QUICKSENSE™ FIRMWARE API

1. Introduction

This document describes the QuickSense™ Firmware API and outlines the process of using the API to develop capacitive sensing firmware systems.

This document cover the following:

- Firmware API description
- Guide to configuring the API for use in a firmware system
- Complete specifications for the QuickSense Firmware API and the API's serial interface

For information about the QuickSense Firmware API's method for baselining, please see Application Note 418. For a detailed discussion on setting thresholds and SNR measurement, please see Application Note AN367.

2. Terminology

In this application note, the following definitions apply:

- Channel— port pin to be measured by MCU capacitive sensing hardware.
- Threshold—a system configuration set point that allows a change in parametric state when crossed by a channel's measured value.
- Threshold state— description of how a channel's current value falls in relation to that channel's defined thresholds.
- Group—a binding of channels into a single input object such as a slider or control wheel.
- Group position—an interpretation of a group's bound channel values to determine a finger's placement.

3. QuickSense Firmware API Overview

The QuickSense Firmware API performs the following tasks:

- Captures and stores capacitive measurements from multiple input channels.
- Computes threshold state information on enabled input channels.
- Compensates for changes to environmental conditions using a method of baselining.
- Allows users to bind input channels into groups such as control wheels and sliders.
- Provides a high-level, easy-to-use set of routines and variables that can be accessed by the firmware's application layer.
- Abstracts low-level, MCU-specific routines so that the API can be easily expanded to include other MCU families and other capacitive sensing methods.
- Creates a serial interface that enables users to output channel values, thresholds, and group values and adjust threshold levels without recompiling code using the Human Interface Studio.

This version of the QuickSense Firmware API does not include any algorithms that filter noise from capacitive sensing samples. Firmware system performance may be compromised in an electrically noisy environment."

Figure 1, "Basic Data Flow Diagram," shows the data flow diagram of the QuickSense Firmware API. The sections that follow give a brief description of the Figure 1 diagram. For a more detailed specification of all of the QuickSense Firmware API's functions, variables, and definitions, see section "6. Human Interface Serial Interface (HISI) version 1.1."

