configuration Status byte structure definition       */
/*****************************************************************************/

typedef struct _QST_CONFIG_STATUS
{
    BIT_FIELD_IN_UINT32          bConfigSuccessful: 1;
    BIT_FIELD_IN_UINT32          uFailingEntityType: 4;
    BIT_FIELD_IN_UINT32          uReserved1: 3;
    BIT_FIELD_IN_UINT32          uFailingEntityIndex: 8;
    BIT_FIELD_IN_UINT32          uFailingEntityParam: 8;
    BIT_FIELD_IN_UINT32          uReserved2: 8;

} QST_CONFIG_STATUS;

/*****************************************************************************/
/* QST_GET_SUBSYSTEM_STATUS_RSP - Command response structure definition.    */
/*****************************************************************************/

typedef struct _QST_GET_SUBSYSTEM_STATUS_RSP
{
    UINT8                        byStatus;
    QST_SUBSYSTEM_STATUS         stSubsystemStatus;
    QST_LOCK_MASK                stLockMask;
    QST_CONFIG_STATUS            stConfigStatus;

} QST_GET_SUBSYSTEM_STATUS_RSP, *P_QST_GET_SUBSYSTEM_STATUS_RSP;
```

3.2.3 GetSubsystemConfiguration

This command is used to obtain the current configuration from the Intel® QST Subsystem.

Table 15: GetSubsystemConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	02h
01h	Entity Index	0
02h-03h	Command Data Size	0
04h-05h	Response Data Size	<as required>

Table 16: GetSubsystemConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h-????h	Subsystem Configuration (see Appendix A)

3.2.3.1 Response Data Size

Since Intel® QST configuration payloads vary in size, depending upon the support levels of the firmware and the architecture of the system (sensors and controllers present), it is recommended that the application provide a buffer large enough to handle the maximum configuration. It is recommended that the maximum command size, obtained using the GetSubsystemInfo command, be used to allocate a buffer of sufficient size.

3.2.3.2 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used to deliver it. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_SUBSYSTEM_CONFIG_RSP - Command response structure definition.     */
/* Note: QST_PAYLOAD_STRUCT is defined in Header file QstCfg.h.              */
/*****************************************************************************/

typedef struct _QST_GET_SUBSYSTEM_CONFIG_RSP
{
   UINT8                        byStatus;
   QST_ABS_PAYLOAD_STRUCT       stConfigPayload;

} QST_GET_SUBSYSTEM_CONFIG_RSP, *P_QST_GET_SUBSYSTEM_CONFIG_RSP;
```

See Appendix A for more information on the contents of the Binary Configuration Payload.

Note: The configuration payload returned will typically be in compressed form and may need to be decompressed to make access to its contents simpler. The Intel® QST SDK provides source modules showing how to expand and compress configuration payloads.

3.2.4 GetSubsystemConfigurationProfile

This command is used to obtain a profile of the configuration of the Intel® QST Subsystem. The command's response includes a set of bitmaps. Each bitmap indicates which entities (Temperature, Fan Speed, Voltage and Current Monitors, Temperature Response Units and Fan Speed Controllers) of a particular type have been configured and are in operation.

Table 17: GetSubsystemConfigurationProfile Command Packet

Byte Offset	Description	Contents
00h	Command Code	03h
01h	Entity Index	0
02h-03h	Command Data Size	0
04h-05h	Response Data Size	25

Table 18: GetSubsystemConfigurationProfile Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h-04h	Temperature Monitors Configured
05h-08h	Fan Speed Monitors Configured
09h-0Ch	Voltage Monitors Configured
0Dh-10h	Current Monitors Configured
11h-14h	Temperature Response Units Configured
15H-18H	Fan Controllers Configured

3.2.4.1 Temperature Monitors Configured

This 32-bit field indicates which Temperature Monitors are currently configured and enabled. Bits 0 through 31 respectively specify whether (1) or not (0) Temperature Monitors 1 through 32 are currently configured and enabled.

3.2.4.2 Fan Speed Monitors Configured

This 32-bit field indicates which Fan Speed Monitors are currently configured and enabled. Bits 0 through 31 respectively specify whether (1) or not (0) Fan Speed Monitors 1 through 32 are currently configured and enabled.

3.2.4.3 Voltage Monitors Configured

This 32-bit field indicates which Voltage Monitors are currently configured and enabled. Bits 0 through 31 respectively specify whether (1) or not (0) Voltage Monitors 1 through 32 are currently configured and enabled.

3.2.4.4 Current Monitors Configured

This 32-bit field indicates which Current Monitors are currently configured and enabled. Bits 0 through 31 respectively specify whether (1) or not (0) Current Monitors 1 through 32 are currently configured and enabled.

3.2.4.5 Temperature Response Units Configured

This 32-bit field indicates which Temperature Response Units are currently configured and enabled. Bits 0 through 31 respectively specify whether (1) or not (0) Temperature Response Units 1 through 32 are currently configured and enabled.

3.2.4.6 Fan Controllers Configured

This 32-bit field indicates which Fan Speed Controllers are currently configured and enabled. Bits 0 through 31 respectively specify whether (1) or not (0) Fan Speed Controllers 1 through 32 are currently configured and enabled.

3.2.4.7 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used to deliver it. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
    /*****************************************************************************/
    /* QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP - Command response structure         */
    /* definition                                                                */
    /*****************************************************************************/

    typedef struct _QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP
    {
       UINT8                        byStatus;
       UINT32                       dwTempMonsConfigured;
       UINT32                       dwFanMonsConfigured;
       UINT32                       dwVoltMonsConfigured;
       UINT32                       dwCurrMonsConfigured;
       UINT32                       dwTempRespConfigured;
       UINT32                       dwFanCtrlsConfigured;

    } QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP,
*P_QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP;

    // Useful Macros

    #ifndef SET_BIT
    #define SET_BIT(var,bit)        ((var) |= 1 << (bit))
    #define BIT_SET(var,bit)        (((var) & (1 << (bit))) != 0)
    #define CLR_BIT(var,bit)        ((var) &= (~(1 << (bit))))
    #endif
```

3.2.5 SetSubsystemConfiguration

This command is used to set the configuration of Intel® QST. It will only be accepted if the current Lock Mask permits configuration update (see section 3.1.7 for more details).

Table 19: SetSubsystemConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	04h
01h	Entity Index	0
02h-03h	Command Data Size	<as required>
04h-05h	Response Data Size	1
06h-????h	Subsystem Configuration	<as required>

Table 20: SetSubsystemConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.5.1 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_SUBSYSTEM_CONFIG_CMD - Command structure definition.              */
/* Note: QST_PAYLOAD_STRUCT is defined in Header file QstCfg.h.              */
/*****************************************************************************/

typedef struct _QST_SET_SUBSYSTEM_CONFIG_CMD
{
   QST_CMD_HEADER               stHeader;
   QST_ABS_PAYLOAD_STRUCT       stConfigPayload;

} QST_SET_SUBSYSTEM_CONFIG_CMD, *P_QST_SET_SUBSYSTEM_CONFIG_CMD;
```

Note: It is recommended that the configuration payload be compressed before sending it to the Intel® QST firmware. The Intel® QST SDK provides source modules showing how to both expand and compress configuration payloads.

3.2.6 LockSubsystem

This command will be sent by the BIOS, at some point during its Power-On Self-Test (POST) process, to lock (disable) all or part of the command interface offered by Intel® QST for runtime software.

Table 21: LockSubsystem Command Packet

Byte Offset	Description	Contents
00h	Command Code	05h
01h	Entity Index	0
02h-03h	Command Data Size	2
04h-05h	Response Data Size	1
06h-07h	Lock Mask	See Section 3.1.7

Table 22: LockSubsystem Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

Note: Once a Lock Mask has been applied, it will remain in place until the system makes another transition from S4/S5 to S0. At this time, the BIOS, during POST processing, may apply a new Lock Mask.

3.2.6.1 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_LOCK_SUBSYSTEM_CMD - Command structure definition                     */
/*****************************************************************************/

typedef struct _QST_LOCK_SUBSYSTEM_CMD
{
   QST_CMD_HEADER               stHeader;
   QST_LOCK_MASK                stLockMask;

} QST_LOCK_SUBSYSTEM_CMD, *P_QST_LOCK_SUBSYSTEM_CMD;
```

3.2.7 UpdateCPUConfiguration

The BIOS is required to send UpdateCPUConfiguration command(s) to Intel® QST the first time that the system is powered on and thereafter whenever a processor change is detected. One of these commands must be sent for each physical processor in the system. The command provides Intel® QST with additional information about how the processor's temperature should be monitored and fans should react to its temperature changes; without this additional information, Intel® QST will be unable to control fans based on temperature readings from the processor.

Table 23: UpdateCPUConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	06h
01h	Entity Index	0-3
02h-03h	Command Data Size	28
04h-05h	Response Data Size	1
06h	Relative Temperature	<as required>
07h	MCH Temperature Supported	<as required>
08h – 0Bh	$T_{CONTROL}$ Temperature	<as required>
0Ch – 0Fh	Accuracy Correction Offset	<as required>
10h – 13h	Accuracy Correction Slope	<as required>
14h – 17h	Proportional Gain	<as required>
18h – 1Bh	Integral Gain	<as required>
1Ch – 1Fh	Derivative Gain	<as required>
20h	Integral Time Window	<as required>
21h	Derivative Time Window	<as required>

Table 24: UpdateCPUConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.7.1 Entity Index

Specifies which processor's configuration is to be updated. Which indexes are valid is dependent upon how many processors are supported by the current firmware build. The absolute maximum is 4 (i.e. indexes 0–3); this limitation is imposed by the Platform Environment Control Interface (PECI) specification, which current defines only 4 processor addresses on the PECI Bus.

3.2.7.2 Relative Temperature

This parameter, if set (1), specifies that the processor's temperature sensor returns readings that are relative to some specific temperature target (typically the Thermal Control Circuit's (TCC) assertion point). If cleared (0), this bit indicates that the temperature sensor returns reading(s) in absolute °C.

Note: All of the processors that are supported by systems that utilize the Intel® QST 2.0 firmware provide support for the return of absolute temperature readings. As a result, this variable should typically be cleared (0).

3.2.7.3 MCH Temperature Supported

This variable, if set (1), specifies that an independent MCH thermal sensor is present in the processor. If cleared (0), this variable specifies that an independent MCH thermal sensor is not present in the CPU.

3.2.7.4 $T_{CONTROL}$ Temperature

This field specifies the processor's $T_{CONTROL}$ temperature, which represents the temperature that the processor can reliably sustain for long periods of time. The BIOS will obtain this temperature by extracting the TCC Activation Temperature and the $T_{CONTROL}$ Offset from the IA32 Temperature Target MSR, add the two values together and pass them in the UpdateCPUConfiguration command. The value is specified in the command using the special INT32F format, which presents is as a signed, 32-bit 2's-complement value that represents the temperature in 1/100ths of a degree Celsius.

3.2.7.5 Accuracy Correction Factors

These fields specify the Slope and Offset Correction Factors that should be used to correct for inaccuracies in the temperature readings that are obtained from a particular temperature sensor associated with this Processor. A fully qualified temperature correction capability requires the application of an equation of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor.

For sensors that do not require or do not have an available Slope Correction Factor, value 1.0 should be specified for the Accuracy Correction Slope. For sensors that do not require or do not have an available Offset Correction Factor, value 0.0 should be specified for the Accuracy Correction Offset.

Both of these parameters are specified as signed, 32-bit, 2's-complement values to 1/100ths unit resolution. For example, the default 1.0 slope would be specified using value 100 (64h), a slope of 1.5 would be specified using value 150 (96h), an offset of 10.0 degrees would be specified as 1000 (3E8h) and an offset of 10.5 degrees would be specified as 1050 (41Ah).

3.2.7.6 Temperature Response Coefficients

In some situations, an alternate set of Proportional-Integral-Derivative (PID) equation coefficients may need to be utilized to ensure proper response to processor temperature change. For example,

- A particular processor could generate more volatile temperature readings than can be handled by the default PID coefficients and an alternate set that offers a more-highly damped response is necessary.
- A poorer quality heatsink/fan assembly could require an alternate set of coefficients that is more responsible to temperature change.
- A particular system may support Responsiveness and/or Damping parameters in BIOS Setup. Based upon the settings of these parameters, particular sets of coefficients would be utilized.

Note: If the BIOS does not need to change the PID Coefficients, it should indicate so by setting all of the Coefficient parameters to 0.

3.2.7.6.1 ProportionalGain

This parameter is used to specify the percentage gain to be applied against the Proportional Term of the Proportional-Integral-Derivative (PID) response. It is specified as a 32-bit, two's complement value, to 1/100ths unit resolution.

3.2.7.6.2 IntegralGain

This parameter is used to specify the percentage gain to be applied against the integral term of the Proportional-Integral-Derivative (PID) response. It is specified as a 32-bit, two's complement value, to 1/100ths unit resolution.

3.2.7.6.3 DerivativeGain

This parameter is used to specify the percentage gain to be applied against the Derivative Term of the Proportional-Integral-Derivative (PID) response. It is specified as a 32-bit, two's complement value, to 1/100ths unit resolution.

3.2.7.6.4 IntegralTimeWindow

This parameter is used to specify the time window over which the Integral Term is applied. It is specified as an 8-bit unsigned value.

3.2.7.6.5 DerivativeTimeWindow

This parameter is used to specify the time window over which the Derivative Term is applied. It is specified as an 8-bit unsigned value.

3.2.7.7 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_UPDATE_CPU_CONFIG_CMD - Command structure definition                  */
/*****************************************************************************/

typedef struct _QST_UPDATE_CPU_CONFIG_CMD
{
   QST_CMD_HEADER               stHeader;
   INT8                         bRelativeTemp;
   INT8                         bMchTempSupported;
   INT32F                       lfTcontrolTemp;
   INT32F                       lfCorrectionOffset;
   INT32F                       lfCorrectionSlope;
   INT32F                       lfProportionalGain;
   INT32F                       lfIntegralGain;
   INT32F                       lfDerivativeGain;
   UINT8                        uIntegralTimeWindow;
   UINT8                        uDerivativeTimeWindow;

} QST_UPDATE_CPU_CONFIG_CMD, *P_QST_UPDATE_CPU_CONFIG_CMD;
```

3.2.8 GetCPUConfigurationUpdate

This command is used to obtain any Processor configuration updates that have been sent to Intel® QST (usually) by the BIOS.

Table 25: GetCPUConfigurationUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	07h
01h	Entity Index	0-3
02h-03h	Command Data Size	0
04h-05h	Response Data Size	30

Table 26: GetCPUConfigurationUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Update Exists
02h	Relative Temperature (See Section 3.2.7.2)
03h	MCH Temperature Supported (See section 3.2.7.3)
04h-07h	$T_{CONTROL}$ Temperature (See Section 3.2.7.4)
08h-0Bh	Accuracy Correction Offset (See Section 3.2.7.5)
0Ch-0Fh	Accuracy Correction Slope (See Section 3.2.7.5)
10h-13h	Proportional Gain (See Section 3.2.7.6.1)
14h-17h	Integral Gain (See Section 3.2.7.6.2)
18h-1Bh	Derivative Gain (See Section 3.2.7.6.3)
1Ch	Integral Time Window (See Section 3.2.7.6.4)
1Dh	Derivative Time Window (See Section 3.2.7.6.5)

3.2.8.1 Entity Index

Specifies which processor's configuration is to be retrieved. Which indexes are valid is dependent upon how many processors are supported by the current firmware build. The absolute maximum is 4 (i.e. indexes 0–3); this limitation is imposed by the Platform Environment Control Interface (PECI) specification, which current defines only 4 processor addresses on the PECI Bus.

3.2.8.2 Update Exists

This field indicates whether the requested CPU Configuration Update exists. If set (1), an update exists. If clear (0), no update exists. In this case, the contents of the remaining fields of the response are undefined.

3.2.8.3 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_CPU_CONFIG_UPDATE_RSP - Command response structure definition    */
/*****************************************************************************/

typedef struct _QST_GET_CPU_CONFIG_UPDATE_RSP
{
   UINT8                        byStatus;
   INT8                         bUpdateAvailable;
   INT8                         bRelativeTemp;
   INT8                         bMchTempSupported;
   INT32F                       lfTcontrolTemp;
   INT32F                       lfCorrectionOffset;
   INT32F                       lfCorrectionSlope;
   INT32F                       lfProportionalGain;
   INT32F                       lfIntegralGain;
   INT32F                       lfDerivativeGain;
   UINT8                        uIntegralTimeWindow;
   UINT8                        uDerivativeTimeWindow;

} QST_GET_CPU_CONFIG_UPDATE_RSP, *P_QST_GET_CPU_CONFIG_UPDATE_RSP;
```

3.2.9 UpdateCPUDTSConfiguration

This command is used by the BIOS to configure the temperature response for a processor, based upon the processor DTS-based thermal specification and the response characteristics of the heatsink-fan assembly.

Table 27: UpdateCPUDTSConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	08h
01h	Entity Index	0-3
02h-03h	Command Data Size	72
04h-05h	Response Data Size	1
06h-09h	Ψ_{CA} at Tcontrol Offset	<as required>
0Ah-0Dh	Ψ_{CA} at Tcontrol Slope	<as required>
0Eh-11h	Ψ_{CA} at -1 Temperature Offset	<as required>
12h-15h	Ψ_{CA} at -1 Temperature Slope	<as required>
16h-19h	Ψ_{CA} Intersect Point 1	<as required>
1Ah-1Bh	RPM at Ψ_{CA} Intersect Point 1	<as required>
1Ch-1Fh	Ψ_{CA} Intersect Point 2	<as required>
20h-21h	RPM at Ψ_{CA} Intersect Point 2	<as required>
22h-25h	Ψ_{CA} Intersect Point 3	<as required>
26h-27h	RPM at Ψ_{CA} Intersect Point 3	<as required>
28h-2Bh	Ψ_{CA} Intersect Point 4	<as required>
2Ch-2Dh	RPM at Ψ_{CA} Intersect Point 4	<as required>
2Eh-31h	Ψ_{CA} Intersect Point 5	<as required>
32h-33h	RPM at Ψ_{CA} Intersect Point 5	<as required>
34h-37h	Ψ_{CA} Intersect Point 6	<as required>
38h-39h	RPM at Ψ_{CA} Intersect Point 6	<as required>
3Ah-3Dh	Maximum $T_{AMBIENT}$	<as required>
3Eh	Ambient Temperature Monitor	<as required>
3Fh	Associated Fan Speed Monitor	<as required>
40h-43h	Proportional Gain	<as required>
44h-47h	Integral Gain	<as required>
48h-4Bh	Derivative Gain	<as required>
4Ch	Integral Time Window	<as required>
4Dh	Derivative Time Window	<as required>

Table 28: UpdateCPUDTSConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.9.1 Entity Index

Specifies which processor's configuration is to be updated. What indexes are valid is dependent upon how many processors are supported by the current firmware build. The absolute maximum is 4 (i.e. indexes 0–3); this limitation is imposed by the Platform Environment Control Interface (PECI) specification, which current defines only 4 processor addresses on the PECI Bus.

3.2.9.2 Ψ_{CA} at Tcontrol Slope & Offset

These parameters specify the slope (M) and offset (B) coefficients for the "Y = MX + B" equation of the Ψ_{CA} at Tcontrol line. Values are represented as 32-bit signed 2's-complement values in $1/10000^{th}$ units.

3.2.9.3 Ψ_{CA} at -1 Temperature Slope & Offset

These parameters specify the slope (M) and offset (B) coefficients for the "Y = MX + B" equation of the Ψ_{CA} at $T_{J\text{-}MAX}$ Temperature line. Values are represented as 32-bit signed 2's-complement values in $1/10000^{th}$ units.

3.2.9.4 Ψ_{CA} Intersect Point

These parameters specify the Ψ_{CA} value at a specific point on the RPM control curve. Values are represented as 32-bit signed 2's-complement values in $1/10000^{th}$ units.

Note: The points on the RPM control curve must be specified in increasing Ψ_{CA} order.

3.2.9.5 RPM at Ψ_{CA} Intersect Point

These parameters specify the RPM value that corresponds with the associated Ψ_{CA} value of a specific point on the RPM control curve.

3.2.9.6 Maximum $T_{AMBIENT}$

Specifies the maximum $T_{AMBIENT}$ temperature value supported. It is specified as a 32-bit signed 2's-complement values in $1/100^{th}$ units.

3.2.9.7 Ambient Temperature Monitor

Specifies the index of the Temperature Monitor that provides the ambient temperature readings. This is specified as an unsigned 8-bit quantity; indexes 0-31 are used to specify Temperature Monitors 1-32 respectively.

3.2.9.8 Associated Fan Speed Monitor

Specifies the index of the fan speed monitor that will provide RPM readings from the associated fan. This is specified as an unsigned 8-bit quantity.

3.2.9.9 RPM to Duty Cycle Mapping Coefficients

Specifies the coefficients for the Proportional-Integral-Derivative (PID) equation that determines the RPM to Duty Cycle Response.

3.2.9.9.1 ProportionalGain

This parameter is used to specify the percentage gain to be applied against the Proportional Term of the Proportional-Integral-Derivative (PID) response. It is specified as a 32-bit signed 2's-complement values in $1/100^{th}$ units.

3.2.9.9.2 IntegralGain

This parameter is used to specify the percentage gain to be applied against the integral term of the Proportional-Integral-Derivative (PID) response. It is specified as a 32-bit signed 2's-complement values in $1/100^{th}$ units.

3.2.9.9.3 DerivativeGain

This parameter is used to specify the percentage gain to be applied against the Derivative Term of the Proportional-Integral-Derivative (PID) response. It is specified as a 32-bit signed 2's-complement values in $1/100^{th}$ units.

3.2.9.9.4 IntegralTimeWindow

This parameter is used to specify the time window over which the Integral Term is applied. It is specified as an 8-bit unsigned value.

3.2.9.9.5 DerivativeTimeWindow

This parameter is used to specify the time window over which the Derivative Term is applied. It is specified as an 8-bit unsigned value.

3.2.9.10 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_PSICA_INTERSECT_POINT - Structure definition for coefficient pairs   */
/* on PSI-CA RPM CURVE.                                                      */
/*****************************************************************************/

#define QST_INTERSECT_POINTS    6

typedef struct _QST_PSICA_INTERSECT_POINT
{
   INT32LF                      lfPSICAPoint;
   UINT16                       wRPMValue;

} QST_PSICA_INTERSECT_POINT;

/*****************************************************************************/
/* QST_UPDATE_CPU_DTS_CONFIG_CMD - Command Structure Definition              */
/*****************************************************************************/

typedef struct _QST_UPDATE_CPU_DTS_CONFIG_CMD
{
   QST_CMD_HEADER               stHeader;
   INT32LF                      lfPSICAAtTcontrolOffset;
   INT32LF                      lfPSICAAtTcontrolSlope;
   INT32LF                      lfPSICAAtNeg1Offset;
   INT32LF                      lfPSICAAtNeg1Slope;
   QST_PSICA_INTERSECT_POINT    stRPMAtPSICAIntersect[QST_INTERSECT_POINTS];
   INT32F                       lfMaxTambient;
   UINT8                        uAmbientTempMon;
   UINT8                        uAssocFanMon;
   INT32F                       lfProportionalGain;
   INT32F                       lfIntegralGain;
   INT32F                       lfDerivativeGain;
   UINT8                        uIntegralTimeWindow;
   UINT8                        uDerivativeTimeWindow;

} QST_UPDATE_CPU_DTS_CONFIG_CMD, *P_QST_UPDATE_CPU_DTS_CONFIG_CMD;
```

3.2.10 GetCPUDTSConfigurationUpdate

This command is used to obtain DTS configuration updates for particular CPUs. These updates would have been previously sent to the QST Subsystem (presumably) by the BIOS.

Table 29: GetCPUConfigurationUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	07h
01h	Entity Index	0-3
02h-03h	Command Data Size	0
04h-05h	Response Data Size	74

Table 30: GetCPUConfigurationUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Update Exists (1 = True, 0 = False)
02h-05h	Ψ_{CA} at Tcontrol Slope (See section 3.2.9.2)
06h-09h	Ψ_{CA} at Tcontrol Offset (See section 3.2.9.2)
0Ah-0Dh	Ψ_{CA} at -1 Temperature Slope (See section 3.2.9.3)
0Eh-11h	Ψ_{CA} at -1 Temperature Offset (See section 3.2.9.3)
12h-15h	Ψ_{CA} Intersect Point 1 (See section 3.2.9.4)
16h-17h	RPM at Ψ_{CA} Intersect Point 1 (See section 3.2.9.5)
18h-1Bh	Ψ_{CA} Intersect Point 2 (See section 3.2.9.4)
1Ch-1Dh	RPM at Ψ_{CA} Intersect Point 2 (See section 3.2.9.5)
1Eh-21h	Ψ_{CA} Intersect Point 3 (See section 3.2.9.4)
22h-23h	RPM at Ψ_{CA} Intersect Point 3 (See section 3.2.9.5)
24h-27h	Ψ_{CA} Intersect Point 4 (See section 3.2.9.4)
28h-29h	RPM at Ψ_{CA} Intersect Point 4 (See section 3.2.9.5)
2Ah-2Dh	Ψ_{CA} Intersect Point 5 (See section 3.2.9.4)
2Eh-2Fh	RPM at Ψ_{CA} Intersect Point 5 (See section 3.2.9.5)
30h-33h	Ψ_{CA} Intersect Point 6 (See section 3.2.9.4)
34h-35h	RPM at Ψ_{CA} Intersect Point 6 (See section 3.2.9.5)
36h-39h	Maximum $T_{AMBIENT}$ (See section 3.2.9.6)

Byte Offset	Description
3Ah	Ambient Temperature Monitor (See section 3.2.9.7)
3Bh	Associated Fan Speed Monitor (See section 3.2.9.8)
3Ch-3Fh	Proportional Gain (See section 3.2.9.9.1)
40h-43h	Integral Gain (See section 3.2.9.9.2)
44h-47h	Derivative Gain (See section 3.2.9.9.3)
48h	Integral Time Window (See section 3.2.9.9.4)
49h	Derivative Time Window (See section 3.2.9.9.5)

3.2.10.1 Entity Index

Specifies which processor's configuration is to be retrieved. Which indexes are valid is dependent upon how many processors are supported by the current firmware build. The absolute maximum is 4 (i.e. indexes 0–3); this limitation is imposed by the Platform Environment Control Interface (PECI) specification, which current defines only 4 processor addresses on the PECI Bus.

3.2.10.2 Update Exists

This field indicates whether the requested CPU DTS Configuration Update exists. If set (1), an update exists. If clear (0), no update exists. In this case, the contents of the remaining fields of the response are undefined.

3.2.10.3 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_CPU_DTS_UPDATE_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_CPU_DTS_UPDATE_RSP
{
    UINT8                       byStatus;
    INT8                        bUpdateAvailable;
    INT32LF                     lfPSICAAtTcontrolOffset;
    INT32LF                     lfPSICAAtTcontrolSlope;
    INT32LF                     lfPSICAAtNeg1Offset;
    INT32LF                     lfPSICAAtNeg1Slope;
    QST_PSICA_INTERSECT_POINT   stRPMAtPSICAIntersect[QST_INTERSECT_POINTS];
    INT32F                      lfMaxTambient;
    UINT8                       uAmbientTempMon;
    UINT8                       uAssocFanMon;
    INT32F                      lfProportionalGain;
    INT32F                      lfIntegralGain;
    INT32F                      lfDerivativeGain;
    UINT8                       uIntegralTimeWindow;
    UINT8                       uDerivativeTimeWindow;

} QST_GET_CPU_DTS_UPDATE_RSP, *P_QST_GET_CPU_DTS_UPDATE_RSP;

// useful Macros

#define QST_INT32LF_TO_FLOAT(x)    ((float)(x) / 10000)
#define QST_INT32LF_TO_DOUBLE(x)   ((double)(x) / 10000)

#define QST_INT32LF_FROM_FLOAT(x)  ((INT32LF)((x) * 10000))
#define QST_INT32LF_FROM_DOUBLE(x) ((INT32LF)((x) * 10000))
```

3.2.11 UpdateFanControllerConfiguration

This command is sent, typically by the BIOS during POST, whenever a change is necessary in the configuration of the Fan Speed (PWM) Controller and/or connected fan(s). This is usually predicated by the insertion, removal or replacement of one or more fans controlled by a particular Fan Speed Controller.

Table 31: UpdateFanController#Configuration Command Packet

Byte Offset	Description	Contents
00h	Command Code	0Ah
01h	Entity Index	0-31
02h-03h	Command Data Size	30
04h-05h	Response Data Size	1
06h-07h	Fan Controller Configuration	<as required>
08h-09h	Associated Fan Sensor 1 Configuration	<as required>
0Ah-0Bh	Associated Fan Sensor 1 Minimum RPM Range Low	<as required>
0Ch-0Dh	Associated Fan Sensor 1 Minimum RPM Range High	<as required>
0Eh-0Fh	Associated Fan Sensor 2 Configuration	<as required>
10h-11h	Associated Fan Sensor 2 Minimum RPM Range Low	<as required>
12h-13h	Associated Fan Sensor 2 Minimum RPM Range High	<as required>
14h-15h	Associated Fan Sensor 3 Configuration	<as required>
16h-17h	Associated Fan Sensor 3 Minimum RPM Range Low	<as required>
18h-19h	Associated Fan Sensor 3 Minimum RPM Range High	<as required>
1Ah-1Bh	Associated Fan Sensor 4 Configuration	<as required>
1Ch-1Dh	Associated Fan Sensor 4 Minimum RPM Range Low	<as required>
1Eh-1Fh	Associated Fan Sensor 4 Minimum RPM Range High	<as required>
20h	Min/Off Mode	<as required>
21h	Duty Cycle Minimum	<as required>
22h	Duty Cycle On	<as required>
23h	Duty Cycle Maximum	<as required>

Table 32: UpdateFanController#Configuration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.11.1 Entity Index

This parameter specifies the index of the Fan Speed Controller whose configuration is to be updated. Indexes 0-31 are used to reference Fan Speed Controllers 1-32 respectively.

3.2.11.2 Fan Controller Configuration

This field specifies the configuration for the Fan Speed Controller:

15	14	13	12	11	10	9	8
FSCPCI			Reserved	Reserved	Reserved	Reserved	Reserved

7	6	5	4	3	2	1	0
Reserved	FSCINV	FSCSUT			FSCFRQ		

Table 33: Fan Controller Configuration Word Contents

Bit	Description
15:13	**FSCPCI – Physical Controller Index** – Specifies the index (0-7) of the fan speed controller within the set of fan speed controllers that are present in the containing device.
12:7	**Reserved.**
6	**FSCINV – Signal Invert** – When reset (0), indicates that the output signal will always be HIGH when at 100% duty cycle. When set (1), indicates that the output signal will always be LOW when at 100% duty cycle.
5:3	**FSCSUT – Spin-Up Time** – Specifies the amount of time that the PWM signal will be asserted with a 100% duty cycle, in order to overcome inertia and get the fan spinning. See Section 3.2.11.3 for information on the values for this field.
2:0	**FSCFRQ – PWM Signal Frequency** – Specifies the frequency for the PWM Control signal. See Section 3.2.11.4 for information on the values for this field.

3.2.11.3 Spin Up Time

This field specifies the amount of time that the fan must be supplied with 100% duty cycle in order to ensure that it overcomes its inertia and begins spinning.

Table 34: Fan Spin-Up Time

Value	Spin Up Time
0	0 ms
1	250 ms (default)
2	500 ms
3	750 ms
4	1000 ms
5	1500 ms
6	2000 ms
7	4000 ms

3.2.11.4 Signal Frequency

This field specifies the frequency for the PWM signal that is used to control the fan.

Table 35: PWM Signal Frequency

Value	PWM Frequency
0	~10 Hz
1	~23 Hz
2	~38 Hz
3	~62 Hz
4	~94 Hz
5	~22 kHz
6	~25 kHz (default)
7	~28 kHz

Note: The frequency of the PWM signal frequency depends upon the type of circuit that is used to control fan speed. 3-Wire Fans could require either a low-frequency (10–94 Hz) PWM signal or a high-frequency (21–28 KHz) PWM signal. 4-Wire Fans, by specification, require a high-frequency (21–28 KHz) PWM signal.

Note: The frequency of the PWM signal can have acoustic repercussions for 3-Wire fans that are controlled using the Pulsed-Power control methodology. The optimal frequency varies from fan to fan, but typically is best at the lowest setting (~10 Hz). The frequency of the PWM signal has no bearing on the acoustics of 3-wire fans controlled using a Voltage-Variance control methodology, regardless of whether it is based upon a low- or a high-frequency PWM signal. For 4-wire fans, PWM frequency has no bearing on acoustics. In this case, the 25 KHz frequency is recommended.

3.2.11.5 Associated Fan Sensor # Configuration

This field specifies the configuration for an associated fan speed sensor. As many as 4 fans may be associated to (and controlled by) a single fan controller.

15	14	13	12	11	10	9	8
AFSFSP	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

7	6	5	4	3	2	1	0
Reserved	Reserved	Reserved	Reserved	AFSDSM	AFSPPR		

Table 36: Associated Fan Sensor # Configuration Word Contents

Bit	Description
15	**AFSFSP – Fan Sensor Present** – This bit, if set (1), indicates that the fan speed sensor is present. If cleared (0), this bit indicates that no fan speed sensor is present. In this case, the remainder of the fields should be ignored.
14:4	**Reserved.**
3	**AFSDSM – Dependent Speed Measurement.** This bit, if cleared (0), indicates that fan speed measurements may be performed independent of the state of any associated Fan Speed Controller. If set (1), this bit indicates that the state of the associated Fan Speed Controller must be taken into account. This will be the case only if 3-wire fans are being controlled using the pulsed-power methodology. Many fans are incapable of returning tachometer pulses while not receiving power. Thus, extraordinary measures must be taken to accurately measure fan speed.
2:0	**AFSPPR – Pulses Per Revolution** – Specifies the number of tachometer pulses that the fan produces per revolution: 000 – 1 Pulse 001 – 2 Pulses (default) 010 – 3 Pulses 011 – 4 Pulses

Note: It is important not to confuse the contents of this fan sensor configuration word with the contents of the word that is used to configure the physical fan sensors within the chipset (despite the fact that they contain some subfields that are identical in definition).

3.2.11.6 Associated Fan # Minimum RPM Range Low

This field specifies the lower limit of the range of acceptable RPM values that the associated fan speed must fall within while operating at its Minimum Duty Cycle. If the associated fan's speed falls outside this range, the Minimum Duty Cycle will be adjusted upwards or downwards to compensate. This allows variations in individual fans to be eliminated. This field is ignored if the **AFSFSP** bit in the corresponding Associated Fan Sensor # Configuration field is reset.

3.2.11.7 Associated Fan # Minimum RPM Range High

This field specifies the upper limit of the range of acceptable RPM values that the associated fan speed must fall within while operating at its Minimum Duty Cycle. If the associated fan's speed falls outside this range, the Minimum Duty Cycle will be adjusted upwards or downwards to compensate. This allows variations in individual fans to be eliminated. This field is ignored if the **AFSFSP** bit in the corresponding Associated Fan Sensor # Configuration field is reset.

3.2.11.8 Min/Off Mode

This field specifies whether the Controller should operate in Min Mode (0) or Off Mode (1). In Min Mode, the Duty Cycle is never allowed to go below the threshold specified in the **Duty Cycle Minimum** parameter. In Off Mode, if the Duty Cycle falls below the threshold specified in the **Duty Cycle Minimum** parameter, the fan(s) will be stopped (taken to 0% Duty Cycle). Once stopped, the fan will not be turned back on

until the required Duty Cycle rises above the threshold specified in the **Duty Cycle On** parameter.

3.2.11.9 Duty Cycle Minimum

This field specifies the minimum Duty Cycle for the Controller. It is specified as an 8-bit unsigned value in the range 0-100 (percent). This threshold is chosen based upon either or both of the following factors:

1. Reliability - As the duty cycle of the control signal is reduced, a level will be reached below which the fan(s) either stop spinning (stall) or become unstable in their reaction (speed fluctuates, causing undesirable acoustic effects and/or motor noise).

2. Air Flow – Minimum airflow levels can be determined based upon the minimum thermal cooling requirements of the associated system/motherboard components.

3.2.11.10 Duty Cycle On

This field specifies the duty cycle above which the fan(s) will be restarted if they had previously been stopped. It is specified as an 8-bit unsigned value in the range 0-100 (percent). This parameter is not used if the Controller is operating in Min Mode.

Note: It is recommended that this threshold be a minimum of 2-4% higher than the Duty Cycle Minimum parameter, in order to avoid the fan oscillating on and off rapidly.

3.2.11.11 Duty Cycle Maximum

This field specifies the maximum Duty Cycle that is to be used by the Controller. It is specified as an 8-bit unsigned value in the range 0-100 (percent). Many fans produce a significantly large percentage of their overall acoustics when operating in some specific range at the top of the Duty Cycle scale. This parameter allows this range to be avoided.

Note: Care must be taken when using this feature, in order to avoid operating the fans below the levels necessary to address possible thermals.

Note: During those times when fan control is being overridden as a result of fan speed controller failure, temperature sensor failure or a critical (All-On) temperature threshold being exceeded, this parameter's value is ignored. The fans will be taken to their full speed (100% duty cycle).

3.2.11.12 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_FAN_CTRL_CONFIG - Fan Controller Configuration Word structure         */
/* definition. Note: Literals for Signal Frequency and Spin-Up Time are      */
/* defined in QstCfg.h                                                       */
/*****************************************************************************/

typedef struct _QST_FAN_CTRL_CONFIG
{
   BIT_FIELD_IN_UINT16          uPWMSignalFrequency: 3;
   BIT_FIELD_IN_UINT16          uSpinUpTime: 3;
   BIT_FIELD_IN_UINT16          bSignalInvert: 1;
   BIT_FIELD_IN_UINT16          uReserved: 9;

} QST_FAN_CTRL_CONFIG;

/*****************************************************************************/
/* QST_FAN_SENSOR_CONFIG - Fan Sensor Configuration Word structure           */
/* definition. Definitions for uPulsesPerRevolution are in QstCfg.h          */
/*****************************************************************************/

typedef struct _QST_FAN_SENSOR_CONFIG
{
   BIT_FIELD_IN_UINT16          uPulsesPerRevolution: 3;
   BIT_FIELD_IN_UINT16          bDependentMeasurement: 1;
   BIT_FIELD_IN_UINT16          uReserved: 11;
   BIT_FIELD_IN_UINT16          bSensorPresent: 1;

} QST_FAN_SENSOR_CONFIG;

/*****************************************************************************/
/* QST_FAN_SENSOR_ASSOC - Associated Fan Sensor configuration field          */
/* structure definition.                                                     */
/*****************************************************************************/

typedef struct _QST_FAN_SENSOR_ASSOC
{
   QST_FAN_SENSOR_CONFIG        stFanSensorConfig;
   UINT16                       wFanMinimumRPMRangeLow;
   UINT16                       wFanMinimumRPMRangeHigh;

} QST_FAN_SENSOR_ASSOC;

/*****************************************************************************/
/* QST_UPDATE_FAN_CONFIG_CMD - Command structure definition                  */
/*****************************************************************************/

typedef struct _QST_UPDATE_FAN_CONFIG_CMD
{
   QST_CMD_HEADER               stHeader;
   QST_FAN_CTRL_CONFIG          stFanCtrlConfig;
   QST_FAN_SENSOR_ASSOC         stFanSensorAssoc[QST_MAX_ASSOC_FAN_SENSORS];
   UINT8                        byMode;
   UINT8                        byDutyCycleMin;
   UINT8                        byDutyCycleOn;
   UINT8                        byDutyCycleMax;

} QST_UPDATE_FAN_CONFIG_CMD, *P_QST_UPDATE_FAN_CONFIG_CMD;

// MIN/OFF Mode (byMode field) Values

#define QST_MIN_MODE            0
#define QST_OFF_MODE            1
```

Further definitions for the fields of these structures are provided in header file QstCfg.h:

```
/*****************************************************************************/
/* Definitions for Fan Controller Pulses Per Revolution Fields               */
/*****************************************************************************/

#define QST_1_PULSE                0
#define QST_2_PULSES               1
#define QST_3_PULSES               2
#define QST_4_PULSES               3

/*****************************************************************************/
/* Definitions for Fan Controller Signal Frequency Field                     */
/*****************************************************************************/

#define QST_FREQ_10                0
#define QST_FREQ_23                1
#define QST_FREQ_38                2
#define QST_FREQ_62                3
#define QST_FREQ_94                4
#define QST_FREQ_22000             5
#define QST_FREQ_25000             6
#define QST_FREQ_28000             7

/*****************************************************************************/
/* Definitions for Fan Controller Spin Up Time Field                         */
/*****************************************************************************/

#define QST_SPIN_0_MS              0
#define QST_SPIN_250_MS            1
#define QST_SPIN_500_MS            2
#define QST_SPIN_750_MS            3
#define QST_SPIN_1000_MS           4
#define QST_SPIN_1500_MS           5
#define QST_SPIN_2000_MS           6
#define QST_SPIN_4000_MS           7
```

3.2.12 GetFanConfigurationUpdate

This command is used to obtain any Fan Sensor/Controller configuration updates that have been sent to Intel® QST by the BIOS.