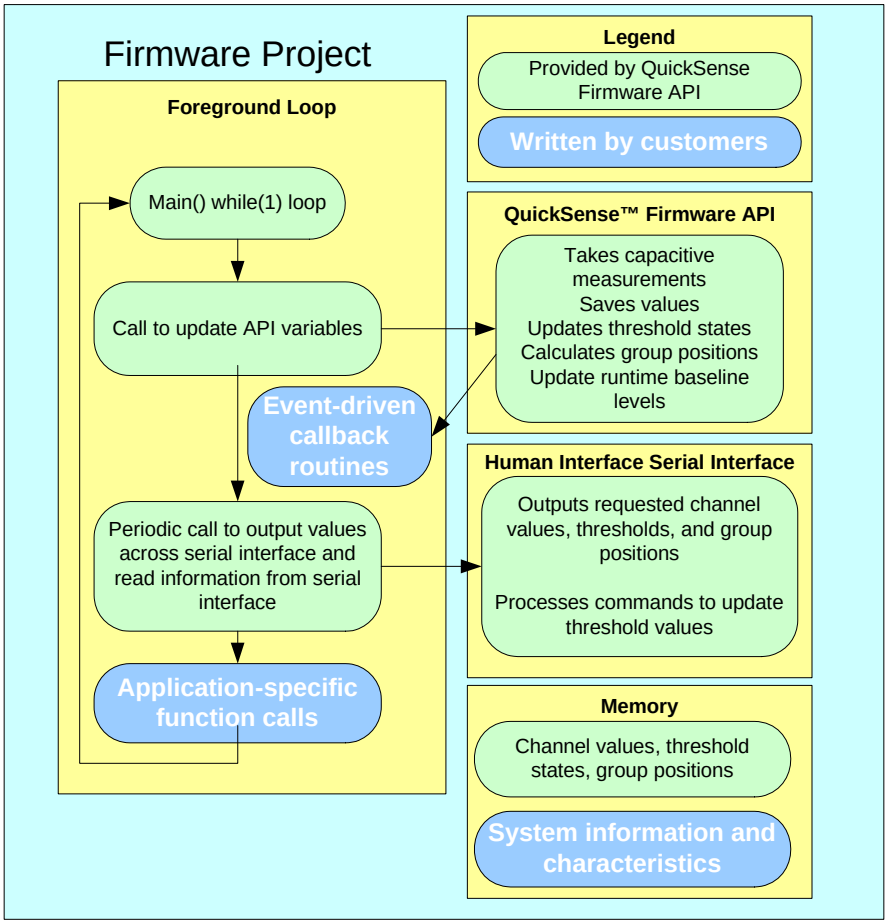

Figure 1. Basic Data Flow Diagram

3.1. System Information and Characteristics

In order to use the QuickSense Firmware API, customers must fill in configuration files with information about the capabilities of their application. This information includes the number of channels being measured, whether threshold state detection is supported, and group position information. Based on information included in the project's build, definitions and pre-compiler directives will size all data arrays appropriately and include only routines that are necessary for the customer's project to execute correctly. For a step-by-step guide to configuring a system, see section “4. Designing with the API.”

3.2. Capturing Data

The QuickSense Firmware API measures capacitance using Silicon Laboratories MCU hardware such as the C8051F70x family's capacitance sensing peripheral. Abstraction methods employed in the API separate higher-level data collection and storage from the low-level physical interface. Once channel data is captured, the Capacitance Sensing API performs calculations on the data, including threshold state updates and group position updates.

3.3. Accessing Data

Captured data and data generated by performing calculations on captured data are stored in arrays located in on-chip RAM. This data can be accessed through memory access routines found in the API. The data can also be accessed through explicit array reads and writes in firmware.

3.4. Outputting Data

The QuickSense Firmware API offers a serial interface for transmitting data out of the device. This interface enables the device to act as a target to a serial interface master. The master device can request data through a simple command interface. After receiving the request, the target device transmits data at a period defined by the master device. The Human Interface Studio uses this serial interface to read the data it displays onscreen.

Note that in addition to outputting values, the serial interface can also be used to configure device threshold levels.

For a detailed description of the serial interface command set, see section “6. Human Interface Serial Interface (HISI) version 1.1.”

3.5. Thresholds

The QuickSense Firmware API has the ability to compare measured channel data against configured threshold levels. Based on the comparison of measured data and defined levels, the API sets the channel's threshold state to one of four values. These states give the application layer a way of monitoring channel values without manually examining values each time they are measured. Callback functions for rising edge events (QS_RisingEdgeEvent()) and falling edge events (QS_FallingEdgeEvent()) give the application layer a way to respond quickly and efficiently to cases where values fall above or below defined thresholds.

Figure 2 shows the four defined threshold states and the points where the QuickSense Firmware API calls rising and falling edge event functions.

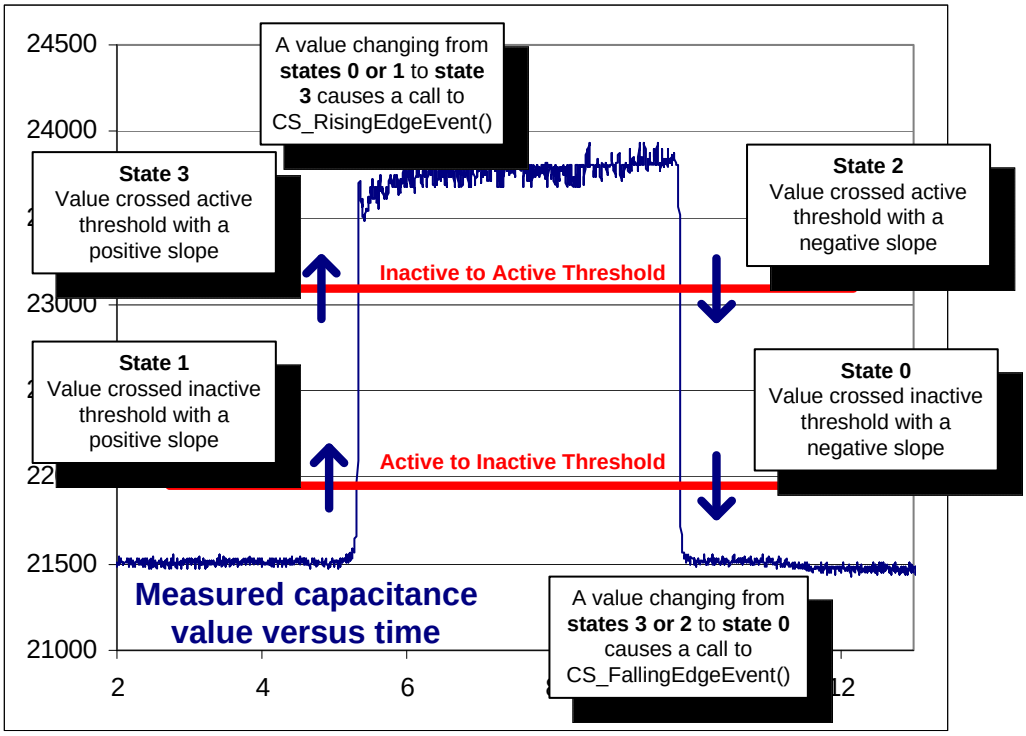

Figure 2. Threshold States and Rising and Falling Edge Events

In order for a system to monitor a channel for threshold state detection, that channel must be configured to allow threshold detection, and the proper threshold values must be saved to the device. For information about how to configure a device's channels for threshold state detection, see section “4. Designing with the API.”

3.6. Groups

Groups are bound collections of channels whose values are input into algorithms to determine finger position information. The two defined group types for the QuickSense Firmware API are sliders and control wheels, which are shown in Figure 3. Please note that the QuickSense Firmware API supports 4-channel sliders and control wheels.

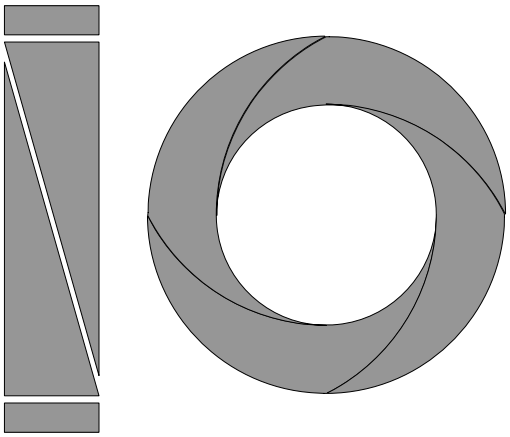

Figure 3. Sliders and Control Wheels Compatible with the API

When a user places a finger on a group, the QuickSense Firmware API algorithm determines the finger's position on the group's pads and outputs a value. A slider's position is defined on a scale of 0 to 511, and a control wheel's position is defined as 0 to 359.

In addition to outputting a value, the QuickSense Firmware API offers three callback functions that allow the application layer to respond quickly to group events. The callback routines provide the application layer with the following:

- QS_RisingGroupEvent()— A new finger press detected on a group
- QS_GroupEvent()— An ongoing finger press on a group
- QS_FallingGroupEvent()— A finger de-presses from a group

A typical sequence of callback routines for a group press goes as follows:

- A finger presses down on a group, and the algorithm calls the rising group event.
- As a finger remains on the group, producing updated group position values, the API calls Group Event once per group position update.
- When the finger is lifted from the group, the API calls falling group event.

4. Designing with the API

The following sections describe the process of setting up the QuickSense Firmware API in a firmware system. This document describes the method by which the API can be customized through code editing. Users can also use the QuickSense Firmware API Configuration Wizard to generate a customized firmware system. This application is available for download on the Silicon Laboratories web page.

4.1. Design Checklist

The QuickSense API implements much of its code within conditional compilation statements. These statements enable the API to include only the code that is needed. Review the checklist below to define the characteristics of the system that is to be created.

1. Which MCU will be used in the system?
2. How many input channels does the system have?
3. What is the hardware multiplexer setting for each channel to be measured?

Note: See the MCU data sheet for multiplexer settings.

4. Do any of the channels require threshold detection? If so, which?
5. Is the serial interface needed by the application? Will the application be communicating with the Human Interface Studio?
6. How many sliders are defined in the application? How many control wheels?
7. Does the system need to respond to threshold detection rising edge or falling edge events?
8. Does the system need to respond to group events?
9. What is the frequency that each channel's value will be updated?
10. What baselining configuration works best for the system?

The sections that follow guide designers through the process of entering the system information into the API's configuration files.

4.2. Configuration Files

The configuration files named QS_Config.h and QS_Config.c allow the designer to tailor the API to suit the characteristics of a system. The QS_Config.h header file contains the definitions that must be set by the user, and the QS_Config.c file contains the non-volatile, code space-based arrays that must be defined. The sections below describe each definition and array, and also point out dependencies between different items.