Table 37: GetFanConfigurationUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	0Bh
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	32

Table 38: GetFanConfigurationUpdate Response Packet

Byte Offset	Description

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Update Exists (1 = True, 0 = False)
02h-03h	Fan Controller Configuration (See Section 3.2.11.2)
04h-05h	Associated Fan Sensor 1 Configuration (See Section 3.2.11.5)
06h-07h	Associated Fan Sensor 1 Minimum RPM Range Low (See Section 3.2.11.6)
08h-09h	Associated Fan Sensor 1 Minimum RPM Range High (See Section 3.2.11.7)
0Ah-0Bh	Associated Fan Sensor 2 Configuration (See Section 3.2.11.5)
0Ch-0Dh	Associated Fan Sensor 2 Minimum RPM Range Low (See Section 3.2.11.6)
0Eh-0Fh	Associated Fan Sensor 2 Minimum RPM Range High (See Section 3.2.11.7)
10h-11h	Associated Fan Sensor 3 Configuration (See Section 3.2.11.5)
12h-13h	Associated Fan Sensor 3 Minimum RPM Range Low (See Section 3.2.11.6)
14h-15h	Associated Fan Sensor 3 Minimum RPM Range High (See Section 3.2.11.7)
16h-17h	Associated Fan Sensor 4 Configuration (See Section 3.2.11.5)
18h-19h	Associated Fan Sensor 4 Minimum RPM Range Low (See Section 3.2.11.6)
1Ah-1Bh	Associated Fan Sensor 4 Minimum RPM Range High (See Section 3.2.11.7)
1Ch	Min/Off Mode (See Section 3.2.11.8)
1Dh	Duty Cycle Minimum (See Section 3.2.11.9)
1Eh	Duty Cycle On (See Section 3.2.11.10)
1Fh	Duty Cycle Maximum (See Section 3.2.11.11)

3.2.12.1 Entity Index

This parameter specifies the index of the Fan Speed Controller whose configuration is to be returned. Indexes 0-31 are used to reference Fan Speed Controllers 1-32 respectively.

3.2.12.2 Update Exists

This field indicates whether the requested Fan Controller Configuration Update exists. If set (1) the update exists. If cleared (0), no update exists. In this case, the remaining fields of the response will be undefined.

3.2.12.3 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_FAN_CONFIG_UPDATE_RSP - Command response structure definition    */
/*****************************************************************************/
```

```
typedef struct _QST_GET_FAN_CONFIG_UPDATE_RSP
{
   UINT8                      byStatus;
   INT8                       bUpdateAvailable;
   QST_FAN_CTRL_CONFIG        stFanCtrlConfig;
   QST_FAN_SENSOR_ASSOC       stFanSensorAssoc[4];
   UINT8                      byMode;
   UINT16                     byDutyCycleMin;
   UINT16                     byDutyCycleOn;
   UINT16                     byDutyCycleMax;

} QST_GET_FAN_CONFIG_UPDATE_RSP, *P_QST_GET_FAN_CONFIG_UPDATE_RSP;
```

3.2.13 SSTPassThrough

This command is used to request that Intel® QST perform the specified transaction on the SST Bus on behalf of the requestor. The SST Command Packet is said to "pass through" Intel® QST in the one direction and any response data to "pass through" Intel® QST in the other direction. A status for the operation is also returned; this would typically be an indication of the successful transmission of the command packet and reception of any response packet.

Table 39: SSTPassThrough Command Packet

Byte Offset	Description	Contents
00h	Command Code	0Ch
01h	Entity Index	0
02h-03h	Command Data Size	3 + SST Command Write Length
04h-05h	Response Data Size	1 + SST Command Read Length
05h	SST Command Device Address	<as required>
06h	SST Command Write Length	<as required>
07h	SST Command Read Length	<as required>
08h-???h	SST Command Data	<as required>

Table 40: SSTPassThrough Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h-???h	SST Command Response Data (SST Command Read Length bytes)

Note: The Temperature Sensor, Fan Speed Sensors and Fan Speed Controllers that are contained within the MCH/ICH/PCH and processor may also be accessed using SST Commands. See Chapter 5 for more information on this capability.

3.2.13.1 Related Definitions

Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* SST_CMD_HEADER - Defines header fields for SST Packets.                   */
/*****************************************************************************/

typedef struct _SST_CMD_HEADER
{
   UINT8                        bySSTAddress;
   UINT8                        byWriteLength;
   UINT8                        byReadLength;

} SST_CMD_HEADER;
```

```
/*****************************************************************************/
/* SST_CMD - Defines contents of SST Packets.                                */
/*****************************************************************************/

typedef struct _SST_CMD
{
   SST_CMD_HEADER               stSSTHeader;
   UINT8                        byCommandByte;
   UINT16                       wCommandData[1];

} SST_CMD;

/*****************************************************************************/
/* QST_SST_PASS_THROUGH_CMD - Command structure definition (with SST         */
/* Packet embedded).                                                         */
/*****************************************************************************/

typedef struct _QST_SST_PASS_THROUGH_CMD
{
    QST_CMD_HEADER              stHeader;
    SST_CMD                     stSSTPacket;

} QST_SST_PASS_THROUGH_CMD, *P_QST_SST_PASS_THROUGH_CMD;

// Useful Macros

#define QST_SST_CMD_DATA(DataWords) (4 + (2 * (DataWords)))
#define QST_SST_CMD_SIZE(DataWords) (sizeof(QST_CMD_HEADER) + \
                                     QST_SST_CMD_DATA(DataWords))

/*****************************************************************************/
/* QST_SST_PASS_THROUGH_RSP - Response Structure Definition                  */
/*****************************************************************************/

typedef struct _QST_SST_PASS_THROUGH_RSP
{
   UINT8                        byStatus;         // QST Status
   UINT16                       wValue[1];

} QST_SST_PASS_THROUGH_RSP, *P_QST_SST_PASS_THROUGH_RSP;

// Useful Macros

#define QST_SST_RSP_SIZE(NumDataWords) (1 + (2 * (NumDataWords)))
```

3.2.14 GetTemperatureMonitorUpdate

This command is used to obtain updated Health Status and Temperature readings for a specified set of Temperature Monitors.

Table 41: GetTemperatureMonitorUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	0Dh
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1 + (5 * #Monitors)

Table 42: GetTemperatureMonitorUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor n Health Status
02h-05h	Monitor n Reading
06h	Monitor n+1 Health Status
07h-0Ah	Monitor n+1 Reading
:	:

3.2.14.1 Entity Index

This parameter specifies the index of the first Temperature Monitor to be included in the update. Indexes 0-31 are used to reference Temperature Monitors 1-32 respectively.

3.2.14.2 Response Data Size

This parameter specifies how many Temperature Monitors are to be included in the update. It must be some multiple of 5 (the number of bytes occupied by a single Temperature Monitor's update) plus 1 byte for the command status.

3.2.14.3 Monitor n Health Status

Provides the health status for the n^{th} Temperature Monitor; see section 2.2.1 for details.

3.2.14.4 Monitor n Temperature Reading

Provides the current reading from the n^{th} Temperature Monitor. Temperature Readings are returned as 32-bit signed two's complement values, represented in

100ths of a degree Celsius. Where possible, temperatures will be returned in absolute form. Some temperature sensors, however, only support temperature readings that are relative to some particular set point. Whether a reading is in absolute or relative form is indicated within the Temperature Monitor configuration.

3.2.14.5 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_TEMP_MON_UPDATE - Temperature Monitor Update structure definition     */
/*****************************************************************************/

typedef struct _QST_TEMP_MON_UPDATE
{
   QST_MON_HEALTH_STATUS        stMonitorStatus;
   INT32F                       lfCurrentReading;

} QST_TEMP_MON_UPDATE;

// useful Macros

#define QST_TEMP_TO_FLOAT(x)    ((float)(x) / 100)
#define QST_TEMP_TO_DOUBLE(x)   ((double)(x) / 100)

/*****************************************************************************/
/* QST_GET_TEMP_MON_UPDATE_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_TEMP_MON_UPDATE_RSP
{
   UINT8                        byStatus;
   QST_TEMP_MON_UPDATE          stMonitorUpdate[QST_ABS_TEMP_MONITORS];

} QST_GET_TEMP_MON_UPDATE_RSP, *P_QST_GET_TEMP_MON_UPDATE_RSP;

#define QST_TEMP_MON_UPDATE_RSP_SIZE(Sensors)                                \
           ((sizeof(QST_TEMP_MON_UPDATE) * (Sensors)) + 1)

#define QST_TEMP_MON_UPDATE_RSP_SIZE_BAD(RspSize)                            \
           ((((RspSize) - 1) % sizeof(QST_TEMP_MON_UPDATE)) != 0)

#define QST_TEMP_MON_UPDATE_SENSOR_COUNT(RspSize)                            \
           (((RspSize) - 1) / sizeof(QST_TEMP_MON_UPDATE))
```

3.2.15 GetTemperatureMonitorConfiguration

This command is used to obtain the configuration for a specific temperature Monitor.

Table 43: GetTemperatureMonitorConfiguration Command

Byte Offset	Description	Contents
00h	Command Code	0Eh
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	20

Table 44: GetTemperatureMonitorConfiguration Response

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor Enabled
02h	Sensor Usage
03h	Relative Readings
04h-07h	Nominal Reading
08h-0Bh	Non-Critical Threshold
0Ch-0Fh	Critical Threshold
10h-13h	Non-Recoverable Threshold

3.2.15.1 Entity Index

This parameter specifies the index of the Temperature Monitor whose configuration information is desired. Indexes 0-31 are used to reference Temperature Monitors 1-32 respectively.

3.2.15.2 Monitor Enabled

This Boolean field indicates whether or not the Monitor is enabled. It will be set to 0 (disabled) or 1 (enabled). If the Monitor is not enabled, the content of the remaining fields is indeterminate.

3.2.15.3 Sensor Usage

This field provides an enumeration for the source of the temperature sensor. The following sources (and the enumerations assigned to them) have been defined:

Table 45: Temperature Monitor Usage

Enumeration	Source/Usage
0	Unknown/Other Temperature Source
1	Processor Core Temperature
2	Processor Die/Package Temperature
3	ICH Temperature
4	G/MCH Temperature
5	Voltage Regulator Temperature
6	Memory Temperature
7	Motherboard Ambient Temperature
8	System Ambient Air Temperature
9	Processor Inlet Air Temperature
10	System Inlet Air Temperature
11	System Outlet Air Temperature
12	Power Supply Internal/Hotspot Temperature
13	Power Supply Inlet Air Temperature
14	Power Supply Outlet Air Temperature
15	Hard Drive Temperature
16	Graphics Processor Temperature
17	IOH Temperature
18	PCH Temperature

3.2.15.4 Relative Readings

This field indicates whether or not the Monitor returns relative (as opposed to absolute) temperature readings. It will be set to 0 (false) or 1 (true).

3.2.15.5 Nominal Reading

This field provides the nominal temperature reading for this sensor. For Processor sensors, this will be the $T_{CONTROL}$ temperature for the Processor. For other sensors, this will be the typical temperature seen at this sensor when the system is idle. Temperature values, absolute or relative, are returned as 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.15.6 Non-Critical Threshold

This field provides the temperature threshold above which the Monitor will return a Non-Critical Health Status. Temperature values, absolute or relative, are returned as 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.15.7 Critical Threshold

This field provides the temperature threshold above which the Monitor will return a Critical Health Status. Temperature values, absolute or relative, are returned as 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.15.8 Non-Recoverable Threshold

This field provides the temperature threshold above which the Monitor will return a Non-Recoverable Health Status. Temperature values, absolute or relative, are returned as 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.15.9 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_TEMP_MON_CONFIG_RSP - Command response structure definition      */
/*****************************************************************************/

typedef struct _QST_GET_TEMP_MON_CONFIG_RSP
{
   UINT8                        byStatus;
   INT8                         bMonitorEnabled;
   UINT8                        byMonitorUsage;
   INT8                         bRelativeReadings;
   INT32F                       lfTempNominal;
   INT32F                       lfTempNonCritical;
   INT32F                       lfTempCritical;
   INT32F                       lfTempNonRecoverable;

} QST_GET_TEMP_MON_CONFIG_RSP, *P_QST_GET_TEMP_MON_CONFIG_RSP;
```

Definitions for the Temperature Monitor Usage field are defined in header file QstCfg.h:

```
/*****************************************************************************/
/* Definitions for Temperature Monitor Usage Field                           */
/*****************************************************************************/

#define QST_OTHER_UNKNOWN          0
#define QST_CPU_CORE_TEMP          1
#define QST_CPU_DIE_TEMP           2
#define QST_ICH_TEMP               3
#define QST_MCH_TEMP               4
#define QST_VR_TEMP                5
#define QST_MEM_TEMP               6
#define QST_MOBO_AMBIENT_TEMP      7
#define QST_SYS_AMBIENT_TEMP       8
#define QST_CPU_INLET_TEMP         9
#define QST_SYS_INLET_TEMP         10
#define QST_SYS_OUTLET_TEMP        11
#define QST_PSU_TEMP               12
#define QST_PSU_INLET_TEMP         13
#define QST_PSU_OUTLET_TEMP        14
#define QST_HARD_DRIVE_TEMP        15
#define QST_GPU_TEMP               16
#define QST_IOH_TEMP               17
#define QST_PCH_TEMP               18
```

```
#define QST_LAST_TEMP_USAGE        18
```

3.2.16 SetTemperatureMonitorHealthThresholds

This command is used to set the health monitoring thresholds for a specific Temperature Monitor.

Table 46: SetTemperatureMonitorHealthThresholds Command Packet

Byte Offset	Description	Contents
00h	Command Code	0Fh
01h	Entity Index	0-31
02h-03h	Command Data Size	12
04h-05h	Response Data Size	1
06h-09h	Non-Critical Threshold	<as required>
0Ah-0Dh	Critical Threshold	<as required>
0Eh-11h	Non-Recoverable Threshold	<as required>

Table 47: SetTemperatureMonitorHealthThresholds Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.16.1 Entity Index

This parameter specifies the index of the Temperature Monitor who's Health Thresholds are to be set. Indexes 0-31 are used to reference Temperature Monitors 1-32 respectively.

3.2.16.2 Non-Critical Threshold

This field provides the temperature threshold above which the Monitor will return a Non-Critical Health Status. Its representation must be consistent with the Reading Mode for the Monitor. That is, if the sensor returns only relative readings, thresholds must be specified in relative form also. Temperature values, absolute or relative, are specified in 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.16.3 Critical Threshold

This field provides the temperature threshold above which the Monitor will return a Critical Health Status. Its representation must be consistent with the Reading Mode for the Monitor. That is, if the sensor returns only relative readings, thresholds must be specified in relative form also. Temperature values, absolute or relative, are

specified in 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.16.4 Non-Recoverable Threshold

This field provides the temperature threshold above which the Monitor will return a Non-Recoverable Health Status. Its representation must be consistent with the Reading Mode for the sensor. That is, if the sensor returns only relative readings, thresholds must be specified in relative form also. Temperature values, absolute or relative, are specified in 32-bit signed two's-complement values, specified in hundredths of a degree Celsius.

3.2.16.5 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_TEMP_MON_THRESHOLDS_CMD - Command structure definition            */
/*****************************************************************************/

typedef struct _QST_SET_TEMP_MON_THRESHOLDS_CMD
{
   QST_CMD_HEADER             stHeader;
   INT32F                     lfTempNonCritical;
   INT32F                     lfTempCritical;
   INT32F                     lfTempNonRecoverable;

} QST_SET_TEMP_MON_THRESHOLDS_CMD, *P_QST_SET_TEMP_MON_THRESHOLDS_CMD;

// Useful Macros

#define QST_TEMP_FROM_FLOAT(x)  ((INT32F)((x) * 100))
#define QST_TEMP_FROM_DOUBLE(x) ((INT32F)((x) * 100))
```

3.2.17 SetTemperatureMonitorReading

This command is used to set the current temperature of a specific Temperature Monitor. It is only valid to do so if the Temperature Monitor is configured as a Virtual Temperature Monitors – a Temperature Monitor that does not have a Physical Temperature Sensor associated with it; the request will be rejected for normal Temperature Monitors.

Table 48: SetTemperatureMonitorReading Command Packet

Byte Offset	Description	Contents
00h	Command Code	10h
01h	Entity Index	0-31
02h-03h	Command Data Size	4
04h-05h	Response Data Size	1
06h-09h	Temperature Reading	<as required>

Table 49: SetTemperatureMonitorReading Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.17.1 Entity Index

This parameter specifies the index of the Temperature Monitor whose temperature reading will be set. Indexes 0-31 are used to reference Temperature Monitors 1-32 respectively.

3.2.17.2 Temperature Reading

Specifies the new temperature reading that is to be processed by the Temperature Monitor. Readings will be specified as signed 32-bit, two's-complement values, represented in 100ths of a degree Celsius.

3.2.17.3 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_TEMP_MON_READING_CMD - Command structure definition               */
/*****************************************************************************/

typedef struct _QST_SET_TEMP_MON_READING_CMD
{
   QST_CMD_HEADER             stHeader;
```

```
    INT32F                          lfTempReading;
} QST_SET_TEMP_MON_READING_CMD, *P_QST_SET_TEMP_MON_READING_CMD;
```

3.2.18 NoTemperatureMonitorReadings

This command is used to indicate that the software entity that was providing temperature readings to the specified Virtual Temperature Monitor is now terminating and will not be providing any subsequent readings. This command can only be used with Virtual Temperature Monitors – those Temperature Monitors that do not have a Physical Temperature Sensor associated with them; it will be rejected for normal Temperature Monitors.

Table 50: NoTemperatureMonitorReadings Command Packet

Byte Offset	Description	Contents
00h	Command Code	11h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1

Table 51: NoTemperatureMonitorReadings Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.18.1 Entity Index

This parameter specifies the index of the Temperature Monitor whose temperature reading will be set. Indexes 0-31 are used to reference Temperature Monitors 1-32 respectively.

3.2.18.2 Related Definitions

Since this command's command and response packets contain only the basic fields, the generic Command packet structure definition, QST_GENERIC_CMD, and the generic response packet structure definition, QST_GENERIC_RSP, should be used.

3.2.19 GetFanSpeedMonitorUpdate

This command is used to obtain updated Health Status and fan speed readings for a specified set of Fan Speed Monitors.

Table 52: GetFanSpeedMonitorUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	12h
01h	Entity Index	0-31
03h-04h	Command Data Size	0
03h-04h	Response Data Size	1 + (3 * #Monitors)

Table 53: GetFanSpeedMonitorUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor n Health Status
02h-03h	Monitor n Reading
04h	Monitor n+1 Health Status
05h-06h	Monitor n+1 Reading
:	:

3.2.19.1 Entity Index

This parameter specifies the index of the first Fan Speed Monitor to be included in the update. Indexes 0-31 are used to reference Fan Speed Monitors 1-32 respectively.

3.2.19.2 Response Data Size

This parameter specifies how many Fan Speed Monitors are to be included in the update. It must be some multiple of 3 (the number of bytes occupied by a single Fan Speed Monitor's update) plus 1 byte for the command status.

3.2.19.3 Monitor n Health Status

Provides the health status for the n^{th} Fan Speed Monitor; see section 2.2.1 for details. For Monitors that are disabled, the Health Status Byte will contain value 00h. This value, by (bit 0) definition, indicates that the Monitor is disabled.

3.2.19.4 Monitor n Fan Speed Reading

Provides the current reading from the n^{th} Fan Speed Monitor. Fan Speed Readings are returned as unsigned 16-bit values, representing the fan's speed in RPMs.

3.2.19.5 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_FAN_MON_UPDATE - Fan Speed Monitor Update structure definition       */
/*****************************************************************************/

typedef struct _QST_FAN_MON_UPDATE
{
   QST_MON_HEALTH_STATUS      stMonitorStatus;
   UINT16                     uCurrentSpeed;

} QST_FAN_MON_UPDATE;

/*****************************************************************************/
/* QST_GET_FAN_MON_UPDATE_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_FAN_MON_UPDATE_RSP
{
   UINT8                      byStatus;
   QST_FAN_MON_UPDATE         stMonitorUpdate[QST_MAX_FAN_MONITORS];

} QST_GET_FAN_MON_UPDATE_RSP, *P_QST_GET_FAN_MON_UPDATE_RSP;


#define QST_FAN_MON_UPDATE_RSP_SIZE(Sensors)                          \
          ((sizeof(QST_FAN_MON_UPDATE) * (Sensors)) + 1)

#define QST_FAN_MON_UPDATE_RSP_SIZE_BAD(RspSize)                      \
          ((((RspSize) - 1) % sizeof(QST_FAN_MON_UPDATE)) != 0)

#define QST_FAN_MON_UPDATE_SENSOR_COUNT(RspSize)                      \
          (((RspSize) - 1) / sizeof(QST_FAN_MON_UPDATE))
```

3.2.20 GetFanSpeedMonitorConfiguration

This command is used to obtain the configuration for a specific Fan Speed Monitor.

Table 54: GetFanSpeedMonitorConfiguration Command

Byte Offset	Description	Contents
00h	Command Code	13h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	11

Table 55: GetFanSpeedMonitorConfiguration Response

Byte Offset	Description
00h	Response Code
01h	Monitor Enabled
02h	Sensor Usage
03h-04h	Nominal Reading
05h-06h	Non-Critical Threshold
07h-08h	Critical Threshold
09h-0Ah	Non-Recoverable Threshold

3.2.20.1 Entity Index

This parameter specifies the index of the Fan Speed Monitor whose configuration is desired. Indexes 0-31 are used to reference Fan Speed Monitors 1-32 respectively.

3.2.20.2 Monitor Enabled

This field indicates whether the Fan Speed Monitor is enabled (1) or disabled (0). If it is not enabled, the content of the remaining fields is indeterminate.

3.2.20.3 Sensor Usage

This parameter specifies the source of the fan speed sensor associated with this Monitor. The following sources (and enumerations assigned to them) have been defined:

Table 56: Fan Speed Sensor/Controller Usage

Enumeration	Source/Usage
0	Unknown/Other Cooling Source
1	Processor Cooling Fan
2	Processor/System Cooling Fan
3	G/MCH Cooling Fan
4	Voltage Regulator Cooling Fan
5	Chassis Cooling Fan
6	Chassis Inlet Fan
7	Chassis Outlet Fan
8	Power Supply Cooling Fan
9	Power Supply Inlet Fan
10	Power Supply Outlet Fan
11	Hard Drive Cooling Fan
12	Graphics Processor Cooling Fan
13	Auxiliary Cooling Fan
14	IOH Cooling Fan
15	PCH Cooling Fan

3.2.20.4 Nominal Reading

This parameter provides the nominal speed, in RPMs, for the fan being monitored. Fan Speed Readings are returned as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.20.5 Non-Critical Threshold

This parameter provides the fan speed threshold below which the Monitor will return a Non-Critical Health Status. Note that this does not include those instances where the fan has been commanded to stop. Fan Speed Readings are returned as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.20.6 Critical Threshold

This parameter provides the fan speed threshold below which the Monitor will return a Critical Health Status. Note that this does not include those instances where the fan has been commanded to stop. Fan Speed Readings are returned as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.20.7 Non-Recoverable Threshold

This parameter provides the fan speed threshold below which the Monitor will return a Non-Recoverable Health Status. Note that this does not include those instances where the fan has been commanded to stop. Fan Speed Readings are returned as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.20.8 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_FAN_MON_CONFIG_RSP - Command response structure definition        */
/*****************************************************************************/

typedef struct _QST_GET_FAN_MON_CONFIG_RSP
{
   UINT8                    byStatus;
   INT8                     bMonitorEnabled;
   UINT8                    byMonitorUsage;
   UINT16                   uSpeedNominal;
   UINT16                   uSpeedNonCritical;
   UINT16                   uSpeedCritical;
   UINT16                   uSpeedNonRecoverable;

} QST_GET_FAN_MON_CONFIG_RSP, *P_QST_GET_FAN_MON_CONFIG_RSP;
```

Definitions for the Fan Speed Monitor Usage field are defined in header file QstCfg.h:

```
/*****************************************************************************/
/* Definitions for Fan Monitor/Controller Usage Field                        */
/*****************************************************************************/

#define QST_CPU_FAN               1
#define QST_CPU_SYS_FAN           2
#define QST_MCH_FAN               3
#define QST_VR_FAN                4
#define QST_CHASSIS_FAN           5
#define QST_CHASSIS_INLET_FAN     6
#define QST_CHASSIS_OUTLET_FAN    7
#define QST_PSU_FAN               8
#define QST_PSU_INLET_FAN         9
#define QST_PSU_OUTLET_FAN        10
#define QST_HARD_DRIVE_FAN        11
#define QST_GPU_FAN               12
#define QST_AUX_FAN               13
#define QST_IOH_FAN               14
#define QST_PCH_FAN               15

#define QST_LAST_FAN_USAGE        15
```

3.2.21 SetFanSpeedMonitorHealthThresholds

This command is used to set the health monitoring thresholds for a specific Fan Speed Monitor.

Table 57: SetFanSpeedMonitorHealthThresholds Command Packet

Byte Offset	Description	Contents
00h	Command Code	14h
01h	Entity Index	0-31
02h-03h	Command Data Size	6
04h-05h	Response Data Size	1
06h-07h	Non-Critical Threshold	<as required>
08h-09h	Critical Threshold	<as required>
0Ah-0Bh	Non-Recoverable Threshold	<as required>

Table 58: SetFanSpeedMonitorHealthThresholds Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.21.1 Entity Index

This parameter specifies the index of the Fan Speed Monitor whose thresholds are to be set. Indexes 0-31 are used to reference Fan Speed Monitors 1-32 respectively.

3.2.21.2 Non-Critical Threshold

This parameter specifies the fan speed threshold below which the Monitor will return a Non-Critical Health Status. Note that this does not include those instances where the fan has been commanded to stop. Fan Speed Readings are specified as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.21.3 Critical Threshold

This parameter specifies the fan speed threshold below which the Monitor will return a Critical Health Status. Note that this does not include those instances where the fan has been commanded to stop. Fan Speed Readings are specified as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.21.4 Non-Recoverable Threshold

This parameter specifies the fan speed threshold below which the Monitor will return a Non-Recoverable Health Status. Note that this does not include those instances where the fan has been commanded to stop. Fan Speed Readings are specified as unsigned 16-bit values, representing fan speeds in RPMs.

3.2.21.5 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_FAN_MON_THRESHOLDS_CMD - Command response structure definition   */
/*****************************************************************************/

typedef struct _QST_SET_FAN_MON_THRESHOLDS_CMD
{
   QST_CMD_HEADER             stHeader;
   UINT16                     uSpeedNonCritical;
   UINT16                     uSpeedCritical;
   UINT16                     uSpeedNonRecoverable;

} QST_SET_FAN_MON_THRESHOLDS_CMD, *P_QST_SET_FAN_MON_THRESHOLDS_CMD;
```

3.2.22 EnableFanSpeedMonitor

This command is typically sent by the BIOS, in order to enable a Fan Speed Monitor, (typically) when a fan is added to the system.

Table 59: EnableFanSpeedMonitor Command Packet

Byte Offset	Description	Contents
00h	Command Code	15h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1

Table 60: EnableFanSpeedMonitor Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.22.1 Entity Index

This parameter specifies the index of the Fan Speed Monitor that is to be enabled. Indexes 0-31 are used to reference Fan Speed Monitors 1-32 respectively.

3.2.22.2 Related Definitions

Since this command's command and response packets contain only the basic fields, the generic Command packet structure definition, QST_GENERIC_CMD, and response packet structure definition, QST_GENERIC_RSP, should be used.

3.2.23 DisableFanSpeedMonitor

This command is normally sent by the BIOS, in order to disable a Fan Speed Monitor, (typically) when a fan is removed from the system.

Table 61: DisableFanSpeedMonitor Command Packet

Byte Offset	Description	Contents
00h	Command Code	16h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1

Table 62: DisableFanSpeedMonitor Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.23.1 Entity Index

This parameter specifies the index of the Fan Speed Monitor that is to be disabled. Indexes 0-31 are used to reference Fan Speed Monitors 1-32 respectively.

3.2.23.2 Related Definitions

Since this command's command and response packets contain only the basic fields, the generic Command packet structure definition, QST_GENERIC_CMD, and response packet structure definition, QST_GENERIC_RSP, should be used.

3.2.24 RedetectFanPresence

This command is used by BIOS or runtime software to initiate a process that determines what fan headers actually have fans connected to them. Fan Monitors that were previously enabled that no longer appear to have fans connected to them will be disabled. Fan Monitors that were previously disabled but now have fans connected to them will be enabled.

Table 63: RedetectFanPresence Command Packet

Byte Offset	Description	Contents
00h	Command Code	17h
01h	Entity Index	0
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1

Table 64: RedetectFanPresence Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.24.1 Related Definitions

Since this command's command and response packets contain only the basic fields, the generic Command packet structure definition, QST_GENERIC_CMD, and response packet structure definition, QST_GENERIC_RSP, should be used.

3.2.25 GetVoltageMonitorUpdate

This command is used to obtain updated Health Status and Voltage readings from some number of Voltage Monitors.

Table 65: GetVoltageMonitorUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	18h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1 + (5 * #Monitors)

Table 66: GetVoltageMonitorUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor n Health Status
02h-05h	Monitor n Reading
06h	Monitor n+1 Health Status
07h-0Ah	Monitor n+1 Reading
:	:

3.2.25.1 Entity Index

This parameter specifies the index of the first Voltage Monitor to be included in the update. Indexes 0-31 are used to reference Voltage Monitors 1-32 respectively.

3.2.25.2 Response Data Size

This parameter specifies how many Voltage Monitors are to be included in the update. It must be some multiple of 5 (the number of bytes occupied by a single Voltage Monitor update) plus 1 byte for the command status.

3.2.25.3 Monitor n Health Status

Provides the health status for the n^{th} Voltage Monitor; see section 2.2.1 for details. For Monitors that are disabled, the Health Status Byte will contain value 00h. This value, by (bit 0) definition, indicates that the Monitor is disabled.

3.2.25.4 Monitor n Voltage Reading

Provides the current reading from the n^{th} Voltage Monitor. Voltage Readings are returned as signed 32-bit two's complement values, representing voltage levels in millivolts.

3.2.25.5 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_VOLT_MON_UPDATE - Voltage Monitor Update structure definition         */
/*****************************************************************************/

typedef struct _QST_VOLT_MON_UPDATE
{
   QST_MON_HEALTH_STATUS        stMonitorStatus;
   INT32                        iCurrentVoltage;

} QST_VOLT_MON_UPDATE;

// Useful Macros

#define QST_VOLT_TO_FLOAT(x)    ((float)(x) / 1000)
#define QST_VOLT_TO_DOUBLE(x)   ((double)(x) / 1000)

/*****************************************************************************/
/* QST_GET_VOLT_MON_UPDATE_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_VOLT_MON_UPDATE_RSP
{
   UINT8                        byStatus;
   QST_VOLT_MON_UPDATE          stMonitorUpdate[QST_ABS_VOLT_MONITORS];

} QST_GET_VOLT_MON_UPDATE_RSP, *P_QST_GET_VOLT_MON_UPDATE_RSP;

#define QST_VOLT_MON_UPDATE_RSP_SIZE(Sensors)                       \
           ((sizeof(QST_VOLT_MON_UPDATE) * (Sensors) ) + 1)

#define QST_VOLT_MON_UPDATE_RSP_SIZE_BAD(RspSize)                   \
           ((((RspSize) - 1) % sizeof(QST_VOLT_MON_UPDATE)) != 0)

#define QST_VOLT_MON_UPDATE_SENSOR_COUNT(RspSize)                   \
           (((RspSize) - 1) / sizeof(QST_VOLT_MON_UPDATE))
```

3.2.26 GetVoltageMonitorConfiguration

This command is used to obtain the configuration for a specific Voltage Monitor.