4.3. QS_Config.h

After all configuration settings have been included in QS_Config.c and QS_Config.h, application layer code can be added to the main() routine, any included callback functions, and elsewhere in the project. Capacitive sensing data can be accessed through application layer code using routines defined in section "5.2.1. Memory Access Routines."

4.3.1. BoardID

Description: If the firmware will be using the serial interface to communicate with the Human Interface Studio, then the system needs to have a defined board identification value. The Human Interface Studio uses this value to determine if the graphical interface can display a custom board layout screen, or if it must display a generic screen to show values and threshold information. All firmware systems created by customers should use the board ID of '0xFF' to force the Human Interface Studio to use the generic display method.

4.3.2. NUM_OSC_CHANNELS

Can be set to: Numeric value

Description: This definition describes the number of input channels that will use the relaxation oscillator method of capacitive sensing.

Notes: This method is currently only supported in low-level API routines for the C8051F93x/92x family of MCUs.

4.3.3. NUM_CS0_CHANNELS

Can be set to: Numeric value

Description: This definition describes the number of input channels that will use the CS0 method of capacitive sensing.

Notes: This method is currently only supported in low-level API routines for the C8051F70x family of MCUs.

4.3.4. NUM_CONTROLWHEELS

Can be set to: Numeric value

Description: This definition describes the number of control wheels that will be measured by the API.

Notes: When this number is not set to 0, control wheel information must be included in QS_GroupType and QS_GroupChannelAssignmentTable.

Because groups use channel information and threshold information to calculate group position, if control wheels are defined in an application, CHANNEL_VALUE_SAVE and THOLD_VALUE_SAVE must be set to “ENABLED”.

4.3.5. NUM_SLIDERS

Can be set to: Numeric value

Description: This definition describes the number of sliders that will be measured by the API.

Notes: When this number is not set to 0, slider information must be included QS_GroupType and QS_GroupChannelAssignmentTable.

Because groups use channel information and threshold information to calculate group position, if sliders are defined in an application, CHANNEL_VALUE_SAVE and THOLD_VALUE_SAVE must be set to “ENABLED”.

4.3.6. CHANNEL_VALUE_SAVE

Can be set to: ENABLED or DISABLED

Description: This definition determines whether an array will be allocated in memory to store measured channel values.

Notes: If the serial interface is being used to connect to the Human Interface Studio, then channel values must be saved to memory. In this case, CHANNEL_VALUE_SAVE must be set to ENABLED.

If groups are defined within the API, then CHANNEL_VALUE_SAVE must be set to ENABLED.

4.3.7. GROUP_VALUE_SAVE

Can be set to: ENABLED or DISABLED

Description: This definition determines whether an array will be allocated in memory to store measured group positions.

4.3.8. THOLD_VALUE_SAVE

Can be set to: ENABLED or DISABLED

Description: This definition determines whether an array will be allocated in memory to store measured threshold states.

Notes: If groups are defined within the API, then THOLD_VALUE_SAVE must be set to ENABLED.

4.3.9. SERIAL INTERFACE

Can be set to: ENABLED or DISABLED

Description: This definition determines whether routine and variables used to create the serial interface will be included in the build.

4.3.10. MCU

Can be set to: F700 or F930

Description: This definition determines which low-level routines should be included in the API's build.

4.3.11. SWITCH_EVENTS

Can be set to: RISING, FALLING, RISING_AND_FALLING, DISABLED

Description: This definition determines what types of threshold state-related callback routines can be called by the API whenever threshold events are detected.

4.3.12. GROUP_EVENTS

Can be set to: RISING, FALLING, RISING_AND_FALLING, GROUP_ONLY, DISABLED

Description: This definition determines what types of group position-related callback routines can be called by the API whenever group events are detected.

4.3.13. QS_THRESHOLD_LOCATION

Can be set to: Numeric value

Description: This definition determines where in code space threshold level information will be stored.

Notes: Threshold state information must be stored on a page or pages of code space that do not contain executable code.

4.3.14. CHANNEL_UPDATE_RATE

Can be set to: Numeric value

Description: This definition is a user-provided estimate on the frequency that channels will be updated by the system. The baselining algorithm bases its Baseline Update Rate on this value.

Notes: This definition does not control the channel update rate. The channel update rate must be controlled through application layer code. In most applications, the channel update rate equals the frequency that the application layer calls QS_UpdateChannels().

4.3.15. ACTIVE_BASELINE_DECAY_PERCENT

Can be set to: Numeric value

Description: This value controls the minimum level that the runtime active baseline can fall to. For more detailed information on this definition, please see AN418.

4.3.16. HISM

Can be set to: Numeric value

Description: The High Inactive Sensitivity Margin (HISM) sets the threshold above which the baselining algorithm assumes that a channel could be active. Values passed the HISM threshold will not be used to update that channel's runtime inactive baseline. For more information about this definition, please see AN.

4.3.17. LISM

Can be set to: Numeric value

Description: The Low Inactive Sensitivity Margin (LISM) sets the threshold below which the baselining algorithm forces a runtime baseline update. For more information about this definition, please see AN.

4.4. QS_Config.c

The declarations listed below are included in QS_Config.c.

4.4.1. QS_ChannelInfo

Can be set to: One array element per channel, with each element including the capacitance sensing method and the threshold detection type. The defined methods are CS0 and OSC, and the defined threshold types are NO_THRESHOLD_DETECT, THRESHOLD_NOT_MODIFIABLE, THRESHOLD_MODIFIABLE, and GROUP_CALIBRATION.

Description: This array describes each of the defined channels being measured. The API requires that designers describe the characteristics of each channel so that the channel value measurements and other functions will know which lower-level routines to call to execute correctly.

To define a channel's characteristics, include information as an array element that corresponds to that channel. For example, a system might have two defined channels, one that does not need to have threshold detection supported, and one that does. In this example, QS_ChannelInfo would be defined as follows:

```
code U8 QS_ChannelInfo[NUM_CHANNELS] =

{

CS0 | NO_THRESHOLD_DETECT,

CS0 | THRESHOLD_MODIFIABLE,

};
```

Notes: The number of channels defined in QS_ChannelInfo must equal the sum of NUM_OSC_CHANNELS and NUM_CS0_CHANNELS.

Channels that will be bound to groups must have a threshold detection type defined as "GROUP_CALIBRATION".

4.4.2. QS_MuxInput

Can be set to: One array element per channel, with each element corresponding to the input multiplexer setting needed by the MCU hardware

Description: This definition describes the input multiplexer configuration for each channel being measured by the API.

Notes: The number of channels defined in QS_MuxInput must equal the sum of NUM_OSC_CHANNELS and NUM_CS0_CHANNELS.

4.4.3. UpdateFrequency

Can be set to: Two numeric values, one for the slowest possible update frequency for the serial interface, and one for the highest possible update frequency. Each value can be within the range of 1-100.

Description: This two-byte array defines how quickly or slowly data can be retrieved by the Human Interface Studio. This allows designers to prevent the firmware performance from being affected by too many resources being devoted to data transfer.

4.4.4. QS_GroupType

Can be set to: One array element for each defined group, with values of SLIDER_TYPE or CONTROL_WHEEL_TYPE

Description: This array defines what type of group is being measured for the list of defined groups. Similar to QS_ChannelInfo, this array lets the API routines know which low-level routines should be used to calculate group position for each defined group.

Notes: The number of groups described in QS_GroupType must equal the sum of NUM_CONTROLWHEELS and NUM_SLIDERS

This array will be included in the project's build only if at least one group is defined in QS_Config.h.

4.4.5. QS_GroupChannelAssignmentTable

Can be set to: One element per defined group, with each element being a pointer to an array listing number of channels bound to that group and the sequence of channels to be bound to that particular group

Description: This array defines what channels make up a given control wheel or slider. Each slider or control wheel should have an array that lists all channels bound to that group. For example, a system might use a single control that is composed of channels 3, 4, 5, and 6. The array describing these channels might look as follows:

```
code U8 ControlWheel1[] =
{
 4, // Size
 3, 4, 5, 6 // Channels used
};
```

Note that the first value in the array equals the number of channels in the list that follows. This array would then be included in QS_GroupChannelAssignmentTable as follows:

```
code U8* QS_GroupChannelAssignmentTable[] =
{
 ControlWheel1
};
```

Note: The number of groups described in QS_ GroupChannelAssignmentTable must equal the sum of NUM_CONTROLWHEELS and NUM_SLIDERS.

Groups should be defined in the same order here as they are defined in QS_GroupType.

This array will be included in the project's build only if at least one group is defined in QS_Config.h.

Please note that channels bound to a group must be defined in a set order. Figure 4 shows the order in which channels need to be defined in an element of QS_GroupChannelAssignmentTable.