Table 67: GetVoltageMonitorConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	19h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	31

Table 68: GetVoltageMonitorConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor Enabled
02h	Sensor Usage
03h-06h	Nominal Reading
07h-0Ah	Non-Critical Under-Voltage Threshold
0Bh-0Eh	Non-Critical Over-Voltage Threshold
0Fh-12h	Critical Under-Voltage Threshold
13h-16h	Critical Over-Voltage Threshold
17h-1Ah	Non-Recoverable Under-Voltage Threshold
1Bh-1Eh	Non- Recoverable Over-Voltage Threshold

3.2.26.1 Entity Index

This parameter specifies the index of the Voltage Monitor whose configuration information is desired. Indexes 0-31 are used to reference Voltage Monitors 1-32 respectively.

3.2.26.2 Monitor Enabled

This field indicates whether or not the Voltage Monitor is enabled. If set (1) the Monitor is enabled. If reset (0), the Monitor is disabled; in this case, the contents of the remaining fields are indeterminate.

3.2.26.3 Sensor Usage

This parameter specifies the source of the Voltage sensor associated with this Voltage Monitor. The following sources (and enumerations assigned to them) have been defined:

Table 69: Voltage Monitor Usage

Enumeration	Source/Usage
0	Unknown/Other Voltage
1	+12 Volts
2	-12 Volts
3	+5 Volts
4	+5 Volt Backup
5	-5 Volts
6	+3.3 Volts
7	+2.5 Volts
8	+1.5 Volts
9	Processor 1 Vccp Voltage
10	Processor 2 Vccp Voltage
11	Processor 3 Vccp Voltage
12	Processor 4 Vccp Voltage
13	PSU Input Voltage
14	MCH Vcc Voltage
15	+3.3 Volt Standby
16	Processor Vtt Voltage
17	+1.8 Volts
18	PCH Vcc Voltage

3.2.26.4 Nominal Reading

This parameter provides the nominal voltage reading for this sensor. Voltage readings are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.5 Non-Critical Under-Voltage Threshold

This parameter provides the Voltage threshold below which the Monitor will return a Non-Critical Health Status. Voltage thresholds are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.6 Non-Critical Over-Voltage Threshold

This parameter provides the Voltage threshold above which the Monitor will return a Non-Critical Health Status. Voltage thresholds are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.7 Critical Under-Voltage Threshold

This parameter provides the Voltage threshold below which the Monitor will return a Critical Health Status. Voltage thresholds are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.8 Critical Over-Voltage Threshold

This parameter provides the Voltage threshold above which the Monitor will return a Critical Health Status. Voltage thresholds are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.9 Non-Recoverable Under-Voltage Threshold

This parameter provides the Voltage threshold below which the Monitor will return a Non-Recoverable Health Status. Voltage thresholds are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.10 Non-Recoverable Over-Voltage Threshold

This parameter provides the Voltage threshold above which the Monitor will return a Non-Recoverable Health Status. Voltage thresholds are provided as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.26.11 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_VOLT_MON_CONFIG_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_VOLT_MON_CONFIG_RSP
{
   UINT8                        byStatus;
   INT8                         bMonitorEnabled;
   UINT8                        byMonitorUsage;
   INT32                        iVoltageNominal;
   INT32                        iUnderVoltageNonCritical;
   INT32                        iOverVoltageNonCritical;
   INT32                        iUnderVoltageCritical;
   INT32                        iOverVoltageCritical;
   INT32                        iUnderVoltageNonRecoverable;
   INT32                        iOverVoltageNonRecoverable;

} QST_GET_VOLT_MON_CONFIG_RSP, *P_QST_GET_VOLT_MON_CONFIG_RSP;
```

Definitions for the Voltage Monitor Usage field are defined in header file QstCfg.h:

```
/*****************************************************************************/
/* Definitions for Voltage Monitor usage                                     */
/*****************************************************************************/

#define QST_12_VOLTS                1
#define QST_NEG_12_VOLTS            2
#define QST_5_VOLTS                 3
#define QST_5_VOLT_BACKUP           4
#define QST_NEG_5_VOLTS             5
#define QST_3P3_VOLTS               6
#define QST_2P5_VOLTS               7
#define QST_1P5_VOLTS               8
#define QST_CPU1_VCCP_VOLTS         9
#define QST_CPU2_VCCP_VOLTS         10
#define QST_CPU3_VCCP_VOLTS         11
#define QST_CPU4_VCCP_VOLTS         12
#define QST_PSU_INPUT_VOLTAGE       13
#define QST_MCH_VCC_VOLTS           14
#define QST_3P3_VOLT_STANDBY        15
#define QST_CPU_VTT_VOLTAGE         16
#define QST_1P8_VOLTS               17
#define QST_PCH_VCC_VOLTAGE         18

#define QST_LAST_VOLT_USAGE         18
```

3.2.27 SetVoltageMonitorHealthThresholds

This command is used to set the health monitoring thresholds for a specific Voltage Monitor.

Table 70: SetVoltageMonitorHealthThresholds Command Packet

Byte Offset	Description	Contents
00h	Command Code	1Ah
01h	Entity Index	0-31
02h-03h	Command Data Size	24
04h-05h	Response Data Size	1
06h-09h	Non-Critical Under-Voltage Threshold	<as required>
0Ah-0Dh	Non-Critical Over-Voltage Threshold	<as required>
0Eh-11h	Critical Under-Voltage Threshold	<as required>
12h-15h	Critical Over-Voltage Threshold	<as required>
16h-19h	Non-Recoverable Under-Voltage Threshold	<as required>
1Ah-1Dh	Non- Recoverable Over-Voltage Threshold	<as required>

Table 71: SetVoltageMonitorHealthThresholds Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.27.1 Entity Index

This parameter specifies the index of the Voltage Monitor whose thresholds are to be set. Indexes 0-31 are used to reference Voltage Monitors 1-32 respectively.

3.2.27.2 Non-Critical Under-Voltage Threshold

This parameter specifies the Voltage threshold below which the Monitor will return a Non-Critical Health Status. Voltage thresholds are specifies as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.27.3 Non-Critical Over-Voltage Threshold

This parameter specifies the Voltage threshold above which the Monitor will return a Non-Critical Health Status. Voltage thresholds are specifies as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.27.4 Critical Under-Voltage Threshold

This parameter specifies the Voltage threshold below which the Monitor will return a Critical Health Status. Voltage thresholds are specifies as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.27.5 Critical Over-Voltage Threshold

This parameter specifies the Voltage threshold above which the Monitor will return a Critical Health Status. Voltage thresholds are specifies as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.27.6 Non-Recoverable Under-Voltage Threshold

This parameter specifies the Voltage threshold below which the Monitor will return a Non-Recoverable Health Status. Voltage thresholds are specifies as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.27.7 Non-Recoverable Over-Voltage Threshold

This parameter specifies the Voltage threshold above which the Monitor will return a Non-Recoverable Health Status. Voltage thresholds are specifies as 32-bit signed two's-complement values, representing the voltage level in millivolts.

3.2.27.8 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_VOLT_MON_THRESHOLDS_CMD - Command structure definition            */
/*****************************************************************************/

typedef struct _QST_SET_VOLT_MON_THRESHOLDS_CMD
{
   QST_CMD_HEADER               stHeader;
   INT32                        iUnderVoltageNonCritical;
   INT32                        iOverVoltageNonCritical;
   INT32                        iUnderVoltageCritical;
   INT32                        iOverVoltageCritical;
   INT32                        iUnderVoltageNonRecoverable;
   INT32                        iOverVoltageNonRecoverable;

} QST_SET_VOLT_MON_THRESHOLDS_CMD, *P_QST_SET_VOLT_MON_THRESHOLDS_CMD;

// Useful Macros

#define QST_VOLT_FROM_FLOAT(x)  ((INT32)((x) * 1000))
#define QST_VOLT_FROM_DOUBLE(x) ((INT32)((x) * 1000))
```

3.2.28 GetCurrentMonitorUpdate

This command is used to obtain updated Health Status and current readings from some number of Current Monitors.

Table 72: GetCurrentMonitorUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	1Bh
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1 + (5 * #Monitors)

Table 73: GetCurrentMonitorUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor n Health Status
02h-05h	Monitor n Reading
06h	Monitor n+1 Health Status
07h-0Ah	Monitor n+1 Reading
:	:

3.2.28.1 Entity Index

This parameter specifies the index of the first Current Monitor to be included in the update. Indexes 0-31 are used to reference Current Monitors 1-32 respectively.

3.2.28.2 Response Data Size

This parameter specifies how many Current Monitors are to be included in the update. It must be some multiple of 5 (the number of bytes occupied by a single Current Monitor update) plus 1 byte for the command status.

3.2.28.3 Monitor n Health Status

Provides the health status for the n^{th} Current Monitor; see section 2.2.1 for details. For Monitors that are disabled, the Health Status Byte will contain value 00h. This value, by (bit 0) definition, indicates that the Monitor is disabled.

3.2.28.4 Monitor n Current Reading

Provides the current reading from the n^{th} Current Monitor. Current Readings are returned as signed 32-bit two's complement values, representing Current levels in milliamps.

3.2.28.5 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_CURR_MON_UPDATE - Current Monitor Update structure definition         */
/*****************************************************************************/

typedef struct _QST_CURR_MON_UPDATE
{
   QST_MON_HEALTH_STATUS        stMonitorStatus;
   INT32                        iCurrentCurrent;

} QST_CURR_MON_UPDATE;

// Useful Macros

#define QST_CURR_TO_FLOAT(x)    ((float)(x) / 1000)
#define QST_CURR_TO_DOUBLE(x)   ((double)(x) / 1000)

/*****************************************************************************/
/* QST_GET_CURR_MON_UPDATE_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_CURR_MON_UPDATE_RSP
{
   UINT8                        byStatus;
   QST_CURR_MON_UPDATE          stMonitorUpdate[QST_ABS_CURR_MONITORS];

} QST_GET_CURR_MON_UPDATE_RSP, *P_QST_GET_CURR_MON_UPDATE_RSP;

#define QST_CURR_MON_UPDATE_RSP_SIZE(Sensors)                       \
           ((sizeof(QST_CURR_MON_UPDATE) * (Sensors)) + 1)

#define QST_CURR_MON_UPDATE_RSP_SIZE_BAD(RspSize)                   \
           ((((RspSize) - 1) % sizeof(QST_CURR_MON_UPDATE)) != 0)

#define QST_CURR_MON_UPDATE_SENSOR_COUNT(RspSize)                   \
           (((RspSize) - 1) / sizeof(QST_CURR_MON_UPDATE))
```

3.2.29 GetCurrentMonitorConfiguration

This command is used to obtain the configuration for a specific Current Monitor.

Table 74: GetCurrentMonitorConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	1Ch
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	31

Table 75: GetCurrentMonitorConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Monitor Enabled
02h	Sensor Usage
03h-06h	Nominal Reading
07h-0Ah	Non-Critical Under-Current Threshold
0Bh-0Eh	Non-Critical Over-Current Threshold
0Fh-12h	Critical Under-Current Threshold
13h-16h	Critical Over-Current Threshold
17h-1Ah	Non-Recoverable Under-Current Threshold
1Bh-1Eh	Non- Recoverable Over-Current Threshold

3.2.29.1 Entity Index

This parameter specifies the index of the Current Monitor whose configuration information is desired. Indexes 0-31 are used to reference Current Monitors 1-32 respectively.

3.2.29.2 Monitor Enabled

This field indicates whether or not the Current Monitor is enabled. If set (1) the Monitor is enabled. If reset (0), the Monitor is disabled; in this case, the contents of the remaining fields are indeterminate.

3.2.29.3 Sensor Usage

This parameter specifies the source of the Current sensor associated with this Current Monitor. The following sources (and enumerations assigned to them) have been defined:

Table 76: Current Monitor Usage

Enumeration	Source/Usage
0	Unknown/Other Current
1	+12V Current
2	-12V Current
3	+5V Current
4	+5V Backup Current
5	-5V Current
6	+3.3V Current
7	+2.5V Current
8	+1.5V Current
9	Processor 1 Current
10	Processor 2 Current
11	Processor 3 Current
12	Processor 4 Current
13	PSU Input Current
14	MCH Current
15	+3.3V Standby Current
16	+1.8V Current
17	PCH Current

3.2.29.4 Nominal Reading

This parameter provides the nominal Current reading for this sensor. Current readings are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.5 Non-Critical Under-Current Threshold

This parameter provides the Current threshold below which the Monitor will return a Non-Critical Health Status. Current thresholds are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.6 Non-Critical Over-Current Threshold

This parameter provides the Current threshold above which the Monitor will return a Non-Critical Health Status. Current thresholds are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.7 Critical Under-Current Threshold

This parameter provides the Current threshold below which the Monitor will return a Critical Health Status. Current thresholds are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.8 Critical Over-Current Threshold

This parameter provides the Current threshold above which the Monitor will return a Critical Health Status. Current thresholds are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.9 Non-Recoverable Under-Current Threshold

This parameter provides the Current threshold below which the Monitor will return a Non-Recoverable Health Status. Current thresholds are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.10 Non-Recoverable Over-Current Threshold

This parameter provides the Current threshold above which the Monitor will return a Non-Recoverable Health Status. Current thresholds are provided as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.29.11 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_CURR_MON_CONFIG_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_CURR_MON_CONFIG_RSP
{
   UINT8                        byStatus;
   INT8                         bMonitorEnabled;
   UINT8                        byMonitorUsage;
   INT32                        iCurrentNominal;
   INT32                        iUnderCurrentNonCritical;
   INT32                        iOverCurrentNonCritical;
   INT32                        iUnderCurrentCritical;
   INT32                        iOverCurrentCritical;
   INT32                        iUnderCurrentNonRecoverable;
   INT32                        iOverCurrentNonRecoverable;

} QST_GET_CURR_MON_CONFIG_RSP, *P_QST_GET_CURR_MON_CONFIG_RSP;
```

Definitions for the Current Monitor Usage field are defined in header file QstCfg.h:

```
/*****************************************************************************/
/* Definitions for Current Monitor usage                                     */
/*****************************************************************************/

#define QST_12V_CURRENT             1
#define QST_NEG_12V_CURRENT         2
#define QST_5V_CURRENT              3
#define QST_5V_BACKUP_CURRENT       4
#define QST_NEG_5V_CURRENT          5
#define QST_3P3V_CURRENT            6
#define QST_2P5V_CURRENT            7
#define QST_1P5V_CURRENT            8
#define QST_CPU1_CURRENT            9
#define QST_CPU2_CURRENT            10
#define QST_CPU3_CURRENT            11
#define QST_CPU4_CURRENT            12
#define QST_PSU_INPUT_CURRENT       13
#define QST_MCH_CURRENT             14
#define QST_3P3V_STANDBY_CURRENT    15
#define QST_1P8V_CURRENT            16
#define QST_PCH_CURRENT             17

#define QST_LAST_CURR_USAGE         17
```

3.2.30 SetCurrentMonitorHealthThresholds

This command is used to set the health monitoring thresholds for a specific Current Monitor.

Table 77: SetCurrentMonitorHealthThresholds Command Packet

Byte Offset	Description	Contents
00h	Command Code	1Dh
01h	Entity Index	0-31
02h-03h	Command Data Size	24
04h-05h	Response Data Size	1
06h-09h	Non-Critical Under-Current Threshold	<as required>
0Ah-0Dh	Non-Critical Over-Current Threshold	<as required>
0Eh-11h	Critical Under-Current Threshold	<as required>
12h-15h	Critical Over-Current Threshold	<as required>
16h-19h	Non-Recoverable Under-Current Threshold	<as required>
1Ah-1Dh	Non- Recoverable Over-Current Threshold	<as required>

Table 78: SetCurrentMonitorHealthThresholds Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.30.1 Entity Index

This parameter specifies the index of the Current Monitor whose thresholds are to be set. Indexes 0-31 are used to reference Current Monitors 1-32 respectively.

3.2.30.2 Non-Critical Under-Current Threshold

This parameter specifies the Current threshold below which the Monitor will return a Non-Critical Health Status. Current thresholds are specifies as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.30.3 Non-Critical Over-Current Threshold

This parameter specifies the Current threshold above which the Monitor will return a Non-Critical Health Status. Current thresholds are specifies as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.30.4 Critical Under-Current Threshold

This parameter specifies the Current threshold below which the Monitor will return a Critical Health Status. Current thresholds are specifies as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.30.5 Critical Over-Current Threshold

This parameter specifies the Current threshold above which the Monitor will return a Critical Health Status. Current thresholds are specifies as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.30.6 Non-Recoverable Under-Current Threshold

This parameter specifies the Current threshold below which the Monitor will return a Non-Recoverable Health Status. Current thresholds are specifies as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.30.7 Non-Recoverable Over-Current Threshold

This parameter specifies the Current threshold above which the Monitor will return a Non-Recoverable Health Status. Current thresholds are specifies as 32-bit signed two's-complement values, representing the Current level in milliamps.

3.2.30.8 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_CURR_MON_THRESHOLDS_CMD - Command structure definition            */
/*****************************************************************************/

typedef struct _QST_SET_CURR_MON_THRESHOLDS_CMD
{
   QST_CMD_HEADER               stHeader;
   INT32                        iUnderCurrentNonCritical;
   INT32                        iOverCurrentNonCritical;
   INT32                        iUnderCurrentCritical;
   INT32                        iOverCurrentCritical;
   INT32                        iUnderCurrentNonRecoverable;
   INT32                        iOverCurrentNonRecoverable;

} QST_SET_CURR_MON_THRESHOLDS_CMD, *P_QST_SET_CURR_MON_THRESHOLDS_CMD;

// Useful Macros

#define QST_CURR_FROM_FLOAT(x)  ((INT32)((x) * 1000))
#define QST_CURR_FROM_DOUBLE(x) ((INT32)((x) * 1000))
```

3.2.31 GetFanControllerUpdate

This command is used to get the status for one or more of the Fan Controllers.

Table 79: GetFanControllerUpdate Command Packet

Byte Offset	Description	Contents
00h	Command Code	1Eh
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1 + (3 * #Controllers)

Table 80: GetFanControllerUpdate Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Controller n Health Status
02h-03h	Controller n Duty Cycle
04h	Controller n+1 Health Status
05h-06h	Controller n+1 Duty Cycle
:	:

3.2.31.1 Entity Index

This parameter specifies the index of the first Fan Controller to be included in the update. Indexes 0-31 are used to reference Fan Controllers 1-32 respectively.

3.2.31.2 Response Data Size

This parameter specifies how many Fan Controllers are to be included in the update. It must be some multiple of 3 (the number of bytes occupied by a single Fan Controller update) plus 1 byte for the command status.

3.2.31.3 Controller n Health Status

See Section 2.3.3 for information on the contents of the Health Status Byte. For Fan Controllers that are disabled, the Health Status Byte will contain value 00h. This value, by (bit 0) definition, indicates that the Fan Controller is disabled.

3.2.31.4 Controller n Duty Cycle

This field provides the Duty Cycle value that is currently being used to control the speed of the associated fan(s). Duty Cycle values are returned as unsigned, 16-bit quantities, with values 0 through 10000 used to indicate Duty Cycles 0% through 100%. For disabled Controllers, the content of this field is undefined.

3.2.31.5 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_FAN_CTRL_STATUS - Fan Controller Status structure definition          */
/*****************************************************************************/

typedef struct _QST_FAN_CTRL_STATUS
{
   BIT_FIELD_IN_UINT8           bControllerEnabled: 1;
   BIT_FIELD_IN_UINT8           uControllerStatus: 2;
   BIT_FIELD_IN_UINT8           bOverrideSoftware: 1;
   BIT_FIELD_IN_UINT8           bOverrideFanController: 1;
   BIT_FIELD_IN_UINT8           bOverrideTemperatureSensor: 1;
   BIT_FIELD_IN_UINT8           bOverrideFanStall: 1;
   BIT_FIELD_IN_UINT8           bOverrideRedetection: 1;

} QST_FAN_CTRL_STATUS;

/*****************************************************************************/
/* QST_FAN_CTRL_UPDATE - Fan Controller Update structure definition          */
/*****************************************************************************/

typedef struct _QST_FAN_CTRL_UPDATE
{
   QST_FAN_CTRL_STATUS          stControllerStatus;
   UINT16                       uCurrentDutyCycle;

} QST_FAN_CTRL_UPDATE;

// Useful Macros

#define QST_DUTY_TO_FLOAT(x)    ((float)(x) / 100)
#define QST_DUTY_TO_DOUBLE(x)   ((double)(x) / 100)
```

3.2.32 GetFanControllerConfiguration

This command is used to obtain the configuration for a specific Fan Speed Controller.

Table 81: GetFanControllerConfiguration Command Packet

Byte Offset	Description	Contents
00h	Command Code	1Fh
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	3

Table 82: GetFanControllerConfiguration Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)
01h	Controller Enabled (Boolean; 0=Disabled, 1=Enabled)
02h	Controller Usage

3.2.32.1 Entity Index

This parameter specifies the index of the Fan Controller whose configuration is desired. Indexes 0-31 are used to reference Fan Controllers 1-32 respectively.

3.2.32.2 Controller Enabled

This field indicates whether or not the Fan Controller is enabled. If it is not enabled, the contents of the remaining fields are indeterminate.

3.2.32.3 Controller Usage

This parameter specifies the target of the Fan Speed Controller. The following targets (and enumerations assigned to them) have been defined:

Table 83: Fan Speed Controller Usage

Enumeration	Target/Usage
0	Unknown/Other Cooling Source
1	Processor Cooling Fan
2	Processor/System Cooling Fan
3	G/MCH Cooling Fan
4	Voltage Regulator Cooling Fan
5	Chassis Cooling Fan
6	Chassis Inlet Fan
7	Chassis Outlet Fan
8	Power Supply Cooling Fan
9	Power Supply Inlet Fan
10	Power Supply Outlet Fan
11	Hard Drive Cooling Fan
12	Graphics Processor Cooling Fan
13	Auxiliary Cooling Fan

3.2.32.4 Related Definitions

Since this command's command packet contains only the basic fields, the generic command packet structure definition, QST_GENERIC_CMD, should be used. Definitions for this command's response packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_GET_FAN_CTRL_CONFIG_RSP - Command response structure definition       */
/*****************************************************************************/

typedef struct _QST_GET_FAN_CTRL_CONFIG_RSP
{
   UINT8                        byStatus;
   INT8                         bControllerEnabled;
   UINT8                        byControllerUsage;

} QST_GET_FAN_CTRL_CONFIG_RSP, *P_QST_GET_FAN_CTRL_CONFIG_RSP;
```

Definitions for the Fan Speed Controller Usage field are defined in header file QstCfg.h:

```
/*****************************************************************************/
/* Definitions for Fan Monitor/Controller Usage Field                        */
/*****************************************************************************/

#define QST_CPU_FAN                 1
#define QST_CPU_SYS_FAN             2
#define QST_MCH_FAN                 3
#define QST_VR_FAN                  4
#define QST_CHASSIS_FAN             5
#define QST_CHASSIS_INLET_FAN       6
#define QST_CHASSIS_OUTLET_FAN      7
#define QST_PSU_FAN                 8
#define QST_PSU_INLET_FAN           9
#define QST_PSU_OUTLET_FAN          10
#define QST_HARD_DRIVE_FAN          11
#define QST_GPU_FAN                 12
#define QST_AUX_FAN                 13
#define QST_IOH_FAN                 14
#define QST_PCH_FAN                 15

#define QST_LAST_FAN_USAGE          15
```

3.2.33 SetFanControllerDuty

This command is used to specify the current duty cycle (percentage) for the associated Fan Controllers.

Note: When one of these commands is delivered (and accepted; see security discussion in Section 3.1.7), the Fan Controller will be placed in manual mode. From this point on, it will be up to software to specify the fan speed.

Note: This capability is present to support diagnostics only. If used incorrectly, it could result in damaging over-temperature conditions. Its use in production environments is thus strongly discouraged.

Table 84: SetFanControllerDuty Command Packet

Byte Offset	Description	Contents
00h	Command Code	20h
01h	Entity Index	0-31
02h-03h	Command Data Size	2
04h-05h	Response Data Size	1
06h-07h	Fan Controller Duty Cycle	<as required>

Table 85: SetFanControllerDuty Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.33.1 Entity Index

This parameter specifies the index of the Fan Controller whose Duty Cycle is to be set. Indexes 0-31 are used to reference Fan Controllers 1-32 respectively.

3.2.33.2 Controller Duty Cycle

This parameter specifies the Duty Cycle that the Fan Controller is to be set to. Duty Cycle values are specified as unsigned 16-bit quantities with values 0 through 10000 used to indicate Duty Cycles 0 through 100%.

3.2.33.3 Related Definitions

Since this command's response packet contains only the basic fields, the generic response packet structure definition, QST_GENERIC_RSP, should be used. Definitions for this command's command packet are provided in header file QstCmd.h, as follows:

```
/*****************************************************************************/
/* QST_SET_FAN_CTRL_DUTY_CMD - Command structure definition                  */
/*****************************************************************************/

typedef struct _QST_SET_FAN_CTRL_DUTY_CMD
{
   QST_CMD_HEADER               stHeader;
   UINT16                       uDutyCycle;

} QST_SET_FAN_CTRL_DUTY_CMD, *P_QST_SET_FAN_CTRL_DUTY_CMD;

// Useful Macros

#define QST_DUTY_FROM_FLOAT(x)  ((UINT16)((x) * 100))
#define QST_DUTY_FROM_DOUBLE(x) ((UINT16)((x) * 100))
```

3.2.34 SetFanControllerAuto

This command is used to place a particular Fan Controller (back) into automatic control mode. In automatic control mode, Fan Controllers will determine the optimal duty cycle values based upon temperature response decisions and weightings.

Table 86: SetFanController#Auto Command Packet

Byte Offset	Description	Contents
00h	Command Code	21h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1

Table 87: SetFanController#Auto Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.34.1 Entity Index

This parameter specifies the index of the Fan Controller to be put (back) into automatic control mode. Indexes 0-31 are used to reference Fan Controllers 1-32 respectively.

3.2.34.2 Related Definitions

Since this command's command and response packets contain only the basic fields, the generic Command packet structure definition, QST_GENERIC_CMD, and response packet structure definition, QST_GENERIC_RSP, should be used.

3.2.35 ResetFanControllerMinimumDuty

This command is used to reset the minimum duty cycle of particular Fan Controllers. It is typically sent by software or BIOS as part of the process of replacing a fan. The minimum duty cycle in use for the (previous) fan may have been incremented automatically by the Minimum RPM feature (see the Configuration and Tuning Manual for more information on this feature).

Table 88: ResetFanControllerMinimumDuty Command Packet

Byte Offset	Description	Contents
00h	Command Code	22h
01h	Entity Index	0-31
02h-03h	Command Data Size	0
04h-05h	Response Data Size	1

Table 89: ResetFanController#MinimumDuty Response Packet

Byte Offset	Description
00h	Response Code (see Section 3.1.6.1)

3.2.35.1 Entity Index

This parameter specifies the index of the Fan Controller whose Minimum Duty Cycle is to be set. Indexes 0-31 are used to reference Fan Controllers 1-32 respectively.

3.2.35.2 Related Definitions

Since this command's command and response packets contain only the basic fields, the generic Command packet structure definition, QST_GENERIC_CMD, and response packet structure definition, QST_GENERIC_RSP, should be used.

3.2.36 Command Summary

The following table summarizes the commands used to control and query the operation of the Intel® QST Subsystem:

Table 90: Intel® QST Query/Control Commands

Reference	Command	Code(s)	Locked By (See Section 3.1.7)
3.2.1	GetSubsystemInformation	00h	--
3.2.2	GetSubsystemStatus	01h	--
3.2.3	GetSubsystemConfiguration	02h	--
3.2.4	GetSubsystemConfigurationProfile	03h	--
3.2.5	SetSubsystemConfiguration	04h	LCKCFG
3.2.6	LockSubsystem	05h	LCKCFG
3.2.7	UpdateCPUConfiguration	06h	LCKCFG
3.2.8	GetCPUConfigurationUpdate	07h	--
3.2.9	UpdateCPUDTSConfiguration	08h	LCKCFG
3.2.10	GetCPUDTSConfigurationUpdate	09h	--
3.2.11	UpdateFanControllerConfiguration	0Ah	LCKCFG
3.2.12	GetFanControllerConfigurationUpdate	0Bh	--
3.2.13	SSTPassThrough	0Ch	LCKCHP/LCKSST
3.2.14	GetTemperatureMonitorUpdate	0Dh	--
3.2.15	GetTemperatureMonitorConfiguration	0Eh	--
3.2.16	SetTemperatureMonitorHealthThresholds	0Fh	LCKTHR
3.2.17	SetTemperatureMonitorReading	10h	Only accepted for Virtual Monitors
3.2.18	NoTemperatureMonitorReadings	11h	Only accepted for Virtual Monitors
3.2.19	GetFanSpeedMonitorUpdate	12h	--
0	GetFanSpeedMonitorConfiguration	13h	--
3.2.21	SetFanSpeedMonitorHealthThresholds	14h	LCKTHR
0	EnableFanSpeedMonitor	15h	LCKCFG
0	DisableFanSpeedMonitor	16h	LCKCFG
3.2.24	RedetectFanPresence	17h	LCKCFG
3.2.25	GetVoltageMonitorUpdate	18h	--
3.2.26	GetVoltageMonitorConfiguration	19h	--
3.2.27	SetVoltageMonitorHealthThresholds	1Ah	LCKTHR

Reference	Command	Code(s)	Locked By (See Section 3.1.7)
3.2.28	GetCurrentMonitorUpdate	1Bh	--
3.2.29	GetCurrentMonitorConfiguration	1Ch	--
3.2.30	SetCurrentMonitorHealthThresholds	1Dh	LCKTHR
3.2.31	GetFanControllerUpdate	1Eh	--
3.2.32	GetFanControllerConfiguration	1Fh	--
3.2.33	SetFanControllerDuty	20h	LCKMFC
3.2.34	SetFanControllerAuto	21h	LCKMFC
3.2.35	ResetFanControllerMinimumDuty	22h	LCKCFG

3.3 Using Intel® QST Commands

3.3.1 Determining Intel® QST Configuration Status

The following code details how to determine whether or not Intel® QST is configured. It sends a GetSubsystemStatus message and uses the response to determine whether the Subsystem is configured. It also checks the Lock Mask to determine whether it will be possible to perform manual fan speed control functions. QstCtrl, the manual fan speed control tool, uses code similar to this to determine what operations it may successfully perform.

```
static BOOL bFanCtrlUpdatable = FALSE;

/*****************************************************************************/
/* QSTConfigured() - Indicates whether QST Subsystem is configured. Also     */
/* sets global variable bFanCtrlUpdateable to indicate whether Subsystem     */
/* will accept Manual Fan Control commands.                                  */
/*****************************************************************************/

static BOOL QSTConfigured( void )
{
   QST_GENERIC_CMD                stCmd;
   QST_GET_SUBSYSTEM_STATUS_RSP   stRsp;

   // Get the QST Subsystem's Status

   stCmd.stHeader.byCommand       = QST_GET_SUBSYSTEM_STATUS;
   stCmd.stHeader.byEntity        = 0;
   stCmd.stHeader.wCommandLength  = QST_CMD_DATA_SIZE(QST_GENERIC_CMD);
   stCmd.stHeader.wResponseLength = sizeof(QST_GET_SUBSYSTEM_STATUS_RSP);

   if( !QstCommand2( &stCmd, sizeof(QST_GENERIC_CMD),
                     &stRsp, sizeof(QST_GET_SUBSYSTEM_STATUS_RSP) ) )
      return( FALSE );

   if( stRsp.byStatus )
   {
      SetLastErrorQST( stRsp.byStatus );
      return( FALSE );
   }

   // No point in going any further if the subsystem isn't configured

   if( !stRsp.stSubsystemStatus.bSubsystemConfigured )
   {
      SetLastError( ERROR_QST_NOT_CONFIGURED );
      return( FALSE );
   }

   // Check the Lock Mask to see whether we can support fan control operations

   if( !stRsp.stLockMask.bLockManualFanControl )
      bFanCtrlUpdatable = TRUE;

   // Let caller know Subsystem is configured…

   return( TRUE );
}
```

3.3.2 Enumerating Sensors/Controllers

The following code demonstrates how one can enumerate the configured Temperature Monitors. This process can be easily extended to enumerate Fan Speed and Voltage Monitors, as well as Fan Speed Controllers. The routine requests a profile of the current configuration and then uses the appropriate fields of the response to determine which Temperature Monitors are configured. Global variable iTempMons will be set to indicate how many Temperature Monitors are configured. As well, global array iTempMonIndex will be set up to provide logical-to-physical mappings, such that discontiguously-configured Temperature Monitors may be indexed contiguously. Note that this function also invokes function GetTempMonConfig(), discussed in the next section, in order to obtain the configuration data for each Monitor enumerated.

```
    static int  iTempMons = 0;
    static int  iTempMonIndex[QST_MAX_TEMP_MONITORS];

    /*****************************************************************************/
    /* GetTempMonEnum() - Enumerates the configured Temperature Monitors.        */
    /*****************************************************************************/

    static BOOL GetTempMonEnum( void )
    {
       int                                  iBit;
       QST_GENERIC_CMD                      stCmd;
       QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP  stRsp;

       // Get the QST Subsystem's configuration profile

       stCmd.stHeader.byCommand       = QST_GET_SUBSYSTEM_CONFIG_PROFILE;
       stCmd.stHeader.byEntity        = 0;
       stCmd.stHeader.wCommandLength  = QST_CMD_DATA_SIZE(QST_GENERIC_CMD);
       stCmd.stHeader.wResponseLength =
sizeof(QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP);

       if( !QstCommand2( &stCmd, sizeof(QST_GENERIC_CMD),
                         &stRsp, sizeof(QST_GET_SUBSYSTEM_CONFIG_PROFILE_RSP) ) )
          return( FALSE );

       if( stRsp.byStatus )
       {
          SetLastErrorQST( stRsp.byStatus );
          return( FALSE );
       }

       // Ascertain Temperature Monitor count and configuration

       for( iBit = iTempMons = 0; iBit < QST_ABS_TEMP_MONITORS; iBit++ )
       {
          if( BIT_SET( stProfileRsp.wTempMonsConfigured, iBit ) )
          {
             if( !GetTempMonConfig( iBit, iTempMons ) )
                return( FALSE );

             iTempMonIndex[iTempMons++] = iBit;
          }
       }

       // Code to enumerate the other sensors/controllers would go here…

       // Inform caller we were successful

       return( TRUE );
    }
```

3.3.3 Obtaining Sensor/Controller Configuration

The following code demonstrates how one can obtain the configuration data for a specific Temperature Monitor. A similar process can be used to obtain configuration data for configured Fan Speed and Voltage Monitors, as well as for Fan Speed Controllers.

The routine sends a GetTemperatureMonitorConfiguration command to the Intel® QST, in order to obtain the desired information. This information is saved in global array stTempMonCfg, so that it may be used for subsequent operations.