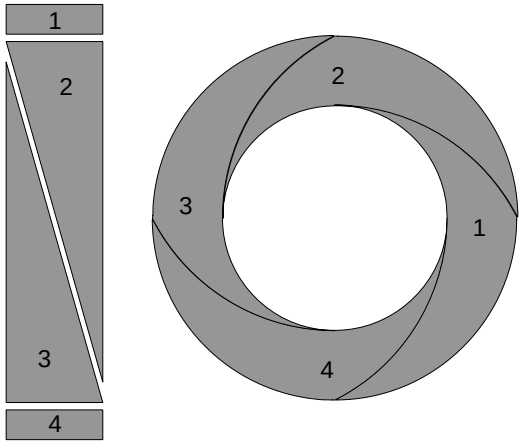

Figure 4. Slider and Control Wheel Channel Assignment Order

5. QuickSense Firmware API Specification

5.1. System Block Diagram

Figure 5 shows a block diagram of the QuickSense Firmware API.

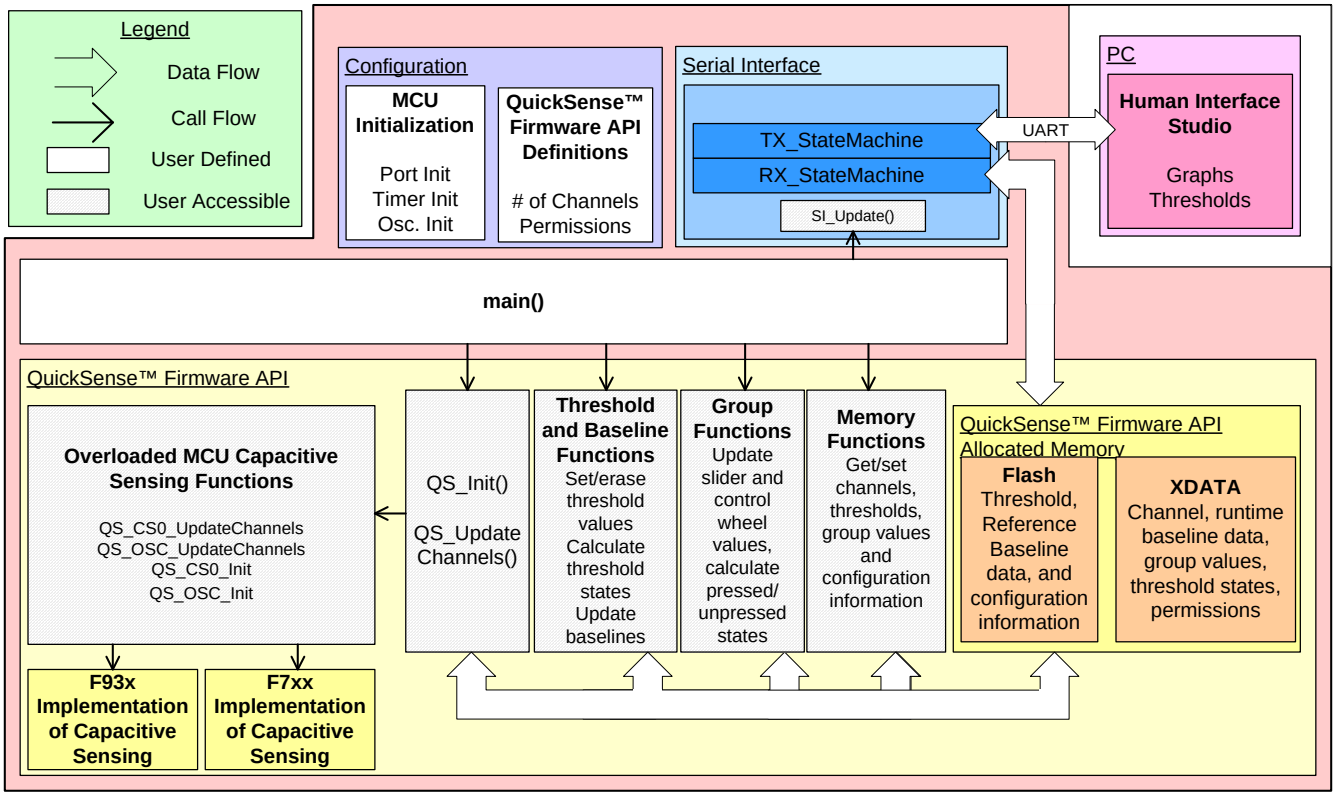

Figure 5. Block Diagram of Application using QuickSense Firmware API

The capacitive sensing routines take measurements on channels and will use interpretive functions to process the data and make updates to threshold states, slider values, and control wheel values. The information created and data gathered by these routines is then stored into memory. The application layer can gain access to this information by using memory access routines or by reading from memory locations directly.