```
static QST_GET_TEMP_MON_CONFIG_RSP stTempMonCfg[QST_ABS_TEMP_MONITORS];

/*****************************************************************************/
/* GetTempMonConfig() - Obtains configuration for a Temperature Monitor      */
/*****************************************************************************/

static BOOL GetTempMonConfig( iRemoteSensor, iLocalSensor )
{
   QST_GENERIC_CMD stCmd;

   // Send the Temperature Monitor Configuration request

   stCmd.stHeader.byCommand       = QST_GET_TEMP_MON_CONFIG;
   stCmd.stHeader.byEntity        = iRemoteSensor;
   stCmd.stHeader.wCommandLength  = QST_CMD_DATA_SIZE(QST_GENERIC_CMD);
   stCmd.stHeader.wResponseLength = sizeof(QST_GET_TEMP_MON_CONFIG_RSP);

   if( !QstCommand2( &stCmd,
                        sizeof(QST_GENERIC_CMD),
                     &stTempMonCfg[iLocalSensor],
                        sizeof(QST_GET_TEMP_MON_CONFIG_RSP) ) )
      return( FALSE );

   // Can't go any further if Subsystem rejected the command

   if( stTempMonCfg[iLocalSensor].byStatus )
   {
      SetLastErrorQST( stTempMonCfg[iLocalSensor].byStatus );
      return( FALSE );
   }

   // Inform caller we were successful

   return( TRUE );
}
```

3.3.4 Obtaining Sensor/Controller Updates

The following code demonstrates how one can obtain updated health status and sensor readings for the configured Temperature Monitors. A similar process can be used to obtain updates for configured Fan Speed and Voltage Monitors, as well as for Fan Speed Controllers.

The routine sends a GetTemperatureMonitorUpdate command to the Intel® QST, requesting health status and sensor readings for all Temperature Monitors. Since the QST_GET_TEMP_MON_UPDATE_RSP structure includes entries for all possible Temperature Monitors, we can simply use the size of the structure for the Response Length.

In this example, Standard C Library function time() is used to determine whether or not fresh data is needed (fresh data is desired on a per-second basis).

```
static QST_GET_TEMP_MON_UPDATE_RSP  stTempMonUpd;
static time_t                       tLastTempMonUpdate = 0;

/*****************************************************************************/
/* GetTempMonUpdate() - Gets updated health status and readings from all     */
/* Temperature Monitors, so long as a second has passed since the last       */
/* invocation.                                                               */
/*****************************************************************************/

static BOOL GetTempMonUpdate( void )
{
   // Only update if readings are at least a second old

   time_t tCurr = time( NULL );

   if( tLastTempMonUpdate < tCurr )
   {
      QST_GENERIC_CMD stCmd;

      // Send the Temperature Monitor Update request

      stCmd.stHeader.byCommand       = QST_GET_TEMP_MON_UPDATE;
      stCmd.stHeader.byEntity        = 0;
      stCmd.stHeader.wCommandLength  = QST_CMD_DATA_SIZE(QST_GENERIC_CMD);
      stCmd.stHeader.wResponseLength = sizeof(QST_GET_TEMP_MON_UPDATE_RSP);

      if( !QstCommand2( &stCmd,
                           sizeof(QST_GENERIC_CMD),
                        &stTempMonUpd,
                           sizeof(QST_GET_TEMP_MON_UPDATE_RSP) ) )
         return( FALSE );

      // Can't go any further if Subsystem rejected the command

      if( stTempMonUpd.byStatus )
      {
         SetLastErrorQST( stTempMonUpd.byStatus );
         return( FALSE );
      }

      // Save the time of the update for later comparison

      tLastTempMonUpdate = tCurr;
   }

   return( TRUE );
}
```

3.3.5 Exposing Sensor/Controller Information

The following code demonstrates how the aforementioned support for obtaining configuration and health/reading information can be used to expose the necessary information to other software layers. Support for Temperature Monitors is shown; similar code can be crafted to support other Monitor types, as well as Fan Controllers.

```
/*****************************************************************************/
/* GetTempCountQst() - Returns number of enabled Temperature Monitors        */
/*****************************************************************************/

int GetTempCountQst( void )
{
   return( iTempMons );
}

/*****************************************************************************/
/* GetTempIndexQst() - Returns physical index for Temperature Monitor        */
/*****************************************************************************/

int GetTempIndexQst( int iLocalSensor )
{
   if( (iLocalSensor < 0) || (iLocalSensor >= iTempMons) )
   {
      SetLastError( ERROR_QST_INVALID_PARAMETER );
      return( -1 );
   }

   return( iTempMonIndex[iLocalSensor] );
}

/*****************************************************************************/
/* GetTempUsageQst() - Returns usage indicator for Temperature Monitor       */
/*****************************************************************************/

int GetTempUsageQst( int iLocalSensor )
{
   if( (iLocalSensor < 0) || (iLocalSensor >= iTempMons) )
   {
      SetLastError( ERROR_QST_INVALID_PARAMETER );
      return( -1 );
   }

   return( (int)stTempMonCfg[iLocalSensor].byMonitorUsage );
}

/*****************************************************************************/
/* GetTempReadingQst() - Returns reading from Temperature Monitor            */
/*****************************************************************************/

float GetTempReadingQst( int iLocalSensor )
{
   if( (iLocalSensor < 0) || (iLocalSensor >= iTempMons) )
   {
      SetLastError( ERROR_QST_INVALID_PARAMETER );
      return( -1 );
   }

   if( !GetTempMonUpdate() )
      return( -1 );

   return( QST_TEMP_TO_FLOAT(stTempMonUpd.
                             stMonitorUpdate[iTempMonIndex[iLocalSensor]].
                               lfCurrentReading ) );
}

/*****************************************************************************/
/* GetTempHealthQst() - Returns health for Temperature Monitor. Monitor      */
/* Physical Health takes precedence over Threshold Health…                   */
/*****************************************************************************/

int GetTempHealthQst( int iLocalSensor )
{
   if( (iLocalSensor < 0) || (iLocalSensor >= iTempMons) )
   {
      SetLastError( ERROR_QST_INVALID_PARAMETER );
      return( -1 );
   }

   if( !GetTempMonUpdate() )
      return( -1 );
```

```
    // Give precedence to monitor status over health status

    if( stTempMonUpd.
          stMonitorUpdate[iTempMonIndex[iLocalSensor]].
            stMonitorStatus.
               uMonitorStatus )

       return( stTempMonUpd.
                 stMonitorUpdate[iTempMonIndex[iLocalSensor]].
                   stMonitorStatus.
                     uMonitorStatus );
    else
       return( stTempMonUpd.
                 stMonitorUpdate[iTempMonIndex[iLocalSensor]].
                   stMonitorStatus.
                     uThresholdStatus );
}

/*****************************************************************************/
/* GetTempThreshQst() - Returns health thresholds for Temperature Monitor    */
/*****************************************************************************/

BOOL GetTempThreshQst(  int iLocalSensor, float *pfNonCrit,
                        Float *pfCrit, float *pfNonRecov          )
{
   if( (iLocalSensor < 0) || (iLocalSensor >= iTempMons) )
   {
      SetLastError( ERROR_QST_INVALID_PARAMETER );
      return( FALSE );
   }

   *pfNonCrit  = QST_TEMP_TO_FLOAT(
                    stTempMonCfg[iTempMonIndex[iLocalSensor]].
                       lfTempNonCritical );

   *pfCrit     = QST_TEMP_TO_FLOAT(
                    stTempMonCfg[iTempMonIndex[iLocalSensor]].
                       lfTempCritical );

   *pfNonRecov = QST_TEMP_TO_FLOAT(
                    stTempMonCfg[iTempMonIndex[iLocalSensor]].
                       lfTempNonRecoverable );

   return( TRUE );
}
```

3.3.6 Updating Health Thresholds

The following code demonstrates how one can apply new health thresholds for a particular Temperature Monitor. A similar process can be used to apply updates for configured Fan Speed and Voltage Monitors.

The example uses command SetTemperatureMonitorHealthThresholds to deliver the updated thresholds to the Intel® QST. If successful, it also updates the response buffer for the GetTemperatureMonitorUpdate command, so that subsequent attempts to obtain Health Thresholds can be fulfilled without requiring communication with the Intel® QST.

```
/*****************************************************************************/
/* SetTempThreshQst() - Updates health thresholds for Temperature Monitor    */
/* This operation could fail if a Lock Mask that was sent by the BIOS that   */
/* precludes Threshold updating                                              */
/*****************************************************************************/

BOOL GetTempThreshQst(  int iLocalSensor, float fNonCrit,
                        Float fCrit, float fNonRecov          )
{
   QST_SET_TEMP_MON_THRESHOLDS_CMD   stCmd;
   QST_GENERIC_RSP                   stRsp;
   int                               iRemIndex;

   if( (iLocalSensor < 0) || (iLocalSensor >= iTempMons) )
   {
      SetLastError( ERROR_QST_INVALID_PARAMETER );
      return( FALSE );
   }

   iRemIndex = iTempMonIndex[iLocalSensor];

   stCmd.stHeader.byCommand       = QST_SET_TEMP_MON_THRESHOLDS;
   stCmd.stHeader.byEntity        = iRemIndex;
   stCmd.stHeader.wResponseLength = sizeof(QST_GENERIC_CMD);

   stCmd.lfTempNonCritical        = QST_TEMP_FROM_FLOAT(fNonCrit);
   stCmd.lfTempCritical           = QST_TEMP_FROM_FLOAT(fCrit);
   stCmd.lfTempNonRecoverable     = QST_TEMP_FROM_FLOAT(fNonRecov);

   stCmd.stHeader.wCommandLength  =
      QST_CMD_DATA_SIZE(QST_SET_TEMP_MON_THRESHOLDS_CMD);

   if (!pfQstCommand2( &stCmd, sizeof(QST_SET_TEMP_MON_THRESHOLDS_CMD),
                       &stRsp, sizeof(QST_GENERIC_RSP) ) )
      return( FALSE );

   if( stRsp.byStatus )
   {
      SetLastErrorQST( stRsp.byStatus );
      return( FALSE );
   }

   stTempMonCfg[iRemIndex].lfTempNonCritical    = stCmd.lfTempNonCritical;
   stTempMonCfg[iRemIndex].lfTempCritical       = stCmd.lfTempCritical;
   stTempMonCfg[iRemIndex].lfTempNonRecoverable = stCmd.lfTempNonRecoverable;

   return( TRUE );
}
```

3.3.7 Extracting/Updating the Intel® QST Configuration

The following code demonstrates how one may obtain a copy of the current Configuration Payload that defines how Intel® QST operates. It also shows how the Configuration Payload may be updated.

The example provides two routines, one that reads the current Configuration Payload and one that writes a new/updated Configuration Payload. A program that seeks to update portion(s) of the Configuration Payload must read the entire payload into a buffer, update the appropriate portion(s) and then write this (entire) buffer back. No support is provided for updating portions of the Configuration Payload in any other way.

Note: This example details how the extracted configuration can be expanded, in order to make access to its contents much simpler, and how this expanded (and presumably modified) configuration can then be compacted before sending it back to Intel® QST. The source for the modules that implement the expansion and compaction are provided in the Intel® QST Software Development Kit (SDK). The contents of the Configuration Payload are documented in Appendix A.

```
    #include "QstCompactConfig.h"
    #include "QstExpandConfig.h"

    /*****************************************************************************/
    /* GetConfig() - Obtains current Configuration from the QST Subsystem       */
    /*****************************************************************************/

    static const QST_CFG_COUNTS stCfgCounts =
    {
        QST_ABS_TEMP_MONITORS,
        QST_ABS_FAN_MONITORS,
        QST_ABS_VOLT_MONITORS,
        QST_ABS_CURR_MONITORS,
        QST_ABS_TEMP_RESPONSES,
        QST_ABS_FAN_CONTROLLERS
    };

    BOOL GetConfig( QST_ABS_PAYLOAD_STRUCT *pstCfg )
    {
        BOOL                            bSuccessful = FALSE;
        void                            *pvBuffer;
        MANIP_STATUS                    eStatus;
        QST_GENERIC_CMD                 stCmd;
        P_QST_GET_SUBSYSTEM_CONFIG_RSP  pstRsp;

        // Allocate a temporary buffer to hold response

        pvBuffer = calloc( 1, sizeof(QST_GET_SUBSYSTEM_CONFIG_RSP) );

        if( pvBuffer )
        {
            pstRsp = (P_QST_GET_SUBSYSTEM_CONFIG_RSP)pvBuffer;

            // Prepare the command header

            stCmd.stHeader.byCommand        = QST_GET_SUBSYSTEM_CONFIG;
            stCmd.stHeader.byEntity         = 0;
            stCmd.stHeader.wCommandLength   = 0;
            stCmd.stHeader.wResponseLength  =
sizeof(QST_GET_SUBSYSTEM_CONFIG_RSP);

            // Send the command to the QST Subsystem

            if( QstCommand2( &stCmd, sizeof(QST_GENERIC_CMD),
                             pstRsp, sizeof(QST_GET_SUBSYSTEM_CONFIG_RSP) ) )
            {
                // Process response returned by QST Subsystem

                if( pstRsp->byStatus == QST_CMD_SUCCESSFUL )
                {
                    eStatus = ExpandConfig( &stPayload, QST_ABS_PAYLOAD_SIZE,
                                  &stCfgCounts, &pstRsp->stConfigPayload,
                                  pstRsp->stConfigPayload.Header.PayloadLength );

                    if( eStatus != MANIP_SUCCESS )
                        SetLastError( ERROR_BAD_CONFIGURATION );
                    else
                        bSuccessful = TRUE;
                }
                else
                    SetQstError( pstRsp->byStatus );
            }

            free( pvBuffer );
```

```
    }
    else
       SetLastError( ERROR_NOT_ENOUGH_MEMORY );

    return( bSuccessful );
}

/*****************************************************************************/
/* PutConfig() - Delivers current configuration to the QST Subsystem         */
/*****************************************************************************/

BOOL PutConfig( QST_ABS_PAYLOAD_STRUCT *pstCfg )
{
    BOOL                            bSuccessful = FALSE;
    void                            *pvBuffer;
    MANIP_STATUS                    eStatus;
    UINT32                          dwCmpLen;
    QST_SET_SUBSYSTEM_CONFIG_CMD    *pstCmd;
    QST_GENERIC_RSP                 stRsp;

    // Allocate a temporary buffer to hold the command

    pvBuffer = calloc( 1, sizeof(QST_SET_SUBSYSTEM_CONFIG_CMD) );

    if( pvBuffer )
    {
        pstCmd = (P_QST_SET_SUBSYSTEM_CONFIG_CMD)pvBuffer;

        // Load the payload into the command data block

        dwCmpLen = sizeof(QST_ABS_PAYLOAD_SIZE);

        eStatus = CompactConfig( pstCfg, QST_ABS_PAYLOAD_SIZE,
                                 &pstCmd->stConfigPayload, &dwCmpLen );

        if( eStatus != MANIP_SUCCESS )
        {
            // Prepare the command header

            pstCmd->stHeader.byCommand        = QST_SET_SUBSYSTEM_CONFIG;
            pstCmd->stHeader.byEntity         = 0;
            pstCmd->stHeader.wCommandLength   = dwCmpLen;
            pstCmd->stHeader.wResponseLength  = sizeof(QST_GENERIC_RSP);

            // Send the command to the QST Subsystem

            if( QstCommand2( pstCmd, sizeof(QST_SET_SUBSYSTEM_CONFIG_CMD),
                             &stRsp, sizeof(QST_GENERIC_RSP) ) )
            {
               // Process response returned by QST Subsystem

               if( stRsp.byStatus == QST_CMD_SUCCESSFUL )
                   bSuccessful = TRUE;
               else
                   SetQstError( stRsp.byStatus );
            }
        }
        else
            SetLastError( ERROR_BAD_CONFIGURATION );

        free( pvBuffer );
    }
    else
        SetLastError( ERROR_NOT_ENOUGH_MEMORY );

    return( bSuccessful );
}
```

3.3.8 Manually Controlling Fan Speed

The following code demonstrates how one may override the fan speed control decision-making processes of Intel® QST and manually set the duty cycle that is output by a particular Fan Speed Controller.

The example provides two functions. The first places a Fan Speed Controller into manual control mode and then sets the Current Duty Cycle to a specific (percentage) value. The second returns a Fan Speed Controller to automatic control mode (which results in the Current Duty Cycle being set by Intel® QST, based upon the present configuration and temperatures).

```
   /*****************************************************************************/
   /* SetDutyManualQst() - Sets duty cycle of the Fan Speed Controller          */
   /*****************************************************************************/

   BOOL SetDutyManualQst( int iLocSensor, float fDutyCycle )
   {
      QST_SET_FAN_CTRL_DUTY_CMD   stCmd;
      QST_GENERIC_RSP             stRsp;
      int                         iRemSensor;

      if( (iLocSensor < 0) || (iLocSensor >= iFanCtrls) )
      {
         SetLastError( ERROR_QST_INVALID_PARAMETER );
         return( FALSE );
      }

      // Abort if updates aren't allowed

      if( !bFanCtrlUpdatable )
      {
         SetLastError( ERROR_QST_CAPABILITY_LOCKED );
         return( FALSE );
      }

      // Send update to QST Subsystem

      stCmd.stHeader.byCommand       = QST_SET_FAN_CTRL_DUTY;
      stCmd.stHeader.byEntity        = iFanCtrlIndex[iLocSensor];
      stCmd.stHeader.wCommandLength  =
QST_CMD_DATA_SIZE(QST_SET_FAN_CTRL_DUTY_CMD);
      stCmd.stHeader.wResponseLength = sizeof(QST_GENERIC_RSP);

      stCmd.uDutyCycle               = QST_DUTY_FROM_FLOAT(fDutyCycle);

      if( !QstCommand2( &stCmd, sizeof(QST_SET_FAN_CTRL_DUTY_CMD),
                        &stRsp, sizeof(QST_GENERIC_RSP) ) )
         return( FALSE );

      if( stRsp.byStatus )
      {
         SetLastErrorQST( stRsp.byStatus );
         return( FALSE );
      }

      return( TRUE );
   }

   /*****************************************************************************/
   /* SetDutyAutoQst() - Puts Fan Speed Controller back into auto mode          */
   /*****************************************************************************/

   BOOL SetDutyAutoQst( int iLocSensor )
   {
      QST_GENERIC_CMD             stCmd;
      QST_GENERIC_RSP             stRsp;
      int                         iRemSensor;
```

```
      if( (iLocSensor < 0) || (iLocSensor >= iFanCtrls) )
      {
         SetLastError( ERROR_QST_INVALID_PARAMETER );
         return( FALSE );
      }

      iRemSensor = iFanCtrlIndex[iLocSensor];

      // Abort if updates aren't allowed

      if( !bFanCtrlUpdatable )
      {
         SetLastError( ERROR_QST_CAPABILITY_LOCKED );
         return( FALSE );
      }

      // Send update to QST Subsystem

      stCmd.stHeader.byCommand       = QST_SET_FAN_CTRL_AUTO;
      stCmd.stHeader.byEntity        = iRemSensor;
      stCmd.stHeader.wCommandLength  = QST_CMD_DATA_SIZE(QST_GENERIC_CMD);
      stCmd.stHeader.wResponseLength = sizeof(QST_GENERIC_RSP);

      if( !QstCommand2( &stCmd, sizeof(QST_GENERIC_CMD),
                        &stRsp, sizeof(QST_GENERIC_RSP) ) )
         return( FALSE );

      if( stRsp.byStatus )
      {
         SetLastErrorQST( stRsp.byStatus );
         return( FALSE );
      }

      return( TRUE );
   }
```

3.3.9 Updating Virtual Temperature Sensors

The following code demonstrates how an application can update the temperature readings within a specific virtual temperature sensor. The example loosely shows the function within a Windows* Service that takes temperature readings obtained from a hard drive and passes these readings Intel® QST.

```
   /*****************************************************************************/
   /* UpdateMonTemp() - Updates temperature reading for a specific Temperature */
   /* monitor. Remember, indexes are zero-based but monitors are numbered from */
   /* one. Thus, index 5 would be used for TemperatureMonitor6...               */
   /*****************************************************************************/

   BOOL UpdateMonTemp( int iMonIndex, float fMonTemp )
   {
      QST_SET_TEMP_MON_READING_CMD   stCmd;
      QST_GENERIC_RSP                stRsp;

      // Send update to QST Subsystem

      stCmd.stHeader.byCommand       = QST_SET_FAN_CTRL_AUTO;
      stCmd.stHeader.byEntity        = iRemSensor;
      stCmd.stHeader.wCommandLength  =
QST_CMD_DATA_SIZE(QST_SET_TEMP_MON_READING_CMD);
      stCmd.stHeader.wResponseLength = sizeof(QST_GENERIC_RSP);
      stCmd.lfTempReading = QST_TEMP_FROM_FLOAT(fMonTemp);

      if( !QstCommand2( &stCmd, sizeof(QST_GENERIC_CMD),
                        &stRsp, sizeof(QST_GENERIC_RSP)   ) )
         return( FALSE );
```

```
    if( stRsp.byStatus )
    {
       SetLastErrorQST( stRsp.byStatus );
       return( FALSE );
    }

    return( TRUE );
}
```

3.3.10 Sending Commands Directly to SST Devices

The following code demonstrates how one may send commands directly to sensors/controllers within devices on the SST Bus. Since the sensors/controllers within the chipset itself are accessed through a SST device abstraction, this example code can also be used to access these sensors/controllers as well.

SST Devices are, by design, absolutely consistent in their support for reading/writing from/to single Sensors/Controllers. They accept commands with a single word quantity when writing and commands that return a single word quantity when reading. As a result, we can define generic functions such as the first two shown below, which generically support reading from a device and writing to a device.

The third function included in the example is more specific. It supports access to the current reading from a temperature sensor within an SST device.

```
/*****************************************************************************/
/* SSTGet() - Executes the specified Get-Data (Read) SST Device Command      */
/*****************************************************************************/

static BOOL SSTGet( UINT8 byAddress, UINT8 byCommand, UINT16 *pwData )
{
   QST_SST_PASS_THROUGH_CMD stCmd;
   QST_SST_PASS_THROUGH_RSP stRsp;

   // Setup SST Command Packet

   stCmd.stSSTPacket.stSSTHeader.bySSTAddress  = byAddress;
   stCmd.stSSTPacket.stSSTHeader.byWriteLength = 1;  // just command byte
   stCmd.stSSTPacket.stSSTHeader.byReadLength  = 2;  // word reading
   stCmd.stSSTPacket.byCommandByte             = byCommand;

   // Setup QST Command Packet

   stCmd.stHeader.byCommand       = QST_SST_PASS_THROUGH;
   stCmd.stHeader.byEntity        = 0;
   stCmd.stHeader.wCommandLength  = QST_SST_CMD_DATA(0);
   stCmd.stHeader.wResponseLength = QST_SST_RSP_SIZE(1);

   // Perform Transaction

   if( !QstCommand2( &stCmd, QST_SST_CMD_SIZE(0),
                     &stRsp, QST_SST_RSP_SIZE(1) ) )
      return( FALSE );

   // Check if Subsystem rejected the command (or had SST Bus issue)

   if( stRsp.byStatus )
   {
      SetLastErrorQST(stRsp.byStatus );
      return( FALSE );
   }

   // Successful; give them the response data (word) to process
```

```
    *pwData = stRsp.wValue[0];
    return( TRUE );
}

/*****************************************************************************/
/* SSTSet() - Executes the specified Set-Data (Write) SST Device Command     */
/*****************************************************************************/

static BOOL SSTSet( UINT8 byAddress, UINT8 byCommand, UINT16 wData )
{
   QST_SST_PASS_THROUGH_CMD   stCmd;
   QST_SST_PASS_THROUGH_RSP   stRsp;

   // Setup SST Command Packet

   stCmd.stSSTPacket.stSSTHeader.bySSTAddress  = byAddress;
   stCmd.stSSTPacket.stSSTHeader.byWriteLength = 3;
   stCmd.stSSTPacket.stSSTHeader.byReadLength  = 0;
   stCmd.stSSTPacket.byCommandByte             = byCommand;
   stCmd.stSSTPacket.wCommandData[0]           = wData;

   // Setup QST Command Packet Wrapper

   stCmd.stHeader.byCommand       = QST_SST_PASS_THROUGH;
   stCmd.stHeader.byEntity        = 0;
   stCmd.stHeader.wCommandLength  = QST_SST_CMD_DATA(1);
   stCmd.stHeader.wResponseLength = QST_SST_RSP_SIZE(0);

   // Perform Transaction

   if( !QstCommand2( &stCmd, QST_SST_CMD_SIZE(1),
                     &stRsp, QST_SST_RSP_SIZE(0) ) )
      return( FALSE );

   // Check if Subsystem rejected the command (or had SST Bus issue)

   if( stRsp.byStatus )
   {
      SetLastErrorQST(stRsp.byStatus );
      return( FALSE );
   }

   return( TRUE );
}

/*****************************************************************************/
/* GetTemp() - Obtains a temperature reading from the specified sensor       */
/*****************************************************************************/

static BOOL GetTemp( UINT8 byAddress, UINT8 byCommand, float *pfReading )
{
   QST_SST_PASS_THROUGH_CMD   stCmd;
   QST_SST_PASS_THROUGH_RSP   stRsp;

   // Setup SST Command Packet

   stCmd.stSSTPacket.stSSTHeader.bySSTAddress  = byAddress;
   stCmd.stSSTPacket.stSSTHeader.byWriteLength = 1;  // just command byte
   stCmd.stSSTPacket.stSSTHeader.byReadLength  = 2;  // word reading
   stCmd.stSSTPacket.byCommandByte             = byCommand;

   // Setup QST Command Packet

   stCmd.stHeader.byCommand       = QST_SST_PASS_THROUGH;
   stCmd.stHeader.byEntity        = 0;
   stCmd.stHeader.wCommandLength  = QST_SST_CMD_DATA(0);
   stCmd.stHeader.wResponseLength = QST_SST_RSP_SIZE(1);

   // Perform Transaction

   if( !QstCommand2( &stCmd, QST_SST_CMD_SIZE(0),
                     &stRsp, QST_SST_RSP_SIZE(1) ) )
      return( FALSE );

   // Check if Subsystem rejected the command (or had SST Bus issue)
```

```
        if( stRsp.byStatus )
        {
           SetLastErrorQST(stRsp.byStatus );
           return( FALSE );
        }

        // Successful; give them the response data (word) to process

        *pfReading = SST_TEMP_TO_FLOAT(stRsp.wValue[0]);
        return( TRUE );
    }
```

§

4 Intel® QST Health Monitoring

In order to simplify the process for Environmental Health Monitoring, Intel® QST has been designed to free Monitoring Applications from requiring any knowledge of the hardware configuration. To further simplify this process, a Health Monitoring Instrumentation Layer (IL) is being provided. This Instrumentation Layer frees applications from the tedium of managing the raw data from Intel® QST. It provides a simple interface for monitoring support, yet offers the richness necessary to meet the varied requirements of manageability infrastructures (CIM/WBEM/WMI, DMI, etc.).

The Instrumentation Layer is provided in different forms for each of the O/S environments: a Dynamic Link Library (QstInst.dll) for Windows*, a Shared-Object File (libQstInst.so) for Linux and a linkable library (QstComm6x.lib) for DOS.

Note: The function declarations and definitions specific to the use of the Instrumentation Layer are provided in header file QstInst.h.

4.1 Using the Health Monitoring API

4.1.1 Using Windows* QstInst DLL

The QstInst DLL provides the API necessary for Windows-based software applications to instrument the Health Monitoring capabilities of QST, receive health status and readings from the various sensors and receive health status and controller settings from the various fan controllers.

4.1.1.1 DLL Function Access

Applications can either statically or dynamically link themselves with the QstInst DLL and the functions that it provides.

4.1.1.1.1 Static Linking

Statically linked DLLs are automatically loaded into the address space of the application when the application is loaded and will remain there until the application is terminated. Applications can invoke the functions of the DLL just as they would any other functions.

In order to statically link the QstComm DLL with their applications, developers should include the provided **QstInst.lib** file into their applications' project.

Note: The use of static linking is NOT recommended. If something prevents the DLL from loading, this will also cause the load of the application to fail – and will do so without providing any error indication…

4.1.1.1.2 Dynamic Linking

Dynamically linked DLLs are loaded into an application's address space manually, when the application actually requires their services. Manual loading of DLLs is normally only used when applications utilize their services intermittently and wish to minimize their memory footprint whenever possible.

In order to load the QstInst DLL, applications use WIN32* function **LoadLibrary()**. Once the DLL has been loaded, pointers to the DLL functions must be built using WIN32 function **GetProcAddress()**. The functions may then be invoked indirectly through these pointers.

When the application is finished using the services of the DLL, it should unload it using WIN32 function **FreeLibrary()**. Alternatively, the O/S will automatically unload the DLL when the application is terminated.

Example

The following code demonstrates support for the Dynamic Linking process. It provides initialization/finalization functions to handle the loading and unloading of the DLL and wrapper functions that make it possible to utilize the functions as if they were present locally.

```
   #include <windows.h>
   #include "QstInst.h"

   /*****************************************************************************/
   /* Variables                                                                 */
   /*****************************************************************************/

   static BOOL                                  bQstInstDLL = FALSE;
   static HMODULE                               hQstInstDLL = NULL;

   static DWORD                                 dwLoadError = ERROR_INVALID_DLL;

   static PFN_QST_GET_SENSOR_COUNT              pfQstGetSensorCount;
   static PFN_QST_GET_SENSOR_CONFIGURATION      pfQstGetSensorConfiguration;
   static PFN_QST_GET_SENSOR_THRESHOLDS_HIGH    pfQstGetSensorThresholdsHigh;
   static PFN_QST_GET_SENSOR_THRESHOLDS_LOW     pfQstGetSensorThresholdsLow;
   static PFN_QST_GET_SENSOR_HEALTH             pfQstGetSensorHealth;
   static PFN_QST_GET_SENSOR_READING            pfQstGetSensorReading;
   static PFN_QST_SET_SENSOR_THRESHOLDS_HIGH    pfQstSetSensorThresholdsHigh;
   static PFN_QST_SENSOR_THRESHOLDS_HIGH_CHANGED
pfQstSensorThresholdsHighChanged;
   static PFN_QST_SET_SENSOR_THRESHOLDS_LOW     pfQstSetSensorThresholdsLow;
   static PFN_QST_SENSOR_THRESHOLDS_LOW_CHANGED pfQstSensorThresholdsLowChanged;
   static PFN_QST_GET_CONTROLLER_COUNT          pfQstGetControllerCount;
   static PFN_QST_GET_CONTROLLER_CONFIGURATION  pfQstGetControllerConfiguration;
   static PFN_QST_GET_CONTROLLER_STATE          pfQstGetControllerState;
   static PFN_QST_GET_CONTROLLER_DUTY_CYCLE     pfQstGetControllerDutyCycle;
   static PFN_QST_GET_POLLING_INTERVAL          pfQstGetPollingInterval;
   static PFN_QST_SET_POLLING_INTERVAL          pfQstSetPollingInterval;
   static PFN_QST_POLLING_INTERVAL_CHANGED      pfQstPollingIntervalChanged;

   /*****************************************************************************/
   /* QstInstInitialize() - Initializes support for using the QstInst DLL       */
   /*****************************************************************************/

   BOOL QstInstInitialize( void )
   {
      // Load the Instrumentation Layer DLL

      hQstInstDLL = LoadLibrary( QST_INST_DLL );

      if( hQstInstDLL )
```

```
{
   pfQstGetSensorCount =
      (PFN_QST_GET_SENSOR_COUNT) GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_SENSOR_COUNT) );

   pfQstGetSensorConfiguration =
      PFN_QST_GET_SENSOR_CONFIGURATION)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_SENSOR_CONFIGURATION) );

   pfQstGetSensorThresholdsHigh =
      (PFN_QST_GET_SENSOR_THRESHOLDS_HIGH)GetProcAddress( hQstInstDLL,
          MAKEINTRESOURCE(QST_ORD_GET_SENSOR_THRESHOLDS_HIGH) );

   pfQstGetSensorThresholdsLow =
      (PFN_QST_GET_SENSOR_THRESHOLDS_LOW)GetProcAddress( hQstInstDLL,
          MAKEINTRESOURCE(QST_ORD_GET_SENSOR_THRESHOLDS_LOW) );

   pfQstGetSensorHealth =
      (PFN_QST_GET_SENSOR_HEALTH)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_SENSOR_HEALTH) );

   pfQstGetSensorReading =
      (PFN_QST_GET_SENSOR_READING)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_SENSOR_READING) );

   pfQstSetSensorThresholdsHigh =
      (PFN_QST_SET_SENSOR_THRESHOLDS_HIGH)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_SET_SENSOR_THRESHOLDS_HIGH) );

   pfQstSensorThresholdsHighChanged =
      (PFN_QST_SENSOR_THRESHOLDS_HIGH_CHANGED)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_SENSOR_THRESHOLDS_HIGH_CHANGED) );

   pfQstSetSensorThresholdsLow =
      (PFN_QST_SET_SENSOR_THRESHOLDS_LOW)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_SET_SENSOR_THRESHOLDS_LOW) );

   pfQstSensorThresholdsLowChanged =
      (PFN_QST_SENSOR_THRESHOLDS_LOW_CHANGED)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_SENSOR_THRESHOLDS_LOW_CHANGED) );

   pfQstGetControllerCount =
      (PFN_QST_GET_CONTROLLER_COUNT)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_CONTROLLER_COUNT) );

   pfQstGetControllerConfiguration  =
      (PFN_QST_GET_CONTROLLER_CONFIGURATION)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_CONTROLLER_CONFIGURATION) );

   pfQstGetControllerState =
      (PFN_QST_GET_CONTROLLER_STATE)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_CONTROLLER_STATE) );

   pfQstGetControllerDutyCycle =
      (PFN_QST_GET_CONTROLLER_DUTY_CYCLE)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_CONTROLLER_DUTY_CYCLE) );

   pfQstGetPollingInterval =
      (PFN_QST_GET_POLLING_INTERVAL)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_GET_POLLING_INTERVAL) );

   pfQstSetPollingInterval =
      (PFN_QST_SET_POLLING_INTERVAL)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_SET_POLLING_INTERVAL) );

   pfQstPollingIntervalChanged =
      (PFN_QST_POLLING_INTERVAL_CHANGED)GetProcAddress( hQstInstDLL,
         MAKEINTRESOURCE(QST_ORD_POLLING_INTERVAL_CHANGED) );

   // Verify success of pointer build

   if(    pfQstGetSensorCount
       && pfQstGetSensorConfiguration
       && pfQstGetSensorThresholdsHigh
       && pfQstGetSensorThresholdsLow
       && pfQstGetSensorHealth
```

```
            && pfQstGetSensorReading
            && pfQstSetSensorThresholdsHigh
            && pfQstSensorThresholdsHighChanged
            && pfQstSetSensorThresholdsLow
            && pfQstSensorThresholdsLowChanged
            && pfQstGetControllerCount
            && pfQstGetControllerConfiguration
            && pfQstGetControllerState
            && pfQstGetControllerDutyCycle
            && pfQstGetPollingInterval
            && pfQstSetPollingInterval
            && pfQstPollingIntervalChanged
          )
           return( bQstInstDLL = TRUE );

      // failed, so start unwinding things

      dwLoadError = GetLastError();

      FreeLibrary( hQstInstDLL );
      hQstInstDLL = NULL;
   }
   else
      dwLoadError = GetLastError();

   return( bQstInstDLL = FALSE );
}

/*****************************************************************************/
/* QstInstCleanup() - Cleans up the resources supporting use of the          */
/* Instrumentation Layer.                                                    */
/*****************************************************************************/

void QstInstCleanup( void )
{
   if( bQstInstDLL )
   {
      FreeLibrary( hQstInstDLL );

      bQstInstDLL = FALSE;
      hQstInstDLL = NULL;
   }
}

/*****************************************************************************/
/* Function Wrappers                                                         */
/*****************************************************************************/

BOOL APIENTRY QstGetSensorCount( QST_SENSOR_TYPE eType, int *piCount )
{
   if( bQstInstDLL )
      return( pfQstGetSensorCount( eType, piCount ) );
   else
   {
      SetLastError( dwLoadError );
      return( FALSE );
   }
}

BOOL APIENTRY QstGetSensorConfiguration( QST_SENSOR_TYPE eType, int iIndex,
    QST_FUNCTION *peFunction, BOOL *pbRelative, float *pfNominal )
{
   if( bQstInstDLL )
      return( pfQstGetSensorConfiguration( eType, iIndex, peFunction,
              pbRelative, pfNominal ) );
   else
   {
      SetLastError( dwLoadError );
      return( FALSE );
   }
}

BOOL APIENTRY QstGetSensorThresholdsHigh( QST_SENSOR_TYPE eType, int iIndex,
    float *pfNonCritical, float *pfCritical, float *pfNonRecoverable )
{
   if( bQstInstDLL )
```

```
            return( pfQstGetSensorThresholdsHigh( eType, iIndex, pfNonCritical,
                    pfCritical, pfNonRecoverable ) );
         else
         {
            SetLastError( dwLoadError );
            return( FALSE );
         }
      }

      BOOL APIENTRY QstGetSensorThresholdsLow( QST_SENSOR_TYPE eType, int iIndex,
          float *pfNonCritical, float *pfCritical, float *pfNonRecoverable )
      {
         if( bQstInstDLL )
            return( pfQstGetSensorThresholdsLow( eType, iIndex, pfNonCritical,
                    pfCritical, pfNonRecoverable ) );
         else
         {
            SetLastError( dwLoadError );
            return( FALSE );
         }
      }

      BOOL APIENTRY QstGetSensorHealth( QST_SENSOR_TYPE eType, int iIndex,
          QST_HEALTH *peHealth )
      {
         if( bQstInstDLL )
            return( pfQstGetSensorHealth( eType, iIndex, peHealth ) );
         else
         {
            SetLastError( dwLoadError );
            return( FALSE );
         }
      }

      BOOL APIENTRY QstGetSensorReading( QST_SENSOR_TYPE eType, int iIndex,
          float *pfReading )
      {
         if( bQstInstDLL )
            return( pfQstGetSensorReading( eType, iIndex, pfReading ) );
         else
         {
            SetLastError( dwLoadError );
            return( FALSE );
         }
      }

      BOOL APIENTRY QstSetSensorThresholdsHigh( QST_SENSOR_TYPE eType, int iIndex,
          float fNonCritical, float fCritical, float fNonRecoverable )
      {
         if( bQstInstDLL )
            return( pfQstSetSensorThresholdsHigh( eType, iIndex, fNonCritical,
                    fCritical, fNonRecoverable ) );
         else
         {
            SetLastError( dwLoadError );
            return( FALSE );
         }
      }

      BOOL APIENTRY QstSensorThresholdsHighChanged( QST_SENSOR_TYPE eType, int
iIndex,
          time_t tLast, BOOL *pbUpdated )
      {
         if( bQstInstDLL )
            return( pfQstSensorThresholdsHighChanged(eType,iIndex,tLast,pbUpdated)
);
         else
         {
            SetLastError( dwLoadError );
            return( FALSE );
         }
      }

      BOOL APIENTRY QstSetSensorThresholdsLow( QST_SENSOR_TYPE eType, int iIndex,
          float fNonCritical, float fCritical, float fNonRecoverable )
      {
```

```
        if( bQstInstDLL )
            return( pfQstSetSensorThresholdsLow( eType, iIndex, fNonCritical,
                    fCritical, fNonRecoverable ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstSensorThresholdsLowChanged( QST_SENSOR_TYPE eType, int
iIndex,
         time_t tLast, BOOL *pbUpdtd )
    {
        if( bQstInstDLL )
            return( pfQstSensorThresholdsLowChanged( eType, iIndex, tLast, pbUpdtd )
);
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstGetControllerCount( int *piCount )
    {
        if( bQstInstDLL )
            return( pfQstGetControllerCount( piCount ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstGetControllerConfiguration( int iIndex, QST_FUNCTION *peFunc
)
    {
        if( bQstInstDLL )
            return( pfQstGetControllerConfiguration( iIndex, peFunc ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstGetControllerState( int iIndex, QST_HEALTH *peHealth,
         QST_CONTROL_STATE *peState )
    {
        if( bQstInstDLL )
            return( pfQstGetControllerState( iIndex, peHealth, peState ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstGetControllerDutyCycle( int iIndex, float *pfDuty )
    {
        if( bQstInstDLL )
            return( pfQstGetControllerDutyCycle( iIndex, pfDuty ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstGetPollingInterval( DWORD *pdwInterval )
    {
        if( bQstInstDLL )
            return( pfQstGetPollingInterval( pdwInterval ) );
        else
        {
```

```
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstSetPollingInterval( DWORD dwInterval )
    {
        if( bQstInstDLL )
            return( pfQstSetPollingInterval( dwInterval ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }

    BOOL APIENTRY QstPollingIntervalChanged( time_t tLast, BOOL *pbUpdated )
    {
        if( bQstInstDLL )
            return( pfQstPollingIntervalChanged( tLast, pbUpdated ) );
        else
        {
            SetLastError( dwLoadError );
            return( FALSE );
        }
    }
```

Note: Prototypes for functions QstInstInitialize() and QstInstCleanup() have been included in header file QstInst.h. In order to use these prototypes, you should define symbol DYNAMIC_DLL_LOADING within your project file and include example source file QstInst.c (which provides the code shown above).

4.1.2 Using Linux*/Solaris* QstInst Shared-Object File

The Linux QstInst Shared-Object (SO) File (libQstInst.so) provides the API necessary for Linux-based software applications to instrument the Health Monitoring capabilities of QST, receive health status and readings from the various sensors and receive health status and controller settings from the various fan controllers.

Note: For more information regarding SO files, consult the Program Library HOWTO. Additional information is included in the C++ dlopen mini HOWTO. Both can be downloaded from The Linux Documentation Project (http://tldp.org/docs.html).

4.1.2.1 SO File Function Access

Applications may either statically or dynamically link themselves with the QstInst SO.

4.1.2.1.1 Static Linking

Statically linked SO Files are automatically loaded into the address space of the application when the application is itself loaded and will remain there until the application is terminated. Applications may invoke the functions of the SO File just as they would any other functions.

In order to statically link the QstComm SO File with their applications, developers should include the libQstComm.so file into their applications' project. This is accomplished by, for example, adding the "-lQstComm" parameter to GCC invocation.

4.1.2.1.2 Dynamic Linking

Dynamically linked SO Files are manually loaded into an application's address space when the application actually requires their services. Manual loading of SO Files is normally used only when applications utilize their services intermittently and wish to minimize their memory footprint where possible.

SO Files are dynamically loaded and unloaded using the dlopen API. Function dlopen() is used to load the SO file. Once loaded, function dlsym() is used to obtain pointers for the various functions. These functions can then be invoked indirectly through the pointers provided. Finally, when the program is finished with the SO file, it unloads it using function dlclose().

Example

The following code example demonstrates how one can facilitate the dynamic load, use and unload of the QstComm SO File:

```
/*****************************************************************************/
/* Definitions                                                               */
/*****************************************************************************/

typedef BOOL (*PFN_QST_GET_SENSOR_COUNT)
(
    IN  QST_SENSOR_TYPE   SensorType,
    OUT int               *pSensorCount
);

typedef BOOL (*PFN_QST_GET_SENSOR_CONFIGURATION)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    OUT QST_FUNCTION      *pSensorFunction,
    OUT BOOL              *pRelativeReadings,
    OUT float             *pNominalReading
);

typedef BOOL (*PFN_QST_GET_SENSOR_THRESHOLDS_HIGH)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    OUT float             *pThresholdNonCritical,
    OUT float             *pThresholdCritical,
    OUT float             *pThresholdNonRecoverable
);

typedef BOOL (*PFN_QST_GET_SENSOR_THRESHOLDS_LOW)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    OUT float             *pThresholdNonCritical,
    OUT float             *pThresholdCritical,
    OUT float             *pThresholdNonRecoverable
);

typedef BOOL (*PFN_QST_GET_SENSOR_HEALTH)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    OUT QST_HEALTH        *pSensorHealth
);

typedef BOOL (*PFN_QST_GET_SENSOR_READING)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    OUT float             *pSensorReading
);
```

```
typedef BOOL (*PFN_QST_SET_SENSOR_THRESHOLDS_HIGH)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    IN  float             ThresholdNonCritical,
    IN  float             ThresholdCritical,
    IN  float             ThresholdNonRecoverable
);

typedef BOOL (*PFN_QST_SENSOR_THRESHOLDS_HIGH_CHANGED)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    IN  time_t            LastUpdate,
    OUT BOOL              *pUpdated
);

typedef BOOL (*PFN_QST_SET_SENSOR_THRESHOLDS_LOW)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    IN  float             ThresholdNonCritical,
    IN  float             ThresholdCritical,
    IN  float             ThresholdNonRecoverable
);

typedef BOOL (*PFN_QST_SENSOR_THRESHOLDS_LOW_CHANGED)
(
    IN  QST_SENSOR_TYPE   SensorType,
    IN  int               SensorIndex,
    IN  time_t            LastUpdate,
    OUT BOOL              *pUpdated
);

typedef BOOL (*PFN_QST_GET_CONTROLLER_COUNT)
(
    OUT int               *pControllerCount
);

typedef BOOL (*PFN_QST_GET_CONTROLLER_CONFIGURATION)
(
    IN  int               ControllerIndex,
    OUT QST_FUNCTION      *pControllerFunction
);

typedef BOOL (*PFN_QST_GET_CONTROLLER_STATE)
(
    IN  int               ControllerIndex,
    OUT QST_HEALTH        *pHealthState,
    OUT QST_CONTROL_STATE *pControlState
);

typedef BOOL (*PFN_QST_GET_CONTROLLER_DUTY_CYCLE)
(
    IN  int               ControllerIndex,
    OUT float             *pControllerDuty
);

typedef BOOL (*PFN_QST_GET_POLLING_INTERVAL)
(
    OUT DWORD             *pPollingInterval
);

typedef BOOL (*PFN_QST_SET_POLLING_INTERVAL)
(
    IN  DWORD             PollingInterval
);

typedef BOOL (*PFN_QST_POLLING_INTERVAL_CHANGED)
(
    IN  time_t            LastUpdate,
    OUT BOOL              *pUpdated
);

/*****************************************************************************/
```

```
    /* Variables                                                                */
    /****************************************************************************/

    static void *                                   hQstInstSO = NULL;

    static PFN_QST_GET_SENSOR_COUNT                 pfQstGetSensorCount;
    static PFN_QST_GET_SENSOR_CONFIGURATION         pfQstGetSensorConfiguration;
    static PFN_QST_GET_SENSOR_THRESHOLDS_HIGH       pfQstGetSensorThresholdsHigh;
    static PFN_QST_GET_SENSOR_THRESHOLDS_LOW        pfQstGetSensorThresholdsLow;
    static PFN_QST_GET_SENSOR_HEALTH                pfQstGetSensorHealth;
    static PFN_QST_GET_SENSOR_READING               pfQstGetSensorReading;
    static PFN_QST_SET_SENSOR_THRESHOLDS_HIGH       pfQstSetSensorThresholdsHigh;
    static PFN_QST_SENSOR_THRESHOLDS_HIGH_CHANGED
pfQstSensorThresholdsHighChanged;
    static PFN_QST_SET_SENSOR_THRESHOLDS_LOW        pfQstSetSensorThresholdsLow;
    static PFN_QST_SENSOR_THRESHOLDS_LOW_CHANGED    pfQstSensorThresholdsLowChanged;
    static PFN_QST_GET_CONTROLLER_COUNT             pfQstGetControllerCount;
    static PFN_QST_GET_CONTROLLER_CONFIGURATION     pfQstGetControllerConfiguration;
    static PFN_QST_GET_CONTROLLER_STATE             pfQstGetControllerState;
    static PFN_QST_GET_CONTROLLER_DUTY_CYCLE        pfQstGetControllerDutyCycle;
    static PFN_QST_GET_POLLING_INTERVAL             pfQstGetPollingInterval;
    static PFN_QST_SET_POLLING_INTERVAL             pfQstSetPollingInterval;
    static PFN_QST_POLLING_INTERVAL_CHANGED         pfQstPollingIntervalChanged;

    /****************************************************************************/
    /* QstInitialize() - Initializes I/F for accessing service of the SO File   */
    /****************************************************************************/

    BOOL QstInitialize( void )
    {
       // Load the SO File

       hQstInstSO = dlopen( “libQstInst.so.1”, RTLD_LAZY );

       if( hQstInstSO )
       {
          // Build pointer to SO File's function(s)

          pfQstGetSensorCount =
             dlsym( hQstInstSO, “QstGetSensorCount” );

          pfQstGetSensorConfiguration =
             dlsym( hQstInstSO, “QstGetSensorConfiguration” );

          pfQstGetSensorThresholdsHigh =
             dlsym( hQstInstSO, “QstGetSensorThresholdsHigh” );

          pfQstGetSensorThresholdsLow =
             dlsym( hQstInstSO, “QstGetSensorThresholdsLow” );

          pfQstGetSensorHealth =
             dlsym( hQstInstSO, “QstGetSensorHealth” );

          pfQstGetSensorReading =
             dlsym( hQstInstSO, “QstGetSensorReading” );

          pfQstSetSensorThresholdsHigh =
             dlsym( hQstInstSO, “QstSetSensorThresholdsHigh” );

          pfQstSensorThresholdsHighChanged =
             dlsym( hQstInstSO, “QstSensorThresholdsHighChanged” );

          pfQstSetSensorThresholdsLow =
             dlsym( hQstInstSO, “QstSetSensorThresholdsLow” );

          pfQstSensorThresholdsLowChanged =
             dlsym( hQstInstSO, “QstSensorThresholdsLowChanged” );

          pfQstGetControllerCount =
             dlsym( hQstInstSO, “QstGetControllerCount” );

          pfQstGetControllerConfiguration =
             dlsym( hQstInstSO, “QstGetControllerConfiguration” );

          pfQstGetControllerState =
             dlsym( hQstInstSO, “QstGetControllerState” );
```

```
      pfQstGetControllerDutyCycle =
         dlsym( hQstInstSO, "QstGetControllerDutyCycle" );

      pfQstGetPollingInterval =
         dlsym( hQstInstSO, "QstGetPollingInterval" );

      pfQstSetPollingInterval =
         dlsym( hQstInstSO, "QstSetPollingInterval" );

      pfQstPollingIntervalChanged =
         dlsym( hQstInstSO, "QstPollingIntervalChanged" );

      if(    pfQstGetSensorCount
          && pfQstGetSensorConfiguration
          && pfQstGetSensorThresholdsHigh
          && pfQstGetSensorThresholdsLow
          && pfQstGetSensorHealth
          && pfQstGetSensorReading
          && pfQstSetSensorThresholdsHigh
          && pfQstSensorThresholdsHighChanged
          && pfQstSetSensorThresholdsLow
          && pfQstSensorThresholdsLowChanged
          && pfQstGetControllerCount
          && pfQstGetControllerConfiguration
          && pfQstGetControllerState
          && pfQstGetControllerDutyCycle
          && pfQstGetPollingInterval
          && pfQstSetPollingInterval
          && pfQstPollingIntervalChanged
        )
         return( TRUE );

      dlclose( hQstInstSO );
      hQstInstSO = NULL;
      errno = ENOENT;
   }
   else
      errno = ENODEV;

   return( FALSE );
}

/*****************************************************************************/
/* QstCleanup() - Cleans up I/F to the SO File                               */
/*****************************************************************************/

void QstCleanup( void )
{
   if( hQstInstSO )
   {
      dlclose( hQstInstSO );
      hQstInstSO = NULL;
   }
}

/*****************************************************************************/
/* Function Wrappers                                                         */
/*****************************************************************************/

BOOL QstGetSensorCount( QST_SENSOR_TYPE eType, int *piCount )
{
   if( hQstInstSO )
      return( pfQstGetSensorCount( eType, piCount ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstGetSensorConfiguration( QST_SENSOR_TYPE eType, int iIndex,
    QST_FUNCTION *peFunction, BOOL *pbRelative, float *pfNominal )
{
   if( hQstInstSO )
      return( pfQstGetSensorConfiguration( eType, iIndex, peFunction,
```

```
                pbRelative, pfNominal ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstGetSensorThresholdsHigh( QST_SENSOR_TYPE eType, int iIndex,
    float *pfNonCritical, float *pfCritical, float *pfNonRecoverable )
{
   if( hQstInstSO )
      return( pfQstGetSensorThresholdsHigh( eType, iIndex, pfNonCritical,
                pfCritical, pfNonRecoverable ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstGetSensorThresholdsLow( QST_SENSOR_TYPE eType, int iIndex,
    float *pfNonCritical, float *pfCritical, float *pfNonRecoverable )
{
   if( hQstInstSO )
      return( pfQstGetSensorThresholdsLow( eType, iIndex, pfNonCritical,
                pfCritical, pfNonRecoverable ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstGetSensorHealth( QST_SENSOR_TYPE eType, int iIndex,
    QST_HEALTH *peHealth )
{
   if( hQstInstSO )
      return( pfQstGetSensorHealth( eType, iIndex, peHealth ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstGetSensorReading( QST_SENSOR_TYPE eType, int iIndex,
    float *pfReading )
{
   if( hQstInstSO )
      return( pfQstGetSensorReading( eType, iIndex, pfReading ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstSetSensorThresholdsHigh( QST_SENSOR_TYPE eType, int iIndex,
    float fNonCritical, float fCritical, float fNonRecoverable )
{
   if( hQstInstSO )
      return( pfQstSetSensorThresholdsHigh( eType, iIndex, fNonCritical,
                fCritical, fNonRecoverable ) );
   else
   {
      errno = ENOEXEC;
      return( FALSE );
   }
}

BOOL QstSensorThresholdsHighChanged( QST_SENSOR_TYPE eType, int iIndx,
    time_t tLast, BOOL *pbUpdtd )
{
   if( hQstInstSO )
```

```
            return( pfQstSensorThresholdsHighChanged( eType, iIndx, tLast, pbUpdtd )
);
        else
        {
            errno = ENOEXEC;
            return( FALSE );
        }
    }

    BOOL QstSetSensorThresholdsLow( QST_SENSOR_TYPE eType, int iIndex,
         float fNonCritical, float fCritical, float fNonRecoverable )
    {
        if( hQstInstSO )
            return( pfQstSetSensorThresholdsLow( eType, iIndex, fNonCritical,
                    fCritical, fNonRecoverable ) );
        else
        {
            errno = ENOEXEC;
            return( FALSE );
        }
    }

    BOOL QstSensorThresholdsLowChanged( QST_SENSOR_TYPE eType, int iIndex,
         time_t tLast, BOOL *pbUpdtd )
    {
        if( hQstInstSO )
            return( pfQstSensorThresholdsLowChanged( eType, iIndex, tLast, pbUpdtd )
);
        else
        {
            errno = ENOEXEC;
            return( FALSE );
        }
    }

    BOOL QstGetControllerCount( int *piCount )
    {
        if( hQstInstSO )
            return( pfQstGetControllerCount( piCount ) );
        else
        {
            errno = ENOEXEC;
            return( FALSE );
        }
    }

    BOOL QstGetControllerConfiguration( int iIndex, QST_FUNCTION *peFunc )
    {
        if( hQstInstSO )
            return( pfQstGetControllerConfiguration( iIndex, peFunc ) );
        else
        {
            errno = ENOEXEC;
            return( FALSE );
        }
    }

    BOOL QstGetControllerState( int iIndex, QST_HEALTH *peHealth,
         QST_CONTROL_STATE *peState )
    {
        if( hQstInstSO )
            return( pfQstGetControllerState( iIndex, peHealth, peState ) );
        else
        {
            errno = ENOEXEC;
            return( FALSE );
        }
    }

    BOOL QstGetControllerDutyCycle( int iIndex, float *pfDuty )
    {
        if( hQstInstSO )
            return( pfQstGetControllerDutyCycle( iIndex, pfDuty ) );
        else
        {
```

```
        errno = ENOEXEC;
        return( FALSE );
    }
}

BOOL QstGetPollingInterval( DWORD *pdwInterval )
{
    if( hQstInstSO )
        return( pfQstGetPollingInterval( pdwInterval ) );
    else
    {
        errno = ENOEXEC;
        return( FALSE );
    }
}

BOOL QstSetPollingInterval( DWORD dwInterval )
{
    if( hQstInstSO )
        return( pfQstSetPollingInterval( dwInterval ) );
    else
    {
        errno = ENOEXEC;
        return( FALSE );
    }
}

BOOL QstPollingIntervalChanged( time_t tLast, BOOL *pbUpdated )
{
    if( hQstInstSO )
        return( pfQstPollingIntervalChanged( tLast, pbUpdated ) );
    else
    {
        errno = ENOEXEC;
        return( FALSE );
    }
}
```

4.1.3 Using DOS QstInst Library

The DOS QstInst Libraries provides the API necessary for DOS-based software applications to instrument Intel® QST. To use these libraries, software must be built for a 32-bit address space and supported by a DOS Extender that eliminates the inherent DOS 640KB memory limit. Support is provided for building applications using the Open Watcom package, an open source version of Sybase's Watcom C/C++ compiler product.

Note: Applications will be built using the flat memory model. Versions of the DOS QstInst Library are provided for register-based parameter passing (QstInst6r.lib) and stack-based parameter passing (QstInst6s.lib) models.

4.1.3.1 Library Function Access

Applications will statically link themselves with the DOS QstInst Libraries. See Section 3.1.4.2 for an example batch file that shows how debug and release executables may be built for a program.

In order to allow the library to initialize itself, DOS applications are required to invoke function QstInstInitialize(). Similarly, when done with the Instrumentation Layer, function QstInstCleanup() should be invoked.

4.1.4 Function Overview

The API for the Instrumentation Layer contains a rich set of functions for enumerating sensor, obtaining information about them and obtaining health status and readings. These functions are used as follows:

- Function QstGetSensorCount() is used to enumerate the available sensors. It is invoked once for each sensor type.
- Functions QstGetSensorConfiguration(), QstGetSensorThresholdsHigh() and QstGetSensorThresholdsLow() are used to obtain information about particular sensors.
- Functions QstGetSensorHealth() and QstGetSensorReading() are used to obtain sensor health status and current readings for particular sensors.
- Functions QstSetSensorThresholdsHigh() and QstSetSensorThresholdsLow() are used to change the thresholds for particular sensors. Functions QstSensorThresholdsHighChanged() and QstSensorThresholdsLowChanged() are used to determine if another application has modified the thresholds.

The API also provides functions for health monitoring of Fan Controllers. The functions are used as follows:

- Function QstGetControllerCount() is used to enumerate the available fan controllers.
- Function QstGetControllerConfiguration() is used to obtain information about the configuration of particular controllers.
- Function QstGetControllerState() and QstGetControllerDutyCycle() are used to obtain health status and current duty cycle values for particular Fan Controllers.

The API also provides a number of maintenance functions. The functions are used as follows:

- Function QstGetPollingInterval() is used to obtain the time interval that is used to marshal requests to refresh data from the Intel® QST.
- Function QstSetPollingInterval() is used to change the Polling Interval.
- Function QstPollingIntervalChanged() is used to determine whether another application has modified the Polling Interval.

Examples showing how to use these functions is presented in Section 4.3.

4.1.5 Enumerations

The QstInst DLL utilizes a number of enumerations for variables passed into and returned from its various functions. These enumerations are declared in header file QstInst.h.

4.1.5.1 QST_SENSOR_TYPE

This enumeration provides definitions for the types of sensors that may be monitored by the Intel® QST. It is defined as follows:

```
typedef enum _QST_SENSOR_TYPE
{
    TEMPERATURE_SENSOR                = 0,
    VOLTAGE_SENSOR                    = 1,
    FAN_SPEED_SENSOR                  = 2,
    CURRENT_SENSOR                    = 3

}   QST_SENSOR_TYPE;
```

4.1.5.2 QST_FUNCTION

This enumeration provides definitions for the monitor/control functions that particular temperature sensors, voltage sensors, current sensors, fan speed sensors and fan speed controllers are responsible for. It is defined as follows:

```
typedef enum _QST_FUNCTION
{
    UNKNOWN_OTHER                               = 0,

    // Temperature Sensors

    CPU_CORE_TEMPERATURE                        = 1,
    CPU_DIE_TEMPERATURE                         = 2,
    ICH_TEMPERATURE                             = 3,
    MCH_TEMPERATURE                             = 4,
    VR_TEMPERATURE                              = 5,
    MEMORY_TEMPERATURE                          = 6,
    MOTHERBOARD_AMBIENT_TEMPERATURE             = 7,
    SYSTEM_AMBIENT_AIR_TEMPERATURE              = 8,
    CPU_INLET_AIR_TEMPERATURE                   = 9,
    SYSTEM_INLET_AIR_TEMPERATURE                = 10,
    SYSTEM_OUTLET_AIR_TEMPERATURE               = 11,
    PSU_HOTSPOT_TEMPERATURE                     = 12,
    PSU_INLET_AIR_TEMPERATURE                   = 13,
    PSU_OUTLET_AIR_TEMPERATURE                  = 14,
    DRIVE_TEMPERATURE                           = 15,
    GPU_TEMPERATURE                             = 16,
    IOH_TEMPERATURE                             = 17,
    PCH_TEMPERATURE                             = 18,

    // Voltage Sensors

    PLUS_12_VOLTS                               = 1,
    NEG_12_VOLTS                                = 2,
    PLUS_5_VOLTS                                = 3,
    PLUS_5_VOLT_BACKUP                          = 4,
    NEG_5_VOLTS                                 = 5,
    PLUS_3P3_VOLTS                              = 6,
    PLUS_2P5_VOLTS                              = 7,
    PLUS_1P5_VOLTS                              = 8,
    CPU_1_VCCP_VOLTAGE                          = 9,
    CPU_2_VCCP_VOLTAGE                          = 10,
    CPU_3_VCCP_VOLTAGE                          = 11,
    CPU_4_VCCP_VOLTAGE                          = 12,
    PSU_INPUT_VOLTAGE                           = 13,
    MCH_VCC_VOLTAGE                             = 14,
    PLUS_3P3_VOLT_STANDBY                       = 15,
    CPU_VTT_VOLTAGE                             = 16,
    PLUS_1P8_VOLTS                              = 17,
    PCH_VCC_VOLTAGE                             = 18,

    // Current Sensors

    PLUS_12_VOLT_CURRENT                        = 1,
    NEG_12_VOLT_CURRENT                         = 2,
    PLUS_5_VOLT_CURRENT                         = 3,
    PLUS_5_VOLT_BACKUP_CURRENT                  = 4,
    NEG_5_VOLT_CURRENT                          = 5,
    PLUS_3P3_VOLT_CURRENT                       = 6,
    PLUS_2P5_VOLT_CURRENT                       = 7,
    PLUS_1P5_VOLT_CURRENT                       = 8,
    CPU_1_CURRENT                               = 9,
    CPU_2_CURRENT                               = 10,
    CPU_3_CURRENT                               = 11,
    CPU_4_CURRENT                               = 12,
    PSU_INPUT_CURRENT                           = 13,
    MCH_CURRENT                                 = 14,
    PLUS_3P3_VOLT_STANDBY_CURRENT               = 15,
    PLUS_1P8_VOLT_CURRENT                       = 16,
    PCH_CURRENT                                 = 17,
```

```
    // Fan Speed Sensors and Controllers

    CPU_COOLING_FAN                         = 1,
    SYSTEM_COOLING_FAN                      = 2,
    MCH_COOLING_FAN                         = 3,
    VR_COOLING_FAN                          = 4,
    CHASSIS_COOLING_FAN                     = 5,
    CHASSIS_INLET_FAN                       = 6,
    CHASSIS_OUTLET_FAN                      = 7,
    PSU_COOLING_FAN                         = 8,
    PSU_INLET_FAN                           = 9,
    PSU_OUTLET_FAN                          = 10,
    DRIVE_COOLING_FAN                       = 11,
    GPU_COOLING_FAN                         = 12,
    AUX_COOLING_FAN                         = 13,
    IOH_COOLING_FAN                         = 14,
    PCH_COOLING_FAN                         = 15,

} QST_FUNCTION;
```

Note: That the values assigned to the different types of sensors and controllers are in overlapping arrays; the sensor/controller type is required to provide differentiation.

4.1.5.3 QST_HEALTH

This enumeration provides definitions for the health status indicators for the sensors and controllers that are managed by the Intel® QST. It is defined as follows:

```
typedef enum _QST_HEALTH
{
    HEALTH_NORMAL                   = 0,
    HEALTH_NONCRITICAL              = 1,
    HEALTH_CRITICAL                 = 2,
    HEALTH_NONRECOVERABLE           = 3

}   QST_HEALTH;
```

Value HEALTH_NORMAL indicates that a sensor's current reading falls with normal bounds. It is also used to indicate that no issues exist with the communications or operation of the Sensor or Fan Speed Controller.

Value HEALTH_NONCRITIAL indicates that a sensor's current reading has exceeded a non-critical threshold. It is also used to indicate that a non-critical problem has occurred with a Sensor or Fan Speed Controller.

Value HEALTH_CRITICAL indicates that a sensor's current reading has exceeded a critical threshold. It is also used to indicate that a critical problem has occurred with a Sensor or Fan Speed Controller.

Value HEALTH_NONRECOVERABLE indicates that a sensor's current reading has exceeded one of its non-recoverable health thresholds. It is also used to indicate that a non-recoverable problem has occurred with a Sensor or Fan Speed Controller.

4.1.5.4 FSC_CONTROL_STATE

This enumeration provides definitions for the control state of the fan speed controllers maintained by the Intel® QST. It is defined as follows:

```
typedef enum _QST_CONTROL_STATE
{
    CONTROL_NORMAL                    = 0,
    CONTROL_OVERRIDE_SOFTWARE         = 1,
    CONTROL_OVERRIDE_SENSOR_ERROR     = 2,
    CONTROL_OVERRIDE_CONTROLLER_ERROR = 3

}   QST_CONTROL_STATE;
```

Value CONTROL_NORMAL indicates that the fan speed controller is in a normal operational state and duty cycle values are being successfully controlled based upon temperature readings and Subsystem configuration.

Value CONTROL_OVERRIDE_SOFTWARE indicates that software has overridden control of fan speed determination and has set the fan speed to a specific duty cycle percentage.

Value CONTROL_OVERRIDE_SENSOR_ERROR indicates that Intel® QST has overridden the fan speed to 100% duty cycle (full speed), as a result of a persistent inability to trust readings from one or more of the Temperature Sensors. This may be caused by errors being reported by a sensor or an inability to obtain readings from a sensor. When this occurs, Intel® QST drives all fans to full speed, in order to protect the system from possible thermal overrun conditions.

Value CONTROL_OVERRIDE_CONTROLLER_ERROR indicates that Intel® QST has overridden the fan speed to 100% duty cycle (full speed), as a result of persistent problems setting the duty cycle for this or another fan speed controller. Again, this is a protection feature of the Subsystem.

4.2 Health Monitoring API Functions

4.2.1 Initialization Functions

These functions are used to manage the initialization and cleanup of the Health Monitoring functions. Their invocation is required in the DOS environment but is optional in the Windows and Linux environments (see their implementation in the above sections describing DLL & SO file dynamic loading).

4.2.1.1 QstInstInitialize()

This function is used to initialize support for the Instrumentation Layer. It should be invoked from Windows* applications that dynamically link to the QstInst DLL.

Invocation:

```
BOOL    QstInstInitialize( void );
```

Returns:

If the function succeeds, the return value is 1 (TRUE). If the function fails, the return value is zero (FALSE). From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Windows* Exception Codes:

All exception codes returned via Win32 function GetLastError() are the result of a failure occurring during the use of Win32 functions LoadLibrary() or GetProcAddress().

Linux* Exception Codes:

The *errno* variable could receive one of the following exception codes:

ENODEV	The QstInst SO File (libQstComm.so.1) or the underlying QstComm SO File (libQstInst.so.1) could not be located or loaded.
ENOENT	Could not locate the symbolic reference for one or more of the functions in the SO File.

DOS Exception Codes:

The *errno* variable could receive any of the following exception codes:

ENXIO	The IMEI interface could not be located in PCI address space. An unsupported chipset is present.
EIO	An I/O error occurred while attempting to initialize the IMEI interface.
ENOMEM	Buffer space to support communications transaction could not be allocated.

4.2.1.2 QstInstCleanup()

This function is used to free resources used by the Instrumentation Layer. It should be invoked from Windows* applications that dynamically link to the QstInst DLL.

Invocation:

```
void    QstInstCleanup( void );
```

4.2.2 Sensor Support Functions

These functions are used to manage the monitoring of Sensors contained within the system. The functions supported are:

QstGetSensorCount()
QstGetSensorConfiguration()
QstGetSensorThresholdsHigh()
QstGetSensorThresholdsLow()
QstGetSensorHealth()
QstGetSensorReading()
QstSetSensorThresholdsHigh()
QstSensorThresholdsHighChanged()
QstSetSensorThresholdsLow()
QstSensorThresholdsLowChanged()

4.2.2.1 QstGetSensorCount()

This function is used to obtain a count of the number of sensors of a particular type that are being monitored by the Intel® QST Subsystem.

Invocation:

```
BOOL APIENTRY QstGetSensorCount
(
    IN    QST_SENSOR_TYPE    SensorType,
    OUT   int                *pSensorCount
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose support count is desired.

Output Parameters:

pSensorCount	Specifies the address of the variable that will receive a count of the number of sensors of the specified type that are being monitored by the Intel® QST Subsystem.

Returns:

The function returns a Boolean indication of its success returning the requested data. The function will return FALSE if unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other

than TEMPERATURE_SENSOR, FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR.

4.2.2.2 QstGetSensorConfiguration()

This function is used to obtain configuration information for a particular sensor.

Invocation:

```
BOOL APIENTRY QstGetSensorConfiguration
(
    IN    QST_SENSOR_TYPE      SensorType,
    IN    int                  SensorIndex,
    OUT   QST_ FUNCTION        *pSensorFunction,
    OUT   BOOL                 *pRelativeReadings,
    OUT   float                *pNominalReading
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose configuration is desired.
SensorIndex	Specifies the index for the particular sensor of the specified type whose configuration information is desired.

Output Parameters:

pSensorFunction	Specifies the address of the variable that will receive an indication of the function for the sensor.
pRelativeReadings	Specifies the address of the variable that will receive an indication of whether or not the sensor returns readings in relative (as opposed to absolute) form: • For Temperature Sensors, relative readings are typically (but not necessarily) specified in Degrees Celsius; the 0 (zero) reading is typically some particular temperature value, such as, for example, the Processor's Thermal Control Circuit activation temperature. • For Voltage Sensors, relative readings are typically specified in volts (DC/AC); the 0 (zero) value is typically the nominal voltage level for the input to the voltage sensor. • For Fan Speed and Current Sensors, this field is not applicable; FALSE will be returned.
pNominalReading	Specifies the address of the variable that will receive the nominal reading for the sensor.

Returns:

The function returns a Boolean indication of its success returning the requested data. The function will return FALSE if unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than TEMPERATURE_SENSOR, FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.3 QstGetSensorThresholdsHigh()

This function is used to obtain the high (over) health thresholds for the specified Sensor. Specifically, it is used to obtain the over-voltage thresholds for Voltage Sensors, the over-current thresholds from Current Sensors and the over-temperature thresholds for Temperature Sensors. It cannot be used for Fan Speed Sensors, which only support under-speed thresholds.

Invocation:

```
BOOL APIENTRY QstGetSensorThresholdsHigh
(
    IN    QST_SENSOR_TYPE     SensorType,
    IN    int                 SensorIndex,
    OUT   float               *pThresholdNonCritical,
    OUT   float               *pThresholdCritical,
    OUT   float               *pThresholdNonRecoverable
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose high thresholds are desired.
SensorIndex	Specifies the index for the particular sensor of the specified type whose high thresholds are desired.

Output Parameters:

pThresholdNonCritical	Specifies the address of the variable that will receive the Non-Critical High Threshold for the sensor. When readings are above this threshold, the health status for the sensor will be set to HEALTH_NONCRITICAL.
pThresholdCritical	Specifies the address of the variable that will receive the Critical High Threshold for the sensor. When readings are above this threshold, the health status for the sensor will be set to HEALTH_CRITICAL.
pThresholdNonRecoverable	Specifies the address of the variable that will receive the Non-Recoverable High Threshold for the sensor. When readings are above this threshold, the health status for the sensor will be set to HEALTH_NONECOVERABLE.

Returns:

The function returns a Boolean indication of its success returning the requested data. The function will return FALSE if unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than TEMPERATURE_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.4 QstGetSensorThresholdsLow()

This function is used to obtain the low (under) health thresholds for the specified Sensor. Specifically, it is used to obtain under-voltage thresholds for Voltage Sensors, under-current thresholds for Current Sensors and under-speed thresholds for Fan Speed Sensors. It cannot be used for Temperature Sensors, which only support over-temperature thresholds.

Invocation:

```
BOOL APIENTRY QstGetSensorThresholdsLow
(
    IN    QST_SENSOR_TYPE      SensorType,
    IN    int                  SensorIndex,
```

```
    OUT   float                 *pThresholdNonCritical,
    OUT   float                 *pThresholdCritical,
    OUT   float                 *pThresholdNonRecoverable
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose low thresholds are desired.
SensorIndex	Specifies the index for the particular sensor of the specified type whose low thresholds are desired.

Output Parameters:

pThresholdNonCritical	Specifies the address of the variable that will receive the Non-Critical Low Threshold for the sensor. When readings are below this threshold, the health status for the sensor will be set to HEALTH_NONCRITICAL.
pThresholdCritical	Specifies the address of the variable that will receive the Critical Low Threshold for the sensor. When readings are below this threshold, the health status for the sensor will be set to HEALTH_CRITICAL.
pThresholdNonRecoverable	Specifies the address of the variable that will receive the Non-Recoverable Low Threshold for the sensor. When readings are below this threshold, the health status for the sensor will be set to HEALTH_NONRECOVERABLE.

Returns:

The function returns a Boolean indication of its success obtaining the requested data. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.5 QstGetSensorHealth()

This function is used to obtain the current health of the specified sensor.

Invocation:

```
BOOL APIENTRY QstGetSensorHealth
(
    IN    QST_SENSOR_TYPE       SensorType,
```

```
    IN     int                 SensorIndex,
    OUT    QST_HEALTH          *pSensorHealth,
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose low thresholds are desired.
SensorIndex	Specifies the index for the particular sensor of the specified type whose current health are desired.

Output Parameters:

pSensorHealth	Specifies the address of the variable that will receive a health indication for the specified sensor.

Returns:

The function returns a Boolean indication of its success obtaining the requested data. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than TEMPERATURE_SENSOR, FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.6 QstGetSensorReading()

This function is used to obtain the current reading for the specified sensor.

Invocation:

```
BOOL APIENTRY QstGetSensorReading
(
    IN    QST_SENSOR_TYPE     SensorType,
    IN    int                 SensorIndex,
    OUT   float               *pSensorReading
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose current reading is desired.
SensorIndex	Specifies the index for the particular sensor of the specified type whose current reading is desired.

Output Parameters:

pSensorReading	Specifies the address of the variable that will receive the current reading from the specified sensor.

Returns:

The function returns a Boolean indication of its success obtaining the requested data. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than TEMPERATURE_SENSOR, FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.7 QstSetSensorThresholdsHigh()

This function is used to set the high (over) health thresholds for the specified Sensor. Specifically, it is used to set over-voltage thresholds for Voltage Sensors and over-temperature thresholds for Temperature Sensors. It cannot be used for Fan Speed Sensors, which only support under-speed thresholds.

Invocation:

```
BOOL APIENTRY QstSetSensorThresholdsHigh
(
    IN    QST_SENSOR_TYPE      SensorType,
    IN    int                  SensorIndex,
    IN    float                ThresholdNonCritical,
    IN    float                ThresholdCritical,
    IN    float                ThresholdNonRecoverable
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose high thresholds are to be set.
SensorIndex	Specifies the index for the particular sensor of the specified type whose high thresholds are to be set.
ThresholdNonCritical	Specifies the Non-Critical High Threshold for the sensor. When readings are above this threshold, the health status for the sensor will be set to HEALTH_NONCRITICAL.
ThresholdCritical	Specifies the Critical High Threshold for the sensor. When readings are above this threshold, the health status for the sensor will be set to HEALTH_CRITICAL.
ThresholdNonRecoverable	Specifies the Non-Recoverable High Threshold for the sensor. When readings are above this threshold, the health status for the sensor will be set to HEALTH_NONRECOVERABLE.

Returns:

The function returns a Boolean indication of its success applying the requested changes. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than TEMPERATURE_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.8 QstSensorThresholdsHighChanged()

This function is used by an application to ascertain whether or not a sensor's high (over) thresholds have been changed, by this or another application, since the time that this application last checked. Specifically, it is used to check for changes being made to over-voltage thresholds for Voltage Sensors and over-temperature thresholds for Temperature Sensors. It cannot be used for Fan Speed Sensors, which only support under-speed thresholds.

Invocation:

```
BOOL APIENTRY QstSensorThresholdsHighChanged
(
    IN    QST_SENSOR_TYPE     SensorType,
    IN    int                 SensorIndex,
    IN    time_t              LastUpdate,
    OUT   BOOL                *pUpdated
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose high thresholds are to be checked.
SensorIndex	Specifies the index for the particular sensor of the specified type who's High Thresholds are to be checked.
LastUpdate	Specifies the time that the application last checked for changes to the specified sensor's High Thresholds.

Output Parameters:

pUpdated	Specifies the address of the variable that will receive a Boolean indication of whether or not the Sensor's thresholds have been changed since the time specified by the invoker.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for sensor types other than TEMPERATURE_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.9 QstSetSensorThresholdsLow()

This function is used to set the low (under) health thresholds for the specified Sensor. Specifically, it is used to set the under-voltage thresholds for Voltage Sensors and under-speed thresholds for Fan Speed Sensors. It cannot be used for Temperature Sensors, which only support over-temperature thresholds.

Invocation:

```
BOOL APIENTRY QstSetSensorThresholdsLow
(
    IN    QST_SENSOR_TYPE      SensorType,
    IN    int                  SensorIndex,
    IN    float                ThresholdNonCritical,
    IN    float                ThresholdCritical,
    IN    float                ThresholdNonRecoverable
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose low thresholds are to be set.
SensorIndex	Specifies the index for the particular sensor of the specified type whose low thresholds are to be set.
ThresholdNonCritical	Specifies the Non-Critical Low Threshold for the sensor. When readings are below this threshold, the health status for the sensor will be set to HEALTH_NONCRITICAL.
ThresholdCritical	Specifies the Critical Low Threshold for the sensor. When readings are below this threshold, the health status for the sensor will be set to HEALTH_CRITICAL.
ThresholdNonRecoverable	Specifies the Non-Recoverable Low Threshold for the sensor. When readings are below this threshold, the health status for the sensor will be set to HEALTH_NONRECOVERABLE.

Returns:

The function returns a Boolean indication of its success applying the requested changes. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for any sensor type other than FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.2.10 QstSensorThresholdsLowChanged()

This function is used by an application to ascertain whether or not a sensor's low thresholds have been changed, by this or another application, since the time that this application last checked. Specifically, it is used to check for changes being made to under-voltage thresholds for Voltage Sensors and under-speed thresholds for Fan Speed Sensors. It cannot be used for Temperature Sensors, which only support over-temperature thresholds.

Invocation:

```
BOOL APIENTRY QstSensorThresholdsLowChanged
(
    IN    QST_SENSOR_TYPE     SensorType,
    IN    int                 SensorIndex,
    IN    time_t              LastUpdate,
    OUT   BOOL                *pUpdated
);
```

Input Parameters:

SensorType	Specifies the type of sensor whose low thresholds are to be checked.
SensorIndex	Specifies the index for the particular sensor of the specified type who's Low Thresholds are to be checked.
LastUpdate	Specifies the time that the application last checked for changes to the specified sensor's Low Thresholds.

Output Parameters:

pUpdated	Specifies the address of the variable that will receive a Boolean indication of whether or not the Sensor's thresholds have been changed since the time specified by the invoker.

Returns:

The function returns a Boolean indication of its success applying the requested changes. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for any sensor type other than FAN_SPEED_SENSOR, CURRENT_SENSOR or VOLTAGE_SENSOR or for invalid sensor indexes.

4.2.3 Fan Controller Support Functions

These functions are used to manage the monitoring of the Fan Controllers contained within the system. The functions supported are:

QstGetControllerCount()
QstGetControllerConfiguration()
QstGetControllerState()
QstGetControllerDutyCycle()

4.2.3.1 QstGetControllerCount()

This function is used to obtain a count of the number of Fan Controllers being managed by the Intel® QST Subsystem.

Invocation:

```
BOOL APIENTRY QstGetControllerCount
(
    OUT   int                 *pControllerCount
);
```

Output Parameters:

pControllerCount	Specifies the address of the variable that will receive a count of the controllers being managed.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

4.2.3.2 QstGetControllerConfiguration()

This function is used to obtain the configuration of the specified Fan Controller.

Invocation:

```
BOOL APIENTRY QstGetControllerConfiguration
(
    IN    int                 ControllerIndex,
    OUT   QST_FUNCTION        *pControllerFunction
);
```

Input Parameters:

ControllerIndex	Specifies the index for the particular Fan Controller whose configuration is desired.

Output Parameters:

pControllerFunction	Specifies the address of the variable that is to receive an indication of the function of the controller.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for invalid controller indexes.

4.2.3.3 QstGetControllerState()

This function is used to obtain the current state of the specified Fan Controller.

Invocation:

```
BOOL APIENTRY QstGetControllerState
(
    IN    int                 ControllerIndex,
    OUT   QST_HEALTH          *pHealthState,
    OUT   QST_CONTROL_STATE   *pControlState
);
```

Input Parameters:

ControllerIndex	Specifies the index for the particular sensor of the specified type whose current health are desired.

Output Parameters:

pHealthState	Specifies the address of the variable that will receive the health state of the specified Fan Controller.
pControlState	Specifies the address of the variable that will receive the control state of the specified Fan Controller.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error

information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for invalid controller indexes.

4.2.3.4 QstGetControllerDutyCycle()

This function is used to obtain the current duty cycle from the specified Fan Controller.

Invocation:

```
BOOL APIENTRY QstGetControllerDutyCycle
(
    IN     int              ControllerIndex,
    OUT    float            *pControllerDuty
);
```

Input Parameters:

ControllerIndex	Specifies the index for the particular Fan Controller whose current duty Cycle is desired.

Output Parameters:

pControllerDuty	Specifies the address of the variable that will receive the current duty cycle from the specified Fan Controller.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

Note:

WIN32 Error Code ERROR_INVALID_PARAMETER or errno value EINVAL will be received if an attempt is made to execute this function for invalid controller indexes.

4.2.4 DLL Management Functions

These functions are used to manage the overall operation of the DLL. The functions supported are:

QstGetPollingInterval()
QstSetPollingInterval()
QstPollingIntervalChanged()

4.2.4.1 QstGetPollingInterval()

This function is used to obtain the polling interval that is being used by the DLL. This is the maximum rate at which the DLL will perform polls of individual sensor/controller information from the Intel® QST Subsystem, on behalf of all applications utilizing the DLL.

Invocation:

```
BOOL APIENTRY QstGetPollingInterval
(
    OUT   DWORD          *pPollingInterval
);
```

Output Parameters:

pPollingInterval	Specifies the address of the variable that will receive the current polling interval, measured in milliseconds

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

4.2.4.2 QstSetPollingInterval()

This function is used to set the polling interval to be used by the DLL. This is the maximum rate at which the DLL will perform polls of individual sensor/controller information from the ME, on behalf of all applications utilizing the DLL.

Invocation:

```
BOOL APIENTRY QstSetPollingInterval
(
    IN    DWORD          PollingInterval
);
```

Input Parameters:

PollingInterval	Specifies the polling interval, specified in milliseconds, that is to be used by the DLL.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

4.2.4.3 QstPollingIntervalChanged()

This function is used to determine whether or not the polling interval has been changed by another thread or process since the last time that this process performed this check.

Invocation:

```
BOOL APIENTRY QstPollingIntervalChanged
(
    IN    QST_TIME_T    LastUpdate,
    OUT   BOOL          *pUpdated
);
```

Input Parameters:

LastUpdate	Specifies the time that the application last checked for the Polling Interval being changed.

Output Parameters:

pUpdated	Specifies the address of the variable that will receive a Boolean indication of whether or not the Polling Interval has been changed since the time specified by the invoker.

Returns:

The function returns a Boolean indication of its success obtaining the requested information. The function will return FALSE if it was unsuccessful. From Windows* applications, WIN32 function GetLastError() can be used to obtain extended error information. From DOS applications, extended error information is provided via the *errno* variable.

4.3 Using Health Monitoring Functions

4.3.1 Exposing Sensor/Controller Information

In the previous chapter, example code was presented that showed how one may communicate directly with Intel® QST and expose information about the sensors/controllers present. We contrast this with the example presented below which, though its use of the services of the Instrumentation Layer to do the heavy lifting, is far less complex.

Note: Support for Temperature Monitors is shown; similar code can be crafted to support other Monitor types, as well as Fan Controllers.

```
/*****************************************************************************/
/* Variables                                                                 */
/*****************************************************************************/

static int iTempCount;

/*****************************************************************************/
/* GetTempCountInst() - Returns number of temperature sensors                */
/*****************************************************************************/

int GetTempCountInst( void )
{
   if( QstGetSensorCount( TEMPERATURE_SENSOR, &iTempCount ) )
      return( iTempCount );
   else
      return( -1 );
}

/*****************************************************************************/
/* GetTempUsageInst() - Returns usage indicator for temperature sensor       */
/* TBD: Provide support for exposing Nominal and Relative Temperature info   */
/*****************************************************************************/

int GetTempUsageInst( int iTempIndex )
{
   QST_FUNCTION    eSensorFunction;
   BOOL             bRelativeReadings;
   float            fNominalReading;

   if( QstGetSensorConfiguration( TEMPERATURE_SENSOR, iTempIndex,
                     &eSensorFunction, &bRelativeReadings, &fNominalReading ) )
      return( (int)eSensorFunction );
   else
      return( -1 );
}

/*****************************************************************************/
/* GetTempReadingInst() - Returns reading from temperature sensor            */
/*****************************************************************************/

float GetTempReadingInst( int iTempIndex )
{
   float fReading;

   if( QstGetSensorReading( TEMPERATURE_SENSOR, iTempIndex, &fReading ) )
      return( fReading );
   else
      return( -1 );
}
```

```
/*****************************************************************************/
/* GetTempHealthInst() - Returns health indicator for temperature sensor     */
/*****************************************************************************/

int GetTempHealthInst( int iTempIndex )
{
   QST_HEALTH eHealth;

   if( QstGetSensorHealth( TEMPERATURE_SENSOR, iTempIndex, &eHealth ) )
      return( (int)eHealth );
   else
      return( -1 );

}

/*****************************************************************************/
/* GetTempThreshInst() - Returns health thresholds for temperature sensor    */
/*****************************************************************************/

BOOL GetTempThreshInst( int iTempIndex, float *pfNonCrit,
                        Float *pfCrit, float *pfNonRecov          )
{
   return( QstGetSensorThresholdsHigh( TEMPERATURE_SENSOR, iTempIndex,
                                       pfNonCrit, pfCrit, pfNonRecov ) );
}
```

4.3.2 Displaying Sensor/Controller Readings

The following example demonstrates how one can regularly (once per second) display readings from the available temperature sensors, fan speed sensors and fan speed controllers, until such time as a key is pressed. It is a much simplified version of the QstLog Tool.

```
int main( int iArgs, char *pszArg[] )
{
   float     fReading;
   unsigned  uCount;
   int       iIndex, iTempSensors, iFanSensors, iFanControllers;

#if
defined(DYNAMIC_DLL_LOADING)||defined(_DOS)||defined(__DOS__)||defined(MSDOS)

   // Initialize the Instrumentation Layer

   if( !QstInstInitialize() )
   {
      puts( “Unable to initialize QstInst” );
      return( 1 );
   }

#endif

   // Enumerate Sensors/Controllers

   QstGetSensorCount( TEMPERATURE_SENSOR, &iTempSensors );
   QstGetSensorCount( FAN_SPEED_SENSOR, &iFanSensors );
   QstGetControllerCount( &iFanControllers );

   // Loop outputting Readings

   for( uCount = 1; !kbhit(); ++uCount )
   {
      printf( “%d”, uCount );

      // Display Temperature(s)

      for( iIndex = 0; iIndex < iTempSensors; iIndex++ )
      {
         QstGetSensorReading( TEMPERATURE_SENSOR, iIndex, &fReading )
```

```
            printf( ",%.2f", fReading );
         }

         // Display Fan Speed(s)

         for( iIndex = 0; iIndex < iFanSensors; iIndex++ )
         {
            QstGetSensorReading( FAN_SPEED_SENSOR, iIndex, &fReading )
            printf( ",%d", (int)GetFanReadingInst( iIndex ));
         }

         // Display Duty Cycle(s)

         for( iIndex = 0; iIndex < iFanControllers; iIndex++ )
         {
            QstGetControllerDutyCycle( iIndex, &fReading );
            printf( ",%.2f", fReading );
         }

         // End the line

         putchar( '\n' );
         fflush( stdout );

         // Wait a second

         Sleep( 1000 );
      }

   #if
defined(DYNAMIC_DLL_LOADING)||defined(_DOS)||defined(__DOS__)||defined(MSDOS)
      QstInstCleanup();
   #endif

      return( 0 );
   }
```

§

5 Querying and Controlling Hardware

5.1 Introduction

In order to provide a consistent interface for Sensor/Controller hardware access, Intel® QST encapsulates the system's embedded (chipset and processor package) sensors and fan speed controllers with SST Commands. These sensors and fan speed controllers will be presented as a single logical SST device. A special reserved SST Bus Address, 20h, will be used to access this logical device.

The SST Bus Command Protocol requires the specification of a minimum of four byte-wide fields in command packets. These fields specify the (SST) Device Address, Command Code, the amount of command data included and the amount of response data desired.

This chapter details the commands that will be used to access the chipset-based Temperature and Fan Speed Sensors and Fan Speed (PWM) Controller hardware. These commands will support access to this hardware from the Intel® QST firmware, as well as from Host Processor-based software (via the SST Pass-Through interface; see Section 3.3.10 for more details).

Note: Access to sensors and fan controller devices at this level is not normally required. Typically, only knowledge of the command bytes used for the chipset sensors and fan controllers is required (for configuration creation purposes).

5.2 Temperature Sensor Access

A single temperature sensor is embedded within the chipset (the PCH device). This sensor is read-only from the standpoint of the Intel® Management Engine (ME); its initialization will be performed by the system BIOS. Consequently, no functionality will be required to utilize this sensor other than a Temperature-Read command.

The chipset also provides access to the other temperature sensors that are accessible through its services. This includes the processor's package temperature, which is an amalgam of the readings from the Digital Thermal Sensors (DTS) embedded within each of the processor's Cores and Uncore. This temperature is obtained via the Platform Environment Control Interface (PECI). The chipset may provide access to an MCH temperature, if indeed the processor's MCH contains a DTS. As well, in some circumstances, it may also provide access to temperature sensors in the system's memory DIMMs.

5.2.1 Requirements

5.2.1.1 Temperature Data Format

SST Temperature Sensors return temperature readings in the standard format for SST Bus-based Temperature Sensors. This format is defined as follows:

Figure 1: Temperature Data Format

MSB Upper nibble					MSB Lower nibble					LSB Upper nibble					LSB Lower nibble			
S	x	x	x		x	x	x	x		x	x	x	x		x	x	x	x
Sign	Integer Value (0-511 °C)											Fractional Value (~0.016 °C)						

The format for temperature readings is structured to allow values in the range +/- 512 °C to be represented. Temperature sensor data is returned as a signed, two's-complement 16-bit binary value. It represents the number of 1/64 °C increments in the actual reading. This allows the representation of temperatures with approximately a 0.016 °C resolution. Example representations are provided in the following table:

Table 91: Temperature Sensor Data Examples

Temperature	2's complement representation
80 °C	0001 0100 0000 0000
79.875 °C	0001 0011 1111 1000
1 °C	0000 0000 0100 0000
0 °C	0000 0000 0000 0000
-1 °C	1111 1111 1100 0000
-5 °C	1111 1110 1100 0000

Values that would represent temperatures below -273.15 ºC (0 °K or Absolute Zero) are reserved and must not be returned except as specifically noted.

5.2.1.2 Error Reporting

SST Bus-based Temperature Sensors provide no commands allowing problems with Temperature Sensors to be reported. Consequently, problems will be reported using special encodings within temperature values read from the sensors. The special encodings currently defined are detailed as follows:

- 8000h General sensor error; no more details are available.
- 8001h External thermal diode (or other sensing device) is missing or has failed.
- 8002h Sensor is working but has detected a temperature that is below its capabilities to represent.
- 8003h Sensor is working but has detected a temperature that is above its capabilities to represent.

5.2.2 Commands Supported

A single command will be supported for accessing temperature readings from the chipset temperature sensors (digital thermometers):

GetCSTempSensor#Reading

5.2.2.1 GetCSTempSensor#Reading

These commands are used to obtain readings from a particular temperature sensor.

Table 92: GetCSTempSensor#Reading SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	1
02h	Response Data Size	2
03h	Command Code	00h-0Dh

Table 93: GetCSTempSensor#Reading SST Response Packet

Byte Offset	Description
00-01h	Temperature Reading

The Command Codes reserved for chipset temperature sensors are detailed in the following table:

Table 94: GetCSTempSensor#Reading SST Command Codes

Command Code	Sensor	Currently Supported?
00h	PCH Temperature	Yes
01h	MCH Temperature	Yes
02h	CPU 1 Temperature	Yes
03h	CPU 2 Temperature	If Available
04h	CPU 3 Temperature	No
05h	CPU 4 Temperature	No
06h	DIMM 1 Temperature	If Available
07h	DIMM 2 Temperature	If Available
08h	DIMM 3 Temperature	If Available
09h	DIMM 4 Temperature	If Available
0Ah	DIMM 5 Temperature	No
0Bh	DIMM 6 Temperature	No
0Ch	DIMM 7 Temperature	No
0Dh	DIMM 8 Temperature	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_GET_PCH_TEMP_READING         0x00
#define SST_CS_GET_MCH_TEMP_READING         0x01
#define SST_CS_GET_CPU_1_TEMP_READING       0x02
#define SST_CS_GET_CPU_2_TEMP_READING       0x03
#define SST_CS_GET_CPU_3_TEMP_READING       0x04    // currently not supported
#define SST_CS_GET_CPU_4_TEMP_READING       0x05    // currently not supported

#define SST_CS_GET_DIMM_1_TEMP_READING      0x06
#define SST_CS_GET_DIMM_2_TEMP_READING      0x07
#define SST_CS_GET_DIMM_3_TEMP_READING      0x08
#define SST_CS_GET_DIMM_4_TEMP_READING      0x09
#define SST_CS_GET_DIMM_5_TEMP_READING      0x0A    // currently not supported
#define SST_CS_GET_DIMM_6_TEMP_READING      0x0B    // currently not supported
#define SST_CS_GET_DIMM_7_TEMP_READING      0x0C    // currently not supported
#define SST_CS_GET_DIMM_8_TEMP_READING      0x0D    // currently not supported
```

5.3 Fan Speed Sensor Access

5.3.1 Requirements

5.3.1.1 Data Formats

Fan Speed sensors will exist which report fan speed in both RPMs and in Clock Pulse Counts. In the latter case, the clock frequency will also be reported, in order to facilitate RPM calculation. In either case, a 16-bit quantity will be required to report the result.

5.3.2 Commands Supported

Three commands will be supported for accessing fan speed sensors in the chipset:

GetCSFanSensor#Attributes
SetCSFanSensor#Configuration
GetCSFanSensor#Speed

5.3.2.1 GetCSFanSensor#Attributes

These commands are used to obtain attribute information from the available fan speed sensors.

Table 95: GetCSFanSensor#Attributes SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	1
02h	Response Data Size	2
03h	Command Code	0Eh-15h

Table 96: GetCSFanSensor#Attributes SST Response Packet

Byte Offset	Description
00h-01h	Fan Sensor Attributes (see 0)

The Command Codes reserved for chipset fan speed sensors are detailed in the following table:

Table 97: GetCSFanSensor#Attributes SST Command Codes

Command Code	Sensor	Currently Supported?
0Eh	PCH Fan Speed Sensor 1	Yes
0Fh	PCH Fan Speed Sensor 2	Yes
10h	PCH Fan Speed Sensor 3	Yes
11h	PCH Fan Speed Sensor 4	Yes
12h	PCH Fan Speed Sensor 5	No
13h	PCH Fan Speed Sensor 6	No
14h	PCH Fan Speed Sensor 7	No
15h	PCH Fan Speed Sensor 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_GET_FAN_SENS_1_ATTRIBS       0x0E
#define SST_CS_GET_FAN_SENS_2_ATTRIBS       0x0F
#define SST_CS_GET_FAN_SENS_3_ATTRIBS       0x10
#define SST_CS_GET_FAN_SENS_4_ATTRIBS       0x11
#define SST_CS_GET_FAN_SENS_5_ATTRIBS       0x12    // currently not supported
#define SST_CS_GET_FAN_SENS_6_ATTRIBS       0x13    // currently not supported
#define SST_CS_GET_FAN_SENS_7_ATTRIBS       0x14    // currently not supported
#define SST_CS_GET_FAN_SENS_8_ATTRIBS       0x15    // currently not supported
```

5.3.2.1.1 Fan Sensor Attributes

This field provides information about the capabilities of the Fan Speed Sensor.

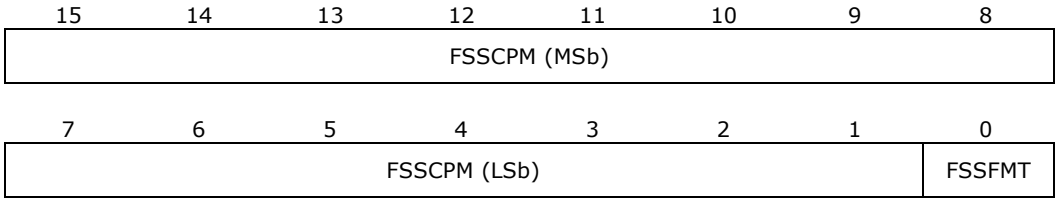


Table 98: Fan Sensor Attributes Content

Bit	Description
15:1	**FSSCPM – Clock Pulse Multiplier** – These bits indicate the resolution of the clock oscillator, as a multiple of 2.5 KHz. Fan speed in RPMs can be calculated using formula: RPMs = (FSSCPM * 2500 * 60) / counts.
0	**FSSFMT – Sensor Reading Format** – This bit, if set (1), indicates that the sensor returns a count of the clock pulses per measured revolution. In this case, field **FSSCPM** provides the resolution of the clock pulses. If cleared (0), indicates that the sensor returns a fan speed measured in RPMs. In this case, field **FSSCPM** is not utilized.

5.3.2.2 SetCSFanSensor#Configuration

These commands are used to set the configuration of the available fan speed sensors.

Table 99: SetCSFanSensor#Configuration SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	3
02h	Response Data Size	0
03h	Command Code	16h-1Dh
04h-05h	Fan Sensor Configuration	See 0

The Command Codes reserved for chipset fan speed sensors are detailed in the following table:

Table 100: SetCSFanSensor#Configuration SST Command Codes

Command Code	Sensor	Currently Supported?
16h	PCH Fan Speed Sensor 1	Yes
17h	PCH Fan Speed Sensor 2	Yes
16h	PCH Fan Speed Sensor 3	Yes
19h	PCH Fan Speed Sensor 4	Yes
1Ah	PCH Fan Speed Sensor 5	No
1Bh	PCH Fan Speed Sensor 6	No
1Ch	PCH Fan Speed Sensor 7	No
1Dh	PCH Fan Speed Sensor 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_SET_FAN_SENS_1_CONFIG        0x16
#define SST_CS_SET_FAN_SENS_2_CONFIG        0x17
```

```
#define SST_CS_SET_FAN_SENS_3_CONFIG        0x18
#define SST_CS_SET_FAN_SENS_4_CONFIG        0x19
#define SST_CS_SET_FAN_SENS_5_CONFIG        0x1A    // currently not supported
#define SST_CS_SET_FAN_SENS_6_CONFIG        0x1B    // currently not supported
#define SST_CS_SET_FAN_SENS_7_CONFIG        0x1C    // currently not supported
#define SST_CS_SET_FAN_SENS_8_CONFIG        0x1D    // currently not supported
```

5.3.2.2.1 Fan Sensor Configuration

This field specifies the configuration of the fan speed sensor.

15	14	13	12	11	10	9	8
FSSENA	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

7	6	5	4	3	2	1	0
Reserved	Reserved	FSSAFC		FSSISM	FSSPPR		

Table 101: Fan Sensor Configuration Content

Bit	Description
15	**FSSENA – Enable Controller.** This bit, if set (1), indicates that the Fan Speed Sensor is to be enabled. In this case, the configuration specified in the remainder of the field is to be applied. If reset (0), this bit indicates that the Fan Speed Sensor is to be disabled. In this case, the configuration specified in the remainder of the field will be ignored. Note: If an attempt is made to obtain a reading from a Fan Speed Sensor that has been disabled, the results are undefined.
14:6	**Reserved.**
5:4	**FSSAFC – Associated Fan Speed Controller.** Specifies the index of the local Fan Speed Controller whose output signal is connected to the fan providing this sensor's tachometer input signal.
3	**FSSISM – Independent Speed Measurement.** This bit, if cleared (0), indicates that fan speed measurements may be performed independent of the state of any associated Fan Speed Controller. If set (1), this bit indicates that the state of the associated Fan Speed Controller must be taken into account. This is the case specifically for 3-wire fans that are being controlled using the pulsed power methodology. Many fans are incapable of returning tachometer pulses while not receiving power. Extraordinary measures must be taken to accurately measure fan speed.
2:0	**FSSPPR – Pulses Per Revolution** – Specifies the number of tachometer pulses that the fan produces per revolution: 000 – 1 Pulse 001 – 2 Pulses (default) 010 – 3 Pulses 011 – 4 Pulses

5.3.2.3 GetCSFanSensor#Speed

These commands are used to get readings from the available fan speed sensors.

Table 102: GetCSFanSensor#Speed SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	1
02h	Response Data Size	2
03h	Command Code	1Eh-25h

Table 103: GetCSFanSensor#Speed SST Response Packet

Byte Offset	Description
00h-01h	Fan Speed Value (see 0)

The Command Codes reserved for chipset fan speed sensors are detailed in the following table:

Table 104: GetCSFanSensor#Speed SST Command Codes

Command Code	Sensor	Currently Supported?
1Eh	PCH Fan Speed Sensor 1	Yes
1Fh	PCH Fan Speed Sensor 2	Yes
20h	PCH Fan Speed Sensor 3	Yes
21h	PCH Fan Speed Sensor 4	Yes
22h	PCH Fan Speed Sensor 5	No
23h	PCH Fan Speed Sensor 6	No
24h	PCH Fan Speed Sensor 7	No
25h	PCH Fan Speed Sensor 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_GET_FAN_SENS_1_SPEED         0x1E
#define SST_CS_GET_FAN_SENS_2_SPEED         0x1F
#define SST_CS_GET_FAN_SENS_3_SPEED         0x20
#define SST_CS_GET_FAN_SENS_4_SPEED         0x21
#define SST_CS_GET_FAN_SENS_5_SPEED         0x22    // currently not supported
#define SST_CS_GET_FAN_SENS_6_SPEED         0x23    // currently not supported
#define SST_CS_GET_FAN_SENS_7_SPEED         0x24    // currently not supported
#define SST_CS_GET_FAN_SENS_8_SPEED         0x25    // currently not supported
```

5.3.2.3.1 Fan Speed Value

Depending upon the attribute data received via the GetCSFanSensor#Attributes command, the fan speed value returned will be either a fan speed measured in RPMs or a count of the number of clock pulses measured per revolution of the fan.

In those cases where fan speed is reported in clock pulse counts, fan speed in RPMs can be calculated using formula "RPMs = (FSSCPM * 2500 * 60) / counts". Special value 0FFFFh will be returned to indicate that the fan is not spinning (has stalled or been stopped) or that the tachometer input is not connected to a valid signal.

5.4 Fan Speed Controller Access

This section defines the command set used for Fan Speed Controller access. Data Formats.

5.4.1 Commands Supported

Three commands will be supported for accessing fan speed controllers in the chipset:

GetFanController#Attributes
SetFanController#Configuration
SetFanController#Speed
GetFanController#Speed

5.4.1.1 GetCSFanController#Attributes

These commands are used to get attribute information from the available Fan Speed Controllers.

Table 105: GetCSFanController#Attributes SST Command Data

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	1
02h	Response Data Size	2
03h	Command Code	26h-2Dh

Table 106: GetCSFanController#Attributes SST Response Data

Byte Offset	Description
00h-01h	Fan Speed Controller Attributes (see 0)

The Command Codes reserved for chipset fan speed controllers are detailed in the following table:

Table 107: GetCSFanController#Attributes SST Command Codes

Command Code	Sensor	Currently Supported?
26h	PCH Fan Speed Controller 1	Yes
27h	PCH Fan Speed Controller 2	Yes
28h	PCH Fan Speed Controller 3	Yes
29h	PCH Fan Speed Controller 4	Yes
2Ah	PCH Fan Speed Controller 5	No
2Bh	PCH Fan Speed Controller 6	No
2Ch	PCH Fan Speed Controller 7	No
2Dh	PCH Fan Speed Controller 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_GET_FAN_CTRL_1_ATTRIBS       0x26
#define SST_CS_GET_FAN_CTRL_2_ATTRIBS       0x27
#define SST_CS_GET_FAN_CTRL_3_ATTRIBS       0x28
#define SST_CS_GET_FAN_CTRL_4_ATTRIBS       0x29
#define SST_CS_GET_FAN_CTRL_5_ATTRIBS       0x2A    // currently not supported
#define SST_CS_GET_FAN_CTRL_6_ATTRIBS       0x2B    // currently not supported
#define SST_CS_GET_FAN_CTRL_7_ATTRIBS       0x2C    // currently not supported
#define SST_CS_GET_FAN_CTRL_8_ATTRIBS       0x2D    // currently not supported
```

5.4.1.1.1 Fan Controller Attributes

This field provides the attributes for the Fan Controller:

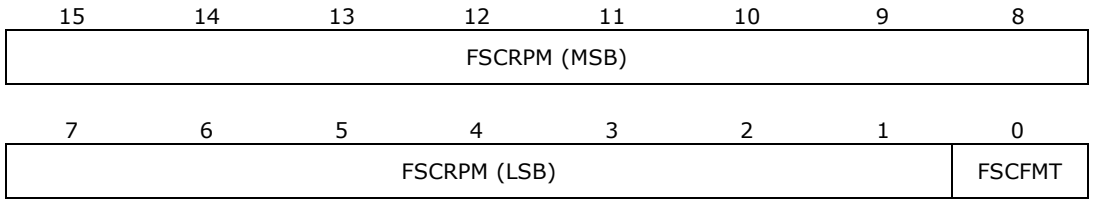


Table 108: Fan Controller Attributes Content

Bit	Description
15:1	**FSCRPM – Maximum Fan RPM** – Specifies the maximum speed of the fan, in RPMs. If the maximum speed of the fan in unknown, value 0 will be specified. Bits 1-15 of the speed are provided (i.e. Max = FSCRPM * 2). This field is ignored if bit FSCFMT is set (specifying use of duty cycle percentages for fan speed control).
0	**FSCFMT – Fan Speed Control Format** – This bit, if set (1), indicates that fan speed will be controlled using Duty Cycle Percentages. If cleared (0), this bit indicates that fan speed will be controlled using RPM values.

5.4.1.2 SetCSFanController#Configuration

These commands are used to set the configuration of the available Fan Speed Controllers.

Table 109: SetCSFanController#Configuration SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	3
02h	Response Data Size	0
03h	Command Code	2Eh-35h
04h-05h	Fan Controller Configuration	See 0

The Command Codes reserved for chipset fan speed controllers are detailed in the following table:

Table 110: SetCSFanController#Configuration SST Command Codes

Command Code	Sensor	Currently Supported?
2Eh	PCH Fan Speed Controller 1	Yes
2Fh	PCH Fan Speed Controller 2	Yes
30h	PCH Fan Speed Controller 3	Yes
31h	PCH Fan Speed Controller 4	Yes
32h	PCH Fan Speed Controller 5	No
33h	PCH Fan Speed Controller 6	No
34h	PCH Fan Speed Controller 7	No
35h	PCH Fan Speed Controller 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_SET_FAN_CTRL_1_CONFIG        0x2E
#define SST_CS_SET_FAN_CTRL_2_CONFIG        0x2F
#define SST_CS_SET_FAN_CTRL_3_CONFIG        0x30
#define SST_CS_SET_FAN_CTRL_4_CONFIG        0x31
#define SST_CS_SET_FAN_CTRL_5_CONFIG        0x32    // currently not supported
#define SST_CS_SET_FAN_CTRL_6_CONFIG        0x33    // currently not supported
#define SST_CS_SET_FAN_CTRL_7_CONFIG        0x34    // currently not supported
#define SST_CS_SET_FAN_CTRL_8_CONFIG        0x35    // currently not supported
```

5.4.1.2.1 Fan Controller Configuration

This field specifies the configuration of the fan speed controller.

15	14	13	12	11	10	9	8
CARENA	Reserved	Reserved	Reserved	Reserved	CARWTI		

7	6	5	4	3	2	1	0
CARTPE	CARPSI	CARFST			CARPSF		

Table 111: Fan Speed Configuration Contents

Bit	Description
15	**CARENA –Controller Enable.** This bit, if set (1), indicates that the Fan Speed Controller is to be enabled. In this case, the configuration specified in the remainder of the field is to be applied. If reset (0), this bit indicates that the Fan Speed Controller is to be disabled. In this case, the configuration specified in the remainder of the field can be ignored. Note: If an attempt is made to get/set the current duty cycle of a Fan Speed Controller that has been disabled, the results are undefined.
14:11	**Reserved.**
10:8	**CARWTI – Watchdog Timer Interval.** Specifies the timeout interval for the Watchdog Timer. Possible values for this field are: 000 - 0 seconds (disabled) 001 - 2 seconds 010 - 4 seconds 011 - 8 seconds 101 - 16 seconds 111 - 32 seconds ***Note:*** The timer is reset whenever the Desired Duty Cycle Register, PWMDDC, is programmed. If this timer then expires (time interval transpires) without the duty cycle being programmed, the current duty cycle will be set to 100%, and will remain there until the Desired Duty Cycle Register is programmed. This protects the system thermally from failures in the closed-loop fan speed control subsystem.

Bit	Description
7	**CARTPE - Timer Preempt Enable.** When this bit is set to 1 the Fan Spin-up Timer will be terminated after a single fan revolution is detected by the Fan Speed Sensor logic. The 100% duty cycle spin-up will be aborted and the value in the PDDC register will be applied to the PWM output. The purpose of this bit is to allow the fan to overcome start-up inertia, but without over-revving the fan in the event the Fan Spin-up Timer Interval is set too large. ***Note:*** If one PWM output is associated with multiple Fan Speed Sensors, A 100% Duty Cycle will be applied to that PWM until one revolution is completed by all associated fan speed sensors.
6	**CARPSI – PWM Signal Invert.** When reset (0), indicates that the output signal will always be HIGH when at 100% duty cycle. When set (1), indicates that the output signal will always be LOW when at 100% duty cycle.
5:3	**CARFST – Fan Spin-up Time.** Specifies the maximum amount of time that the PWM signal will be asserted with a 100% duty cycle, in order to overcome inertia and get the associated fan(s) spinning. Possible settings are as follows: 000 - 0 ms 001 - 250 ms (default) 010 - 500 ms 011 - 750 ms 100 - 1000 ms 101 - 1500 ms 110 - 2000 ms 111 - 4000 ms Note that if the FSTPE bit is set, the 100% duty cycle spin-up timer may be terminated before it expires.
2:0	**CARPSF – PWM Signal Frequency.** Specifies the frequency for the PWM signal. Possible settings for this field are: 000 - ~10 Hz 001 - ~23 Hz 010 - ~38 Hz 011 - ~62 Hz 100 - ~94 Hz 101 - ~22 KHz 110 - ~25 KHz (default) 111 - ~28 KHz

5.4.1.3 SetCSFanController#Speed

This command is used to set the fan speed of the various fan speed controllers.

Table 112: SetCSFanController#Speed SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	3
02h	Response Data Size	0
03h	Command Code	36h-3Dh
04h-05h	Fan Controller Speed	See 5.4.1.3.1

The Command Codes reserved for chipset fan speed controllers are detailed in the following table:

Table 113: SetCSFanController#Speed SST Command Codes

Command Code	Sensor	Currently Supported?
36h	PCH Fan Speed Controller 1	Yes
37h	PCH Fan Speed Controller 2	Yes
38h	PCH Fan Speed Controller 3	Yes
39h	PCH Fan Speed Controller 4	Yes
3Ah	PCH Fan Speed Controller 5	No
3Bh	PCH Fan Speed Controller 6	No
3Ch	PCH Fan Speed Controller 7	No
3Dh	PCH Fan Speed Controller 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_SET_FAN_CTRL_1_SPEED         0x36
#define SST_CS_SET_FAN_CTRL_2_SPEED         0x37
#define SST_CS_SET_FAN_CTRL_3_SPEED         0x38
#define SST_CS_SET_FAN_CTRL_4_SPEED         0x39
#define SST_CS_SET_FAN_CTRL_5_SPEED         0x3A    // currently not supported
#define SST_CS_SET_FAN_CTRL_6_SPEED         0x3B    // currently not supported
#define SST_CS_SET_FAN_CTRL_7_SPEED         0x3C    // currently not supported
#define SST_CS_SET_FAN_CTRL_8_SPEED         0x3D    // currently not supported
```

5.4.1.3.1 Fan Controller Speed

Depending upon the attribute data returned by command GetCSFanController#Attributes, the value will either be a fan speed specified in RPMs or a Duty Cycle Percentage.

For Fan Controllers requiring that speeds be specified (or returned) as Duty Cycle percentages, note the following:

1. Duty Cycle values are specially encoded. Values in the range 00h-FFh (0-255) are used to fractionally specify percentages in the range 0-100%.

2. In order to obtain a true Duty Cycle percentage value, a calculation must be performed. The equation necessary is:

   Duty Cycle = ((float)Speed * 100) / 255;

For Fan Controllers requiring that speeds be specified (or returned) in RPMs, note the following:

1. Fan Controllers that use RPMs attempt to hold the attached fan's speed to the RPM value specified. However, this does not necessarily mean that, at any specific point in time, the fan will be operating at exactly this specified speed.

2. Fan Controllers that use RPMs are also required to provide the maximum speed of the attached fan. This speed, (also) represented in RPMs, is included in the attribute data. Since a bit of the attribute word was lost to the specification of whether duty cycle or RPM values are being used, only 15 bits are available to represent the maximum speed. Thus, the contents of this field (FSCRPM) must be multiplied by 2 (two), in order to get the true maximum RPMs.

   Using this information, a Duty Cycle percentage value can be calculated. The equation for doing so is as follows:

   Duty Cycle = ((float)Speed / ((float)FSCRPM * 2)) * 100;

   The following (simplified) equation may also be used:

   Duty Cycle = ((float)Speed * 50) / (float)FSCRPM;

5.4.1.4 GetCSFanController#Speed

This command is used to set the fan speed of the various fan speed controllers.

Table 114: GetCSFanController#Speed SST Command Packet

Byte Offset	Description	Contents
00h	Device Address	20h
01h	Command Data Size	1
02h	Response Data Size	2
03h	Command Code	3Eh-45h

Table 115: GetCSFanController#Speed SST Response Packet

Byte Offset	Description
00h-01h	Fan Controller Speed (see section **5.4.1.3.1** (above)for details)

The Command Codes reserved for chipset fan speed controllers are detailed in the following table:

Table 116: GetCSFanController#Speed SST Command Codes

Command Code	Sensor	Currently Supported?
3Eh	PCH Fan Speed Controller 1	Yes
3Fh	PCH Fan Speed Controller 2	Yes
40h	PCH Fan Speed Controller 3	Yes
41h	PCH Fan Speed Controller 4	Yes
42h	PCH Fan Speed Controller 5	No
43h	PCH Fan Speed Controller 6	No
44h	PCH Fan Speed Controller 7	No
45h	PCH Fan Speed Controller 8	No

The following literals from QstCmd.h can be used to specify these command codes:

```
#define SST_CS_GET_FAN_CTRL_1_SPEED         0x3E
#define SST_CS_GET_FAN_CTRL_2_SPEED         0x3F
#define SST_CS_GET_FAN_CTRL_3_SPEED         0x40
#define SST_CS_GET_FAN_CTRL_4_SPEED         0x41
#define SST_CS_GET_FAN_CTRL_5_SPEED         0x42    // currently not supported
#define SST_CS_GET_FAN_CTRL_6_SPEED         0x43    // currently not supported
#define SST_CS_GET_FAN_CTRL_7_SPEED         0x44    // currently not supported
#define SST_CS_GET_FAN_CTRL_8_SPEED         0x45    // currently not supported
```

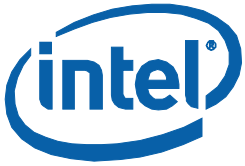
5.5 Summary

The following table summarizes the commands for accessing and controlling the Sensors and Fan Speed Controllers within the chipset:

Table 117: Chipset H/W Access/Control SST Commands

Reference	Command	SST Command Code(s)
5.2.2.1	GetCSTemperature#Reading	00h-0Dh
5.3.2.1	GetCSFanSensor#Attributes	0Eh-15H
5.3.2.2	SetCSFanSensor#Configuration	16h-1Dh
5.3.2.3	GetCSFanSensor#Speed	1Eh-25h
5.4.1.1	GetCSFanController#Attributes	26h-2Dh
5.4.1.2	SetCSFanController#Configuration	2Eh-35h
5.4.1.3	SetCSFanController#Speed	36h-3Dh
5.4.1.4	GetCSFanController#Speed	3Eh-45h

§



Appendix A
Intel® QST 2.0 Configuration Payload

This appendix details the contents of the Intel® QST's binary Configuration Payload. It is NOT intended to be a reference for preparing configurations, however; this is documented separately in the Configuration and Tuning Manual.

Note: The definitions used throughout this appendix are excerpts from header file QstCfg.h.

A.1 Implementation Notes

A.1.1 Indexing Variables

In order to simplify the configuration parameters and improve comprehension, the INI file utilizes process indexing starting from 1 (i.e. Temperature Monitors are numbered from 1-32). Within the (binary) Configuration Payload, however, zero-based indexing is utilized (i.e. Temperature Monitors are indexed from 0-31).

A.1.2 Supporting "NONE"

Indexing (association) and Enumeration Fields that have specific sets of acceptable values – but which also require a special encoding that is used to indicate that no standard choice/association exists (i.e. those INI file parameters that support the "NONE" option) – will use value ~0 (i.e. all bits set (1)) for this encoding. Care must be taken applying this encoding, considering the use of bit fields for some parameters. The following definitions are available to assist with field interpretation:

```
#define QST_VALUE_NONE_INT          -1
#define QST_VALUE_NONE_UINT3        0x07
#define QST_VALUE_NONE_UINT5        0x1F
#define QST_VALUE_NONE_UINT8        0xFF
#define QST_VALUE_NONE_UINT16       0xFFFF
#define QST_VALUE_NONE_UINT32       0xFFFFFFFF
```

A.2 Payload Structure

The Intel® QST 2.0 Binary Configuration Payload will consist of a header record, detailing the size of the payload, etc., followed by some number of entity configuration records, detailing the configuration for each enabled entity. This differs from the Intel® QST 1.x definition, which included a fixed number of each type of entity record, regardless of whether these records were enabled or not.

At the software/firmware level, working with the variable-length payload that is used for communicating the configuration to the QST Subsystem can be difficult. The

larger structure that provides for easy (indexed) access to the entity definitions is much more desirable. Thus, while the header file cannot present a (complete) structure definition for the compressed structure, one can certainly be provided for the larger, fixed-length structure…

Note: Functions providing support for the payload compression and expansion (into a larger, fixed representation) are provided in the Intel® QST SDK.

The following structure definition details the contents of the fixed-size representation of the Binary Configuration Payload:

```
#define QST_ABS_TEMP_MONITORS       32
#define QST_ABS_FAN_MONITORS        32
#define QST_ABS_VOLT_MONITORS       32
#define QST_ABS_CURR_MONITORS       32
#define QST_ABS_TEMP_RESPONSES      32
#define QST_ABS_FAN_CONTROLLERS     32

typedef struct _QST_ABS_PAYLOAD_STRUCT
{
   QST_PAYLOAD_HEADER_STRUCT        Header;
   QST_TEMP_MONITOR_STRUCT          TempMon[QST_ABS_TEMP_MONITORS];
   QST_FAN_MONITOR_STRUCT           FanMon[QST_ABS_FAN_MONITORS];
   QST_VOLT_MONITOR_STRUCT          VoltMon[QST_ABS_VOLT_MONITORS];
   QST_CURR_MONITOR_STRUCT          CurrMon[QST_ABS_CURR_MONITORS];
   QST_TEMP_RESPONSE_STRUCT         Response[QST_ABS_TEMP_RESPONSES];
   QST_FAN_CONTROLLER_STRUCT        FanCtrl[QST_ABS_FAN_CONTROLLERS];

} QST_ABS_PAYLOAD_STRUCT, *P_QST_ABS_PAYLOAD_STRUCT;
```

A.3 Header Record

The Binary Payload will begin with a Header record which can be used to verify that the binary data actually represents a valid FSC configuration. The structure definition for this Header record is defined as follows:

```
typedef struct _QST_PAYLOAD_HEADER_STRUCT
{
   UINT32                           Signature;
   UINT8                            VersionMajor;
   UINT8                            VersionMinor;
   UINT16                           PayloadLength;

} QST_PAYLOAD_HEADER_STRUCT, *P_QST_PAYLOAD_HEADER_STRUCT;
```

A.3.1 Signature

This field is used to quickly verify that the binary data represents a FSC Configuration. The following literal will be used to specify the contents of this field:

```
#define QST_SIGNATURE          "AFSC"
```

A.3.2 VersionMajor

This field identifies the major revision of the specification that this FSC Configuration is compliant with. The current revision of the specification is v2.0, so this field should contain value 2. The following literal will be used to specify the contents of this field:

```
#define QST_VERSION_MAJOR      2
```

A.3.3 VersionMinor

This field identifies the minor revision of the specification that this FSC Configuration is compliant with. The current revision of the specification is v2.0, so this field should contain value 0. The following literal will be used to specify the contents of this field:

```
#define QST_VERSION_MINOR      0
```

A.3.4 PayloadLength

This field specifies the overall length, in bytes, of the Binary Payload. This includes the contents of the header record itself.

A.4 Entity Records

A.4.1 Entity Record Header

Each record begins with a header, defined as follows:

```
typedef struct _QST_HEADER_STRUCT
{
   BIT_FIELD_IN_UINT8           EntityEnabled:1;
   BIT_FIELD_IN_UINT8           EntityType:4;
   BIT_FIELD_IN_UINT8           Filler:3;
   UINT8                        EntityIndex;
   UINT8                        EntityUsage;
   UINT8                        StructLength;

} QST_HEADER_STRUCT, *P_QST_HEADER_STRUCT;
```

A.4.1.1 EntityType

This field specifies the type of entity being configured. These fields will be used by the F/W to determine how/where to apply updates (received from the BIOS) to thresholds, etc. The following literal definitions will be used for this field:

```
#define QST_PAYLOAD_HEADER         0
#define QST_TEMP_MONITOR           1
#define QST_FAN_MONITOR            2
#define QST_VOLT_MONITOR           3
#define QST_CURR_MONITOR           4
#define QST_TEMP_RESPONSE          5
#define QST_FAN_CONTROLLER         6
```

A.4.1.2 EntityIndex

This field specifies the index of the entity being defined (0-based indexing). This value should be consistent with the array index used to access the record.

A.4.1.3 EntityEnabled

This Boolean field specifies whether or not the entity is being enabled. It should be set to TRUE or FALSE. If the entity is disabled, the contents of all other configuration fields for the entity will be ignored (you should set them to 0 or memset the entire payload buffer to 0 ahead of time).

A.4.1.4 StructLength

This field specifies the length of the record. The following literal definitions will be used for this field:

```
#define QST_TEMP_MONITOR_SIZE      sizeof(QST_TEMP_MONITOR_STRUCT)
#define QST_FAN_MONITOR_SIZE       sizeof(QST_FAN_MONITOR_STRUCT)
#define QST_VOLT_MONITOR_SIZE      sizeof(QST_VOLT_MONITOR_STRUCT)
#define QST_CURR_MONITOR_SIZE      sizeof(QST_CURR_MONITOR_STRUCT)
#define QST_TEMP_RESPONSE_SIZE     sizeof(QST_TEMP_RESPONSE_STRUCT)
```

A.4.1.5 EntityUsage

This field specifies the usage for the entity definition. For Fan Monitors and Fan Controllers, the following literal definitions will be used:

```
#define QST_CPU_FAN                1
#define QST_CPU_SYS_FAN            2
#define QST_MCH_FAN                3
#define QST_VR_FAN                 4
#define QST_CHASSIS_FAN            5
#define QST_CHASSIS_INLET_FAN      6
#define QST_CHASSIS_OUTLET_FAN     7
#define QST_PSU_FAN                8
#define QST_PSU_INLET_FAN          9
#define QST_PSU_OUTLET_FAN         10
#define QST_HARD_DRIVE_FAN         11
#define QST_GPU_FAN                12
#define QST_AUX_FAN                13
#define QST_IOH_FAN                14
#define QST_PCH_FAN                15

#define QST_LAST_FAN_USAGE         15
```

For Temperature Monitors and Temperature Response Units, the following literal definitions will be used:

```
#define QST_UNKNOWN_OTHER          0
#define QST_CPU_CORE_TEMP          1
#define QST_CPU_DIE_TEMP           2
#define QST_PCH_TEMP               3
#define QST_MCH_TEMP               4
#define QST_VR_TEMP                5
#define QST_MEM_TEMP               6
#define QST_MOBO_AMBIENT_TEMP      7
#define QST_SYS_AMBIENT_TEMP       8
#define QST_CPU_INLET_TEMP         9
#define QST_SYS_INLET_TEMP         10
#define QST_SYS_OUTLET_TEMP        11
#define QST_PSU_TEMP               12
#define QST_PSU_INLET_TEMP         13
#define QST_PSU_OUTLET_TEMP        14
#define QST_HARD_DRIVE_TEMP        15
#define QST_GPU_TEMP               16
#define QST_IOH_TEMP               17
#define QST_PCH_TEMP               18

#define QST_LAST_TEMP_USAGE        18
```

For Voltage Monitors, the following literal definitions will be used:

```
#define QST_12_VOLTS               1
#define QST_NEG_12_VOLTS           2
#define QST_5_VOLTS                3
#define QST_5_VOLT_BACKUP          4
#define QST_NEG_5_VOLTS            5
#define QST_3P3_VOLTS              6
#define QST_2P5_VOLTS              7
#define QST_1P5_VOLTS              8
#define QST_CPU1_VCCP_VOLTS        9
#define QST_CPU2_VCCP_VOLTS        10
#define QST_CPU3_VCCP_VOLTS        11
#define QST_CPU4_VCCP_VOLTS        12
#define QST_PSU_INPUT_VOLTAGE      13
#define QST_MCH_VCC_VOLTS          14
#define QST_3P3_VOLT_STANDBY       15
#define QST_CPU_VTT_VOLTAGE        16
#define QST_1P8_VOLTS              17
#define QST_PCH_VCC_VOLTAGE        18

#define QST_LAST_VOLT_USAGE        18
```

Finally, for Current Monitors, the following literal definitions will be used:

```
#define QST_12V_CURRENT            1
#define QST_NEG_12V_CURRENT        2
#define QST_5V_CURRENT             3
#define QST_5V_BACKUP_CURRENT      4
#define QST_NEG_5V_CURRENT         5
#define QST_3P3V_CURRENT           6
#define QST_2P5V_CURRENT           7
#define QST_1P5V_CURRENT           8
#define QST_CPU1_CURRENT           9
#define QST_CPU2_CURRENT           10
#define QST_CPU3_CURRENT           11
#define QST_CPU4_CURRENT           12
#define QST_PSU_INPUT_CURRENT      13
#define QST_MCH_CURRENT            14
#define QST_3P3V_STANDBY_CURRENT   15
#define QST_1P8V_CURRENT           16
#define QST_PCH_CURRENT            17

#define QST_LAST_CURR_USAGE        17
```

A.4.2 Temperature Monitor Records

These records define the operation of Temperature Monitor processes. Each is responsible for monitoring temperature readings from a particular sensor and establishing a health status based upon these readings.

The following structure definition will be used for Temperature Monitor records:

```
typedef struct _QST_TEMP_MONITOR_STRUCT
{
   QST_HEADER_STRUCT           Header;
   UINT8                       DeviceAddress;
   UINT8                       GetReadingCommand;
   UINT8                       RelativeReadings;
   UINT8                       TimeoutNonCritical;
   UINT8                       TimeoutCritical;
   UINT8                       TimeoutNonRecoverable;
   INT32F                      RelativeConversion;
   INT32F                      AccuracyCorrectionSlope;
   INT32F                      AccuracyCorrectionOffset;
   INT32F                      TemperatureNominal;
   INT32F                      TemperatureNonCritical;
   INT32F                      TemperatureCritical;
   INT32F                      TemperatureNonRecoverable;

} QST_TEMP_MONITOR_STRUCT, *P_QST_TEMP_MONITOR_STRUCT;
```

A.4.2.1 Header

The Header structure should be initialized in all records, even those that will be marked disabled (field EntityEnabled set to FALSE). The value put into field EntityUsage is unimportant for disabled records.

A.4.2.2 DeviceAddress

This parameter is used to specify the Address of the device that contains the Temperature Sensor. Specify this as follows:

1. For Temperature Sensors within or exposed by the PCH (including Processor and MCH), specify the special address reserved for chipset resources, namely 0x20.

2. For Temperature Sensors within SST Bus-based devices, specify the device's SST Bus Address.

3. For Virtual Temperature Monitors – those that do not directly access a temperature sensor themselves (software supplies readings from the sensor) – set the variable to "None" (~0 (0xFF)).

A.4.2.3 GetReadingCommand

This parameter is used to specify the command byte that is used to obtain a temperature reading from the Sensor. This parameter is ignored for Virtual Temperature Monitors (i.e. if -1 (0xFF) was specified for parameter DeviceAddress). Specify the value for this parameter as follows:

1. For Temperature Sensors within or exposed by the PCH (including Processor and MCH), specify the appropriate command byte. See Table 94 for details.

2. For temperature sensors within SST Bus-based devices, specify the command byte defined for obtaining temperature readings from the appropriate sensor.

A.4.2.4 RelativeReadings

This parameter is used to specify whether or not the sensor returns relative (as opposed to absolute) temperature readings. Specify FALSE (0) for absolute temperature sensors or TRUE (1) for relative temperature sensors. Since the Intel® QST 2.0 firmware provides support for obtaining absolute temperature readings for the processor, it is unlikely that any relative temperature sensors will be encountered…

A.4.2.5 TimeoutNonCritical

This parameter is used to specify the amount of time that can transpire without a temperature reading being available before the system must consider the situation non-critical. For Virtual Temperature Monitors, since temperature readings may not be available that often, values may be specified in the range 1 to 253 seconds. For all other Temperature Monitors, the value should be specified in the range 1 to 10 seconds.

Note: For standard Temperature Monitors, 1 second is recommended. For Virtual Temperature Monitors, 120 seconds is recommended.

A.4.2.6 TimeoutCritical

This parameter is used to specify the amount of time that can transpire without a temperature reading being available before the system must consider the situation critical. For Virtual Temperature Monitors, values may be specified in the range TimeoutNonCritical to 254 seconds. For all other Temperature Monitors, the value should be specified in the range TimeoutNonCritical to 10 seconds.

Note: For standard Temperature Monitors, 2 seconds is recommended. For Virtual Temperature Monitors, 240 seconds is recommended.

A.4.2.7 TimeoutNonRecoverable

This parameter is used to specify the amount of time that can transpire without a temperature reading being available before the system must consider the situation non-recoverable. For Virtual Temperature Monitors, values may be specified in the range TimeoutCritical to 254 seconds. Alternatively, “None” can be specified to

indicate that no Non-Recoverable Timeout is required. For all other Temperature Monitors, the value should be specified in the range TimeoutCritical to 10 seconds.

Note: For standard Temperature Monitors, 2 seconds is recommended. For Virtual Temperature Monitors, "None" (~0, 0xFF) is recommended.

Note: When a temperature sensor enters the Non-Recoverable state, all fans will be driven to full speed, in order to protect the system from any thermal problems that may be occurring (without the Subsystem's knowledge).

A.4.2.8 RelativeConversion

This parameter is used to specify a temperature offset which, when added to relative temperature values, converts them into absolute temperature values. If no conversion factor is available, specify value 0 (zero) for this parameter.

Since Processor DTS readings will be provided in absolute form, this parameter should be set to 0 (zero) for the Temperature Monitor associated with the processor sensor.

A.4.2.9 AccuracyCorrectionSlope

This field, along with AccuracyCorrectionOffset, specifies the factors that are used to correct for inaccuracies in the temperature readings obtained from a temperature sensor. The fully qualified temperature correction equation is of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor.

For sensors that do not require or do not have an available Slope Correction Factor, value 1.0 should be specified for the Accuracy Correction Slope. Values for this field are specified in hundredths units. That is, value 1.0 would be specified as 100.

A.4.2.10 AccuracyCorrectionOffset

This field, along with AccuracyCorrectionSlope, specifies the factors that are used to correct for inaccuracies in the temperature readings obtained from a temperature sensor. The fully qualified temperature correction equation is of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor.

For sensors that do not require or do not have an available Offset Correction Factor, value 0.0 should be specified for the Accuracy Correction Offset. Values for this field are specified in hundredths of a degree (Celsius). That is, value 25.0 would be specified as 2500.

A.4.2.11 TemperatureNominal

This parameter is provided to fulfill the data requirements of Manageability Software stacks such as DMI, CIM/WBEM (WMI), SNMP, etc. The Instrumentation Layer will make this value available to calling manageability software. It is used to identify the nominal temperature expected from this sensor. This is typically a representation of the temperature seen from the sensor while the system is idle. Specify this parameter as follows:

- For temperatures from sources other than the Processor, specify an absolute temperature value in hundredths of a degree (Celsius).
- For temperatures from a Processor, specify a (positive or negative) temperature offset value in hundredths of a degree (Celsius). Intel® QST will sum this value with the Tcontrol temperature for this Processor, in order to establish the actual nominal temperature.

A.4.2.12 TemperatureNonCritical

This parameter is used to specify the temperature threshold above which the Monitor should return a Non-Critical Health Status. Specify this parameter as follows:

- For temperatures from sources other than the Processor, specify an absolute temperature value in hundredths of a degree (Celsius).
- For temperatures from a Processor, specify a (positive or negative) temperature offset in hundredths of a degree (Celsius). The Intel® QST Subsystem will sum this value with the $T_{CONTROL}$ temperature for this Processor, in order to establish the actual temperature threshold.

A.4.2.13 TemperatureCritical

This parameter is used to specify the temperature threshold above which the Monitor should return a Critical Health Status. Specify this parameter as follows:

- For temperatures from sources other than the Processor, specify an absolute temperature value in hundredths of a degree (Celsius).
- For temperatures from a Processor, specify a (positive or negative) temperature offset in hundredths of a degree (Celsius). The Intel® QST Subsystem will sum this value with the $T_{CONTROL}$ temperature for this Processor, in order to establish the actual temperature threshold.

A.4.2.14 TemperatureNonRecoverable

This parameter is used to specify the temperature threshold above which the Monitor should return a Non-Recoverable Health Status. Specify this parameter as follows:

- For temperatures from sources other than the Processor, specify an absolute temperature value in hundredths of a degree (Celsius).
- For temperatures from a Processor, specify a (positive or negative) temperature offset in hundredths of a degree (Celsius). The Intel® QST Subsystem will sum this value with the $T_{CONTROL}$ temperature for this Processor, in order to establish the actual temperature threshold.

A.4.3 Fan Monitor Records

These records define the operation of Fan Monitor processes. Each is responsible for monitoring fan speed readings from a particular sensor and establishing a health status based upon these readings.

The following structure definition will be used for Fan Monitor records:

```
typedef struct _QST_FAN_MONITOR_STRUCT
{
   QST_HEADER_STRUCT           Header;
   UINT8                       DeviceAddress;
   UINT8                       GetAttributesCommand;
   UINT8                       GetReadingCommand;
   UINT16                      SpeedNominal;
   UINT16                      SpeedNonCritical;
   UINT16                      SpeedCritical;
   UINT16                      SpeedNonRecoverable;

} QST_FAN_MONITOR_STRUCT, *P_QST_FAN_MONITOR_STRUCT;
```

A.4.3.1 Header

The Header structure should be initialized in all records, even those that will be marked disabled (field EntityEnabled set to FALSE).

A.4.3.2 DeviceAddress

This parameter is used to specify the address of the device that contains the Fan Speed Sensor. It is specified as an unsigned 8-bit quantity. Specify this as follows:

1. For the Fan Speed Sensors within the PCH, specify the address reserved for chipset resources, namely 0x20.
2. For Fan Speed Sensors within SST Bus-based devices, specify the appropriate device's SST Bus Address.

A.4.3.3 GetAttributesCommand

This parameter is used to specify the command byte that is used to obtain attributes from the sensor. It is specified as an unsigned 8-bit quantity. Specify this as follows:

1. For fan speed sensors within the PCH, specify the appropriate command byte from Table 97.
2. For fan speed sensors located within SST Bus-based devices, specify the command byte defined for obtaining attribute data from the sensor.

A.4.3.4 GetReadingCommand

This parameter is used to specify the command byte that is used to obtain a fan speed reading from the Sensor. Specify this as follows:

1. For fan speed sensors within the PCH, specify the appropriate command byte from Table 104.
2. For fan speed sensors within SST Bus-based devices, specify the command byte defined for obtaining fan speed readings from the sensor.

A.4.3.5 SpeedNominal

This parameter is used to specify, in RPMs, the nominal speed for the fan being monitored by this sensor. This parameter is necessary to fulfill the data requirements of Manageability Software stacks, such as DMI, CIM/WBEM/WMI, SNMP, etc.

A.4.3.6 SpeedNonCritical

This parameter is used to specify, in RPMs, the fan speed below which the Monitor will return a Non-Critical Health Status. Note that this does not include those instances where the fan has been commanded to stop.

A.4.3.7 SpeedCritical

This parameter is used to specify, in RPMs, the fan speed below which the Monitor will return a Critical Health Status. This does not include those instances where the fan has been commanded to stop.

A.4.3.8 SpeedNonRecoverable

This parameter is used to specify, in RPMs, the fan speed below which the Monitor will return a Non-Recoverable Health Status. This does not include those instances where the fan has been commanded to stop.

A.4.4 Voltage Monitor Records

These records define the operation of Voltage Monitor processes. Each is responsible for monitoring voltage readings from a particular sensor and establishing a health status based upon these readings.

The following structure definition will be used for Voltage Monitor records:

```
typedef struct _QST_VOLT_MONITOR_STRUCT
{
   QST_HEADER_STRUCT           Header;
   UINT8                       DeviceAddress;
   UINT8                       GetReadingCommand;
   INT32                       AccuracyCorrectionSlope;
   INT32                       AccuracyCorrectionOffset;
   INT32                       VoltageNominal;
   INT32                       UnderVoltageNonCritical;
   INT32                       UnderVoltageCritical;
   INT32                       UnderVoltageNonRecoverable;
   INT32                       OverVoltageNonCritical;
   INT32                       OverVoltageCritical;
   INT32                       OverVoltageNonRecoverable;

} QST_VOLT_MONITOR_STRUCT, *P_QST_VOLT_MONITOR_STRUCT;
```

A.4.4.1 Header

The Header structure should be initialized in all records, even those that will be marked disabled (field EntityEnabled set to FALSE). The value put into field EntityUsage is unimportant for disabled records.

A.4.4.2 DeviceAddress

This parameter is used to specify the address of the device that contains the Voltage Sensor. It is specified as an unsigned 8-bit quantity. Only Voltage Sensors contained with SST Bus-based devices are supported.

A.4.4.3 GetReadingCommand

This parameter is used to specify the command byte used to obtain a Voltage reading from the (SST Bus-based device's) Sensor. It is specified as an unsigned 8-bit quantity. Specify the command byte appropriate to the device and sensor.

A.4.4.4 AccuracyCorrectionSlope

This field, along with AccuracyCorrectionOffset, specifies the factors that are used to correct for inaccuracies in the temperature readings obtained from a voltage sensor. The fully qualified temperature correction equation is of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor.

For sensors that do not require or do not have an available Slope Correction Factor, value 1.0 should be specified for the Accuracy Correction Slope. Values for this field are specified in 1/1000 units. That is, value 1.0 would be specified as 1000.

A.4.4.5 AccuracyCorrectionOffset

This field, along with AccuracyCorrectionSlope, specifies the factors that are used to correct for inaccuracies in the temperature readings obtained from a temperature sensor. The fully qualified temperature correction equation is of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor.

For sensors that do not require or do not have an available Offset Correction Factor, value 0.0 should be specified for the Accuracy Correction Offset. Values for this field are specified in 1/1000 units (millivolts). That is, value 1.0 would be specified as 1000.

A.4.4.6 VoltageNominal

This parameter is necessary to fulfill the data requirements of Manageability Software stacks, such as DMI, CIM/WBEM/WMI, SNMP, etc. It is used to specify the nominal voltage value from this sensor. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify the nominal voltage as an absolute value in millivolts. For example, for the 3.3 Volt Rail, you would specify value 3300.
- For Processor Vccp Voltages, specify value 0 (zero). While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will replace this value with a sampled voltage level from the Processor.

A.4.4.7 UnderVoltageNonCritical

This parameter is used to specify the Voltage threshold below which the Monitor will return a Non-Critical Health Status. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify this threshold as an absolute value in millivolts.
- For Processor Vccp Voltages, specify a (negative) voltage offset. While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will sum this value with the established nominal voltage to produce the actual threshold.

A.4.4.8 UnderVoltageCritical

This parameter is used to specify, in millivolts, the Voltage below which the Monitor will return a Critical Health Status. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify this threshold as an absolute value in millivolts.
- For Processor Vccp Voltages, specify a (negative) voltage offset. While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will sum this value with the established nominal voltage to produce the actual threshold.

A.4.4.9 UnderVoltageNonRecoverable

This parameter is used to specify the Voltage below which the Monitor will return a Non-Recoverable Health Status. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify this threshold as an absolute value in millivolts.
- For Processor Vccp Voltages, specify a (negative) voltage offset. While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will sum this value with the established nominal voltage to produce the actual threshold.

A.4.4.10 OverVoltageNonCritical

This parameter is used to specify the Voltage above which the Monitor will return a Non-Critical Health Status. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify this threshold as an absolute value in millivolts.
- For Processor Vccp Voltages, specify a (positive) voltage offset. While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will sum this value with the established nominal voltage to produce the actual threshold.

A.4.4.11 OverVoltageCritical

This parameter is used to specify the Voltage above which the Monitor will return a Critical Health Status. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify this threshold as an absolute value in millivolts.
- For Processor Vccp Voltages, specify a (positive) voltage offset. While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will sum this value with the established nominal voltage to produce the actual threshold.

A.4.4.12 OverVoltageNonRecoverable

This parameter is used to specify the Voltage above which the Monitor will return a Non-Recoverable Health Status. Specify this value as follows:

- For voltages sources other than the Processor Vccp Voltage, specify this threshold as an absolute value in millivolts.
- For Processor Vccp Voltages, specify a (positive) voltage offset. While processing the configuration for the first time and whenever a Processor change is detected, Intel® QST will sum this value with the established nominal voltage to produce the actual threshold.

A.4.5 Current Monitor Records

These records define the operation of Current Monitor processes. Each is responsible for monitoring current readings from a particular sensor and establishing a health status based upon these readings.

The following structure definition will be used for Current Monitor records:

```
typedef struct _QST_CURR_MONITOR_STRUCT
{
   QST_HEADER_STRUCT           Header;
   UINT8                       SensorType;
   UINT8                       DeviceAddress;
   UINT8                       GetReadingCommand;
   INT32                       AdjustmentSlope;
   INT32                       AdjustmentOffset;
   INT32                       CurrentNominal;
   INT32                       UnderCurrentNonCritical;
   INT32                       UnderCurrentCritical;
   INT32                       UnderCurrentNonRecoverable;
   INT32                       OverCurrentNonCritical;
   INT32                       OverCurrentCritical;
   INT32                       OverCurrentNonRecoverable;

} QST_CURR_MONITOR_STRUCT, *P_QST_CURR_MONITOR_STRUCT;
```

A.4.5.1 Header

The Header structure should be initialized in all records, even those that will be marked disabled (field EntityEnabled set to FALSE). The value put into field EntityUsage is unimportant for disabled records.

A.4.5.2 Sensor Type

This parameter is used to specify whether Current readings are read from an actual Current sensor or are read from a Voltage sensor and then scaled to produce a Current reading. The following literals from QstCfg.h are used to set this field:

```
#define QST_CURRENT_SENSOR          0       // True current sensor (adjustment
                                            // slope/offset used for accuracy)

#define QST_VOLTAGE_SENSOR          1       // Current derived from voltage
                                            // (using adjustment slope/offset)
```

A.4.5.3 DeviceAddress

This parameter is used to specify the address of the device that contains the Current or Voltage Sensor. Only Sensors contained with SST Bus-based devices are supported.

A.4.5.4 GetReadingCommand

This parameter is used to specify the command byte used to obtain a reading from the appropriate Sensor. It is specified as an unsigned 8-bit quantity. Specify the command byte appropriate to the device and sensor.

A.4.5.5 AdjustmentOffset

For actual Current sensors, this parameter is used for accuracy correction. A fully qualified Current correction capability, used to correct for inaccuracies in the readings obtained from a particular current sensor, applies an equation of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor. This field specifies this Offset Correction Factor.

For Current sensors that are implemented using a Voltage sensor, this parameter is used to scale the reading from a voltage level to a Current value. Scaling the voltage level is also done with an equation of form "Y = MX + B", where "M" is the Slope Adjustment Factor and "B" is the Offset Adjustment Factor. This field specifies the Offset Adjustment Factor.

Values are specified accurate to 3 decimal places (i.e. milliamps). For sensors that do not require any correction for inaccuracies, the Offset Correction Factor should be specified as 0.0.

A.4.5.6 AdjustmentSlope

For actual Current sensors, this parameter is used for accuracy correction. A fully qualified Current correction capability, used to correct for inaccuracies in the readings obtained from a particular current sensor, applies an equation of form "Y = MX + B", where "M" is the Slope Correction Factor and "B" is the Offset Correction Factor. This field specifies this Slope Correction Factor.

For Current sensors that are implemented using a Voltage sensor,, this parameter is used to scale the reading from a voltage level to a Current value. Scaling the voltage level is also done with an equation of form "Y = MX + B", where "M" is the Slope Adjustment Factor and "B" is the Offset Adjustment Factor. This field specifies the Slope Adjustment Factor.

Values are specified accurate to 3 decimal places (i.e. milliamps). For sensors that do not require any correction for inaccuracies, the Slope Correction Factor should be specified as 1.0 (i.e. 1000).

5.5.1.1 CurrentNominal

This parameter is necessary to fulfill the data requirements of Manageability Software stacks, such as DMI, CIM/WBEM/WMI, SNMP, etc. It is used to specify the nominal voltage value from this sensor. Current values are specified in milliamps.

5.5.1.2 UnderCurrentNonCritical

This parameter is used to specify the Current threshold below which the Monitor will return a Non-Critical Health Status. Current values are specified in milliamps.

5.5.1.3 UnderCurrentCritical

This parameter is used to specify the Current below which the Monitor will return a Critical Health Status. Current values are specified in milliamps.

5.5.1.4 UnderCurrentNonRecoverable

This parameter is used to specify the Current below which the Monitor will return a Non-Recoverable Health Status. Current values are specified in milliamps.

5.5.1.5 OverCurrentNonCritical

This parameter is used to specify the Current above which the Monitor will return a Non-Critical Health Status. Current values are specified in milliamps.

5.5.1.6 OverCurrentCritical

This parameter is used to specify the Current above which the Monitor will return a Critical Health Status. Current values are specified in milliamps.

5.5.1.7 OverCurrentNonRecoverable

This parameter is used to specify the Current above which the Monitor will return a Non-Recoverable Health Status. Current values are specified in milliamps.

A.4.6 Temperature Response Records

These records define the operation of Temperature Response processes. Each is responsible for determining the magnitude of the response to current and historical temperatures obtained from the associated temperature monitors.

The following structure definition will be used for Temperature Response Unit records:

```
typedef struct _QST_TEMP_RESPONSE_STRUCT
{
   QST_HEADER_STRUCT            Header;
   UINT8                        TemperatureMonitor;
   BIT_FIELD_IN_UINT16          SmoothingWindow:5;
   BIT_FIELD_IN_UINT16          IntegralTimeWindow:5;
   BIT_FIELD_IN_UINT16          DerivativeTimeWindow:5;
   BIT_FIELD_IN_UINT16          Filler:1;
   INT32F                       ProportionalGain;
   INT32F                       IntegralGain;
   INT32F                       DerivativeGain;
   INT32F                       TempLimit;
   INT32F                       TempAllOn;

} QST_TEMP_RESPONSE_STRUCT, *P_QST_TEMP_RESPONSE_STRUCT;
```

A.4.6.1 Header

The Header structure should be initialized in all records, even those that will be marked disabled (field EntityEnabled set to FALSE). The value put into field EntityUsage is unimportant for disabled records.

A.4.6.2 TemperatureMonitor

This parameter is used to specify the index (0-31) of the Temperature Monitor whose temperature readings are to be responded to.

A.4.6.3 SmoothingWindow

This parameter is used to specify the width, in seconds, of the (sliding) smoothing window that is used to minimize spikes in the input temperature readings. Values in the range 0 to 10 seconds are supported. If no smoothing is necessary or desired, specify 0 (zero). The following literal definition is available for this field:

```
#define QST_MAX_TEMP_SMOOTHING    10
```

A.4.6.4 IntegralTimeWindow

This parameter is used to specify the time window over which the Integral Term of the Proportional-Integral-Derivative (PID) algorithm is applied. This is a value in the range 1 to 20 seconds. The following literal definition is available for this field:

```
#define QST_MAX_INTEGRAL_WINDOW   20
```

A.4.6.5 DerivativeTimeWindow

This parameter is used to specify the time window over which the Derivative Term of the Proportional-Integral-Derivative (PID) algorithm is applied. This is a value in the range 1 to 30 seconds. The following literal definition is available for this field:

```
#define QST_MAX_DERIVATIVE_WINDOW 30
```

A.4.6.6 ProportionalGain

This parameter is used to specify the percentage gain to be applied against the Proportional Term of the Proportional-Integral-Derivative (PID) response. The recommended value is 2 (percent). Values are specified in hundredths of a percent. Thus, 2 percent would be represented using value 200.

In a closed loop, the Proportional Term seeks to reduce the error – the difference between the current and target/limit temperatures – in proportion to its instantaneous value. As the gain is increased, the system responds faster to changes in temperature. Increasing the gain too far will result in overshoots, ringing, and ultimately, undamped oscillation in the output.

A.4.6.7 IntegralGain

This parameter is used to specify the percentage gain to be applied against the integral term of the Proportional-Integral-Derivative (PID) response. The recommended value is 5 (percent). Values are specified in hundredths of a percent. Thus, 5 percent would be represented using value 500.

Although proportional control can reduce the error substantially, it cannot by itself reduce it to zero. The Integral Term helps to slowly drive the error towards zero. The higher the integral gain, the sooner the error heads for zero (and beyond) in response to a change; so to set it too high is to invite oscillation and instability.

A.4.6.8 DerivativeGain

This parameter is used to specify the percentage gain to be applied against the Derivative Term of the Proportional-Integral-Derivative (PID) response. The recommended value is 200 (percent). Values are specified in hundredths of a percent. Thus, 200 percent would be represented using value 20000.

The Derivative Term uses the rate of change in the error to anticipate its future value, speeding up the response to the proportional term and tending to improve loop stability by compensating for the integral term's lag. The Derivative Gain is also known as the dampening term. Providing too little damping may cause the overshoot from proportional control to remain in place. Providing too much damping, however, may cause an unnecessarily slow response. The designer should also note that differentiators amplify high frequency noise appearing in the error signal.

A.4.6.9 TempLimit

This parameter is used to specify the limit, or target, temperature that the Response Unit is trying to maintain. Values are specified in hundredths of a degree. Specify as follows:

- For temperatures from other than the Processor, specify an absolute temperature value.
- For temperatures from the Processor, specify a (positive or negative) temperature offset value. Intel® QST will sum this offset value with the $T_{CONTROL}$ temperature for this Processor, in order to establish the actual limit/target temperature. In most cases, this offset value should be 0 (zero); non-zero values should only be used in certain circumstances, such as when overclocking, wherein temperature change can be so rapid that the algorithms cannot address them fast enough to avoid the Processor's Thermal Control Circuit activating from time to time. In this case, a negative offset may be specified (effectively lowering the temperature to be maintained). Note that, while it is possible to specify a positive offset, this is not recommended and will violate your processor warranty.

A.4.6.10 TempAllOn

This parameter is used to specify the temperature threshold that, if surpassed, causes all fans to be overridden to 100%. Values are specified in hundredths of a degree (Celsius). Specify this as follows:

- For temperatures from other than the Processor, specify an absolute temperature value.
- For temperatures from the Processor, specify a temperature offset value. The Intel® QST Subsystem will sum this value with the $T_{CONTROL}$ temperature for this Processor, in order to establish the actual all-on temperature.

Note: This is a safety feature; if a primary fan, even at its maximum speed, is providing insufficient cooling, the other cooling fan(s) available can attempt to assist it.

A.4.7 Fan Controller Records

These records define the operation of up to eight Fan Controller processes. Each is responsible for determining the final duty cycle values that are to be delivered to their associated H/W Fan Speed Controllers.

The following structure definition will be used for Fan Controller records:

```
#define QST_MAX_ASSOC_FAN_SENSORS 4

typedef struct _QST_FAN_CONTROLLER_STRUCT
{
   QST_HEADER_STRUCT            Header;
   UINT8                        DeviceAddress;
   UINT8                        GetAttributesCommand;
   UINT8                        SetConfigurationCommand;
   UINT8                        SetSpeedCommand;
   UINT8                        GetSpeedCommand;
   QST_FAN_SENSOR_STRUCT        FanSensor[QST_MAX_ASSOC_FAN_SENSORS];
   BIT_FIELD_IN_UINT16          PhysicalControllerIndex:3;
   BIT_FIELD_IN_UINT16          OFFMode:1;         // FALSE = MIN Mode
   BIT_FIELD_IN_UINT16          SignalInvert:1;
   BIT_FIELD_IN_UINT16          SignalFrequency:3;
   BIT_FIELD_IN_UINT16          SpinUpTime:3;
   BIT_FIELD_IN_UINT16          Filler:5;
   UINT16                       DutyCycleMin;
   UINT16                       DutyCycleOn;
   QST_AMBIENT_LIMITER          stAmbientFloor;
   UINT16                       DutyCycleMax;
   QST_AMBIENT_LIMITER          stAmbientCeiling;
   INT32F                       ResponseWeighting[QST_ABS_TEMP_RESPONSES];

} QST_FAN_CONTROLLER_STRUCT, *P_QST_FAN_CONTROLLER_STRUCT;
```

A.4.7.1 Header

The Header structure should be initialized in all records, even those that will be marked disabled (field EntityEnabled set to FALSE). The value put into field EntityUsage is unimportant for disabled records.

A.4.7.2 DeviceAddress

This parameter is used to specify the address of the device that contains the Fan Speed Controller. It is specified as an unsigned 8-bit quantity. Specify this as follows:

1. For fan speed controllers within the PCH, specify the address reserved for chipset resources, namely 0x20.

2. For fan speed controllers within SST Bus-based devices, specify the appropriate device's SST Bus Address.

A.4.7.3 GetAttributesCommand

This parameter is used to specify the command byte that is used to obtain attributes from the fan speed controller. Specify as follows:

1. For fan speed controllers within the PCH, specify the appropriate command byte from Table 107.

2. For fan speed controllers located within SST Bus-based devices, specify the command byte defined for obtaining attribute data from the controllers.

A.4.7.4 SetConfigurationCommand

This parameter is used to specify the command byte that is used to write configuration data to the fan speed controller. Support for this operation is an optional feature for fan controllers. If the associated fan speed controller does not support this command, this field should be set to -1. This is the equivalent of specifying "NONE" in the configuration file.

For fan speed controllers within the PCH, specify the appropriate command byte from Table 110.

A.4.7.5 SetSpeedCommand

This parameter is used to specify the command byte that is used to write a fan speed (Duty Cycle) value to the Fan Speed Controller. Specify this as follows:

1. For fan speed controllers within the PCH, specify the appropriate command byte from Table 113.

2. For fan speed controllers within SST Bus-based devices, specify the command byte defined for writing fan speed Duty Cycle values to the controller.

A.4.7.6 GetSpeedCommand

This parameter is used to specify the command byte that is used to read a fan speed (Duty Cycle) value from the Fan Speed Controller. Specify this as follows:

1. For fan speed controllers within the PCH, specify the appropriate command byte from Table 116.

2. For fan speed controllers within SST Bus-based devices, specify the command byte defined for obtaining fan speed (Duty Cycle) values from the controller.

A.4.7.7 FanSensor[]

Up to 4 (four) fans may be associated with and controlled by a fan speed controller. In order to properly operate the controller (manage fan spin up, dependent monitoring, etc.), it must be configured to understand the dependencies. An array of four fan sensor description structures is provided for specifying the details of the dependencies.

Structure QST_FAN_SENSOR_STRUCT is defined as follows:

```
typedef struct _QST_FAN_SENSOR_STRUCT
{
   BIT_FIELD_IN_UINT8          AssociatedFanSpeedMonitor:5;
   BIT_FIELD_IN_UINT8          PulsesPerRevolution:2;
   BIT_FIELD_IN_UINT8          DependentSpeedMeasurement:1;
   UINT8                       SetConfigurationCommand;
   UINT16                      MinDutyRPMMin;
   UINT16                      MinDutyRPMMax;

} QST_FAN_SENSOR_STRUCT;
```

A.4.7.7.1 AssociatedFanSpeedMonitor

These parameters are used to specify the index of the Fan Speed Monitor(s) responsible for handling tachometer readings from fan(s) controlled by this Fan Controller.

In all cases, a Fan Controller controls at least one fan. Thus, within array entry 0, this parameter is required to properly index a Fan Speed Monitor (using index values 0-7). If only a single fan is being controlled, the contents of this variable within array entries 1, 2 and 3 would be set to 0x1F (011111b; indicating no association).

If two fans are being controlled, this parameter, within array entries 0 and 1, would index the appropriate Fan Speed Monitors. Within array entries 2 and 3, this parameter would be set to 0x1F (011111b; indicating no association).

If three fans are being controlled, this parameter, within array entries 0, 1 and 2, would index the appropriate Fan Speed Monitors. Within array entry 3, this parameter would be set to 0x1F (011111b; indicating no association).

Finally, if four fans are being controlled, this parameter, within array entries 0, 1, 2 and 3, would index the appropriate Fan Speed Monitors.

A.4.7.7.2 PulsesPerRevolution

These parameters are ignored if the fan sensor does not require configuration or if their corresponding AssociatedFanSpeedMonitor parameter is set to "None" (0x1F). They are used to specify the number of tachometer pulses that the associated fan returns per revolution. Supported values are 0-3, representing 1-4 pulses per revolution.

Fans typically return a tachometer pulse for each pole of their motor. Most fans have 2-pole motors and thus return 2 pulses per revolution. A very small number of higher-quality 3-wire fans have 3 and 4 poles. 4-wire fans are required, by specification, to return 2 tachometer pulses per revolution, regardless of the number of motor poles utilized.

Note the following literal definitions for the PulsesPerRevolution fields:

```
#define QST_1_PULSE             0
#define QST_2_PULSES            1
#define QST_3_PULSES            2
#define QST_4_PULSES            3
```

A.4.7.7.3 DependentSpeedMeasurement

These parameters are ignored if their corresponding AssociatedFanSpeedMonitor parameter is set to "None" (0x1F). They are used to specify whether or not (TRUE or FALSE) the state of the fan speed control signal needs to be taken into account when analyzing the tachometer input signal from the fan, in order to accurately determine the speed at which the fan(s) are operating. This will only be the case for 3-wire fans that are being controlled using the pulsed-power control methodology (which is the case for the Intel reference boards). It is not necessary for 4-wire fans or for 3-wire fans that are being controlled using the voltage-level control methodology.

A.4.7.7.4 SetConfigurationCommand

These parameters are used to specify the command byte that is used to configure the fan speed sensor that handles tachometer readings from the appropriate fan. They are ignored if their corresponding AssociatedFanSpeedMonitor parameter is set to "None" (0x1F). For those not set to "None", specify their value as follows:

1. For fan speed sensors within the PCH, specify the appropriate command byte from Table 100.
2. For fan speed sensors located within SST Bus-based devices, specify the command byte defined for setting the fan sensor's configuration. If this sensor does not support a configuration command, specify -1 (this is the equivalent of specifying "NONE" in the Configuration (INI) file).

A.4.7.7.5 MinDutyRPMMin/MinDutyRPMMax

These parameters are ignored if their corresponding AssociatedFanSpeedMonitor parameter is set to "None" (0x1F). They are used to specify, in RPMs, the minimum fan speed range for the corresponding fan.

These parameters are used to adjust the control programming for variations in individual fan characteristics. The Fan Controller will use these parameters to adjust the DutyCycleMin value upwards or downwards, if necessary, to guarantee that the fan's minimum speed falls within this range.

Because the performance of fans may degrade over time, the Fan Controller performs the minimum RPM range check each and every time that it is controlling the fan at its Minimum Duty Cycle setting. Since it may take some time (depending upon the condition of the fan) for the Fan Controller to detect the actual required Minimum Duty Cycle, Intel® QST carefully watches the temperatures within the system and will abort the detection process and speed up the fan, should temperatures begin to unduly rise.

A.4.7.8 PhysicalControllerIndex

This parameter is used to specify the index of the physical fan speed controller, within the set of physical fan speed controllers contained in a device. Specify this parameter as follows:

- For the fan speed controllers contained within the chipset, indices 0-7 can be used to reference the particular fan speed controller, but only indices 0-3 are currently valid in the initial version of the PCH.
- If a device on the SST Bus contained (for example) 4 fan speed controllers, indices 0-3 would be used for a particular controller.

A.4.7.9 OFFMode

This parameter is used to specify the mode of operation for the fan controller. Specify TRUE (1) for OFF Mode or FALSE (0) for MIN Mode:

- In MIN Controlled Mode, the fan(s) are never allowed to operate at a duty cycle lower than is specified in the DutyCycleMin parameter. If a lower duty cycle value is requested, this value will be ignored and the DutyCycleMin value will instead be used.
- In OFF Controlled Mode, the fan(s) are shut off (stopped) if a duty cycle lower than DutyCycleMin is requested. While in this mode, if back-to-back duty cycle values are programmed which take the duty cycle below and then back above DutyCycleMin, the fan would be quickly stopped and then restarted. If this cycle were to continue, the affect will be noticeable acoustically and become a considerable source of irritation. As a result, a second threshold (DutyCycleOn) is used to define a (higher) duty cycle value at which the fan is actually restarted. Care must still be taken to keep these thresholds far enough apart to avoid cycling.

A.4.7.10 SignalInvert

This parameter is ignored for fan speed controllers within SST Bus-based devices. For fan speed controllers within the PCH, this parameter is used to specify whether (1) or not (0) the output signal from the fan speed controller H/W needs to be inverted, in order to ensure the proper operation of the circuitry on the motherboard.

A.4.7.11 SignalFrequency

This parameter is ignored for fan speed controllers within SST Bus-based devices. For fan speed controllers within the PCH, this parameter is used to specify the frequency of the output signal from the fan speed controller. Specify this parameter using values 0-7, which specify frequencies 10, 23, 38, 62, 94, 22000, 25000 or 28000 (Hz), respectively.

For 3-wire fans, methods such as power-pulsing and voltage-scaling are used to control rotational speed. The circuits for the power-pulsing methodology require slow (10-94 Hz) signal frequencies. Voltage-scaling implementations requiring both slow (10-94 Hz) or fast (22-28 kHz) have been encountered, however, and those requiring fast frequencies are now the most prevalent.

4-Wire fan, by specification, require a fast (22-28 kHz) signal frequency. It is not an absolute requirement but the 25 kHz frequency is strongly recommended.

In the case of the power-pulsed 3-wire fan control methodology, the signal frequency can have considerable effect on the psycho-acoustics of the fan. This is not the case for the voltage-variance methodology nor is it for 4-wire fans.

Note the following literal definitions that can be used for this field:

```
#define QST_FREQ_10           0
#define QST_FREQ_23           1
#define QST_FREQ_38           2
#define QST_FREQ_62           3
#define QST_FREQ_94           4
#define QST_FREQ_22000        5
#define QST_FREQ_25000        6
#define QST_FREQ_28000        7
```

A.4.7.12 SpinUpTime

This parameter is ignored for fan speed controllers within SST Bus-based devices. For fan speed controllers within the PCH, this parameter is used to specify the length, in milliseconds, of the spin-up interval for fan(s) connected to the controller. This is the amount of time that a stopped fan will be driven with a full-on (100% duty cycle) signal, in order to overcome its inertia and begin spinning. Supported values are 0-7, specifying spinup times of 0, 250, 500, 750, 1000, 1500, 2000 or 4000 (ms).

Note: Care must be taken to set this parameter properly. Choose an interval that is too short and the fan may never begin rotating. Choose an interval that is too long and the fan will spin to a higher than necessary initial speed, causing a noticeable acoustic effect. Typically, a value of 250ms is sufficient.

Note: The following literal definitions that can be used for this field:

```
#define QST_SPIN_0_MS          0
#define QST_SPIN_250_MS        1
#define QST_SPIN_500_MS        2
#define QST_SPIN_750_MS        3
#define QST_SPIN_1000_MS       4
#define QST_SPIN_1500_MS       5
#define QST_SPIN_2000_MS       6
#define QST_SPIN_4000_MS       7
```

A.4.7.13 DutyCycleMin

The ability of fans to operate at very low speeds varies significantly. Issues such as friction, bearing packing, humidity, voltage sensitivity, etc., all play a role in defining the abilities of a particular fan at any point in time. As the duty cycle of the control signal is reduced, thresholds could be reached below which the fan either fails to continue spinning (stalls) or becomes unstable in its reaction (speed fluctuations, often causing undesirable acoustic effects, and motor noise). In order to handle these issues, a limit (the DutyCycleMin parameter) is used to specify how slowly a fan can be asked to spin.

While in OFF Controlled Mode, this variable is used to specify the duty cycle below which the fan should be stopped if it is currently spinning. While in MIN Controlled Mode, this variable is used to specify the lowest duty cycle at which the fan is allowed to spin. Values 0 – 10000 will be used to specify duty cycle percentages in the range 0 – 100 percent (i.e. specified in hundredths of a percent)

A.4.7.14 DutyCycleOn

This parameter is used to specify the Fan-On Duty Cycle for the Fan Controller. While in MIN Controlled Mode, this parameter is ignored. While in OFF Controlled Mode, this threshold specifies the duty cycle above which the fan, if stopped, will be restarted. See the description of the MinOffMode parameter (above) for more details. Values 0 – 10000 will be used to specify duty cycle percentages in the range 0 – 100 percent (i.e. specified in hundredths of a percent)

A.4.7.15 DutyCycleMax

This parameter is used to specify a Maximum Duty Cycle for the Fan Controller. In some cases, fans have a high-end duty cycle range that delivers little additional airflow benefit yet products significantly higher acoustics. This parameter allows this range to be avoided. Note that, in cases where all fans are overridden to their maximum speed to address a critical situation, this parameter is not enforced. Values 0 – 10000 will be used to specify duty cycle percentages in the range 0 – 100 percent (i.e. specified in hundredths of a percent)

A.4.7.16 stAmbientFloor/stAmbientCeiling

These sets of parameters specify the implementation of limiters on the duty cycle of the fan controller, based upon particular ambient temperature inputs. The stAmbientFloor parameter set implements a limit on the minimum duty cycle while the stAmbientCeiling parameter set implements a limit on the maximum duty cycle.

Structure QST_AMBIENT_LIMITER is defined as follows:

```
typedef struct _QST_AMBIENT_LIMITER
{
   UINT8                        uTempMonitor;
   INT32F                       lfDutyCycleRange;
   INT32F                       lfTempMin;
   INT32F                       lfTempRange;

} QST_AMBIENT_LIMITER;
```

A.4.7.16.1 uTempMonitor

This parameter is used to specify the index of the Temperature Monitor whose temperature readings are to be used to implement ambient temperature floor/ceiling limiting. This is the process of limiting the fan controller's minimum/maximum duty cycle based upon an ambient temperature. If ambient floor limiting is not being implemented, specify "None" (0xff) for this parameter.

A.4.7.16.2 lfDutyCycleRange

This parameter is ignored if the corresponding uTempMonitor parameter is set to "None" (0xff). It specifies the extent of the duty cycle range over which ambient temperature floor/ceiling limiting will occur.

For ambient floor limiting, the lower limit of this range is the current Minimum Duty Cycle value; this is initially the value specified by the DutyCycleMin parameter, but this value may be adjusted by the Minimum RPM feature.

For ambient ceiling limiting, the upper limit of this range is the current Maximum Duty Cycle value; this is the value specified by the DutyCycleMax parameter.

A.4.7.16.3 lfTempMin

This parameter is ignored if the corresponding uTempMonitor parameter is set to "None" (0xff). It specifies the lower limit of the ambient temperature range that is used for ambient temperature floor/ceiling limiting. The extent of this range is specified by the corresponding lfTempRange parameter.

A.4.7.16.4 lfTempRange

This parameter is ignored if the corresponding uTempMonitor parameter is set to "None" (0xff). It specifies the extent of the ambient temperature range that is used for ambient temperature floor/ceiling limiting. The lower limit of this range is specified by the corresponding lfTempMin parameter.

A.4.7.17 ResponseWeighting[]

These parameters specify the weightings that are to be applied against the temperature response (duty cycle delta) values calculated by the various Temperature Response Units. The weighted response with the largest positive

magnitude (or smallest negative magnitude, if all are negative) will be applied against the previous duty cycle for the connected fan(s).

Specify these parameters using values in the range 0 to 100 (percent). Values 0 – 10000 will be used to specify duty cycle percentages over this range (i.e. specified in hundredths of a percent).

The weighting values need to be chosen carefully, taking into account the level of contribution that the associated fan(s) have on the situation at each temperature monitoring point, as well as the acoustic cost of using particular fans at different duty cycle levels. The overall goal is to balance the use of all of the available fans such that temperature levels are maintained at the lowest overall acoustic cost. See the Configuration and Tuning Manual for more information.

§

Appendix B Compacting/Expanding Intel® QST 2.0 Configuration Payloads

B.1 Introduction

Prior to 2.0, Intel® QST Configurations contained a static number of entities of each type, regardless of how many of these entities were actually used. Each configuration contained 12 Temperature Monitors, 8 Fan Monitors, 8 Voltage Monitors, 12 Temperature Response Units and 8 Fan Controllers. Beginning with Intel® QST 2.0, however, support has been added for much larger Workstation/Server configurations. Each configuration may now contain as many as 32 Temperature Monitors, 32 Fan Monitors, 32 Voltage Monitors, 32 Current Monitors, 32 Temperature Response Units and 32 Fan Controllers. To make them more manageable, configuration files are no longer required to contain entries for every single entity. Instead,

- Only those entities that are actually being used (enabled) are required to be present. Unused (disabled) entries are not required to be present. The (binary) configuration payload structure transitions from being a static set of structure arrays for each entity type to being a list of entity structures. The order of these structures remains important, however; necessary Temperature Monitor structures must be followed by necessary Fan Monitor structures, then Voltage Monitor structures, then Current Monitor structures (if any), Temperature Response structures and, finally, Fan Controller structures).

- Fan Controller entities are not required to include weighting values for all 32 possible Temperature Response Units. The number of weighting values specified need only to include up to the last non-zero Weighting. For example, if Temperature Responses 1, 2 and 6 have non-zero Weightings, values would need to be specified for Temperature Responses 1 through 6. Within the (binary) configuration payload, the number of Weightings included in a structure will be indicated/determined using the StructLength field in the structure header.

- The number of entities of each type that is supported by a particular build of the Intel® QST firmware may vary. A firmware build for a Workstation board could support significantly more entities that a build for a Desktop board.

Note: The Workstation and Desktop firmware builds for the Intel® 5 Series Chipset have been configured identically; support is included for 12 Temperature Monitors, 8 Fan Monitors, 8 Voltage Monitors, 0 (zero) Current Monitors, 12 Temperature Response Units and 8 Fan Controllers. This is not expected to be the case for future chipsets, however.

B.2 Working with Intel® QST 2.0 Configurations

While it is certainly possible to design software the works with Intel® QST configurations that exactly match the hardware requirements of a particular design, most software will be designed to maximize reusability and thus will provide support for handling configurations of any size, from smallest to largest. The well-designed software application will thus support the representation of (binary) configuration payloads that include the maximum number of entities.

As mentioned, the size-optimized representation of the configuration payload consists of a list (stream) of entity structures. Because the number of entities of each type that are included in a payload is unknown, the structures must typically be processed sequentially. Using a payload that includes support for the maximum number of entity structures can provide the additional benefit of allowing indexed access to the entity structures.

Because the Intel® QST firmware must save its active configuration to flash (a limited commodity), the size of the configuration payload actually sent to the firmware should be kept as small as possible. Thus, the application requires support for compacting the configuration payload before sending it to Intel® QST. Similarly, when extracting the current configuration from Intel® QST, the application requires support for expanding the payload to include the maximum number of entity structures, in order to support indexed access to it.

The Intel® QST Software Development Kit (SDK) provides source code and header files for a pair of routines that implement support for compacting and expanding configuration payloads. Subsequent sections of this appendix detail how to use these functions. Example code showing how to (1) extract the current configuration from Intel® QST and expand it into a full-sized buffer and (2) compact a configuration in a full-sized buffer and deliver it to Intel® QST is presented in Section 3.3.7 of this document…

B.3 Function Reference

B.3.1 CompactConfig()

This function is used to compact an Intel® QST (binary) configuration payload to the smallest size possible. The resulting payload's contents will be limited to structures for only those entities that are enabled. Fan Controller structures will be made as small as possible, limiting the Weightings included to the smallest non-zero subset.

Invocation:

```
#include "QstCompactConfig.h"

MANIP_STATUS CompactConfig
(
    IN    void      *pvExpConfig,
    IN    UINT32    uExpConfigSize,
    OUT   void      *pvCompConfig,
    IO    UINT32    *puCompConfigSize
);
```

Input Parameters:

pvExpConfig	Pointer to the buffer containing the Intel® QST configuration payload that is to be compacted.
uExpConfigSize	Specifies the size, in bytes, of the Intel® QST configuration payload that is to be compacted.

Output Parameters:

pvCompConfig	Pointer to the buffer into which the compacted Intel® QST configuration payload will be stored.

Input/Output Parameters:

puCompConfigSize	Pointer to the variable that, on input, specifies the size, in bytes, of the buffer into which the compacted Intel® QST configuration payload will be stored. On output, the size of the resulting (compacted) Intel® QST configuration payload will be written to this variable.

Returns:

The function returns one of the following status values:

MANIP_SUCCESS – The function successfully compacted the input configuration payload and stored it in the output buffer.

MANIP_INVALID_PARAMETER – One or more of the pointers passed is invalid (set to NULL).

MANIP_BUFFER_TOO_SMALL – The size of the input buffer is too small to possibly represent a valid configuration or the output buffer is too small to receive the compacted configuration.

MANIP_INVALID_HEADER – The payload header in the input buffer is invalid or represents a Intel® QST 1.x payload, which cannot be compressed.

MANIP_INVALID_CFG_FORMAT – One of more entity structures in the payload in the input buffer is invalidly formatted.

B.3.2 ExpandConfig()

This function is used to expand an Intel® QST (binary) configuration payload to a specific size. The resulting payload will contain the specified number of entities of each particular type. Its Fan Controller entity structures will be expanded to contain the same number of weightings as there are Temperature Response structures in the resulting payload.

Invocation:

```
#include "QstExpandConfig.h"

MANIP_STATUS ExpandConfig
(
    OUT  void                *pvExpConfig,
    IN   UINT32              uExpConfigSize,
    IN   QST_CFG_COUNTS      *pstCfgCounts,
    IN   void                *pvCompConfig,
    IN   UINT32              uCompConfigSize
);
```

Input Parameters:

uExpConfigSize	Specifies the size, in bytes, of the buffer that will receive the expanded Intel® QST configuration payload.
pstCfgCounts	Pointer to a structure that specifies how many entities of each type should be included in the expanded Intel® QST configuration payload. The QST_CFG_COUNTS structure, which is defined in the associated header files, specifies entity counts in the following order: Temperature Monitors, Fan Monitors, Voltage Monitors, Current Monitors, Temperature Response Units and Fan Controllers.
pvCompConfig	Pointer to the buffer containing the Intel® QST configuration payload that will be expanded.
CompConfigSize	Specifies the size, in bytes, of the Intel® QST configuration payload that will be expanded.

Output Parameters:

pvExpConfig	Pointer to the buffer into which the expanded Intel® QST configuration payload will be stored.

Returns:

The function returns one of the following status values:

MANIP_SUCCESS – The function successfully expanded the input configuration payload and stored it in the output buffer.

MANIP_INVALID_PARAMETER – One or more of the pointers passed is invalid (set to NULL).

MANIP_BUFFER_TOO_SMALL – The size of the input buffer is too small to possibly represent a valid configuration or the output buffer is too small to receive the expanded configuration.

MANIP_INVALID_HEADER – The payload header in the input buffer is invalid or represents an Intel® QST 1.x configuration payload, which cannot be expanded.

MANIP_INVALID_CFG_FORMAT – One of more entity structures in the input Intel® QST configuration payload is invalidly formatted.

§