5.2. API Functions

The following is the list of functions implemented by the API. Further details about each function will be explained in later sections. Figure 6 shows how the functions interact.

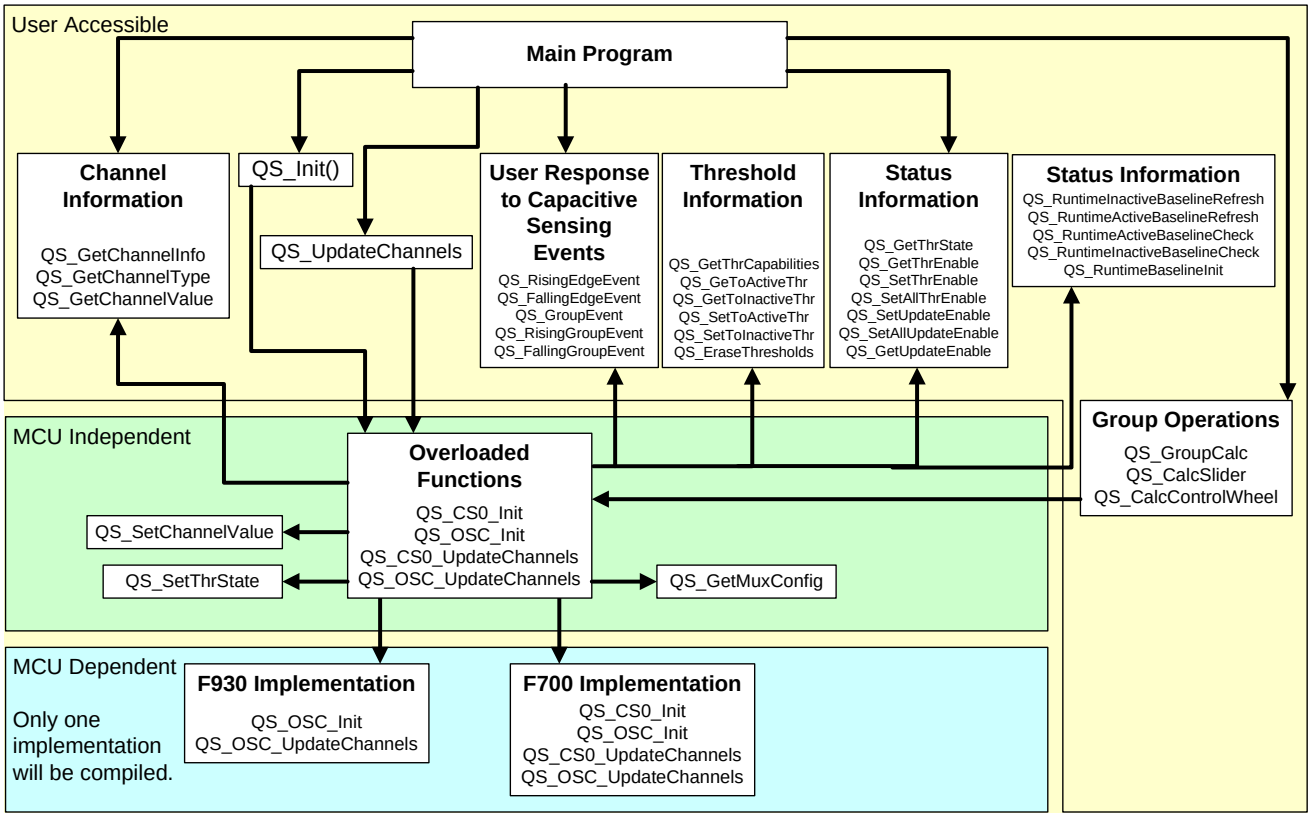

Figure 6. Function Call Graph

5.2.1. Memory Access Routines

The following routines are used throughout the API. They should also be used in the application layer of a project to easily retrieve data stored within the API. These routines are called within the API, but they can also be called by the application layer.

`QS_GetChannelInfo`	- Returns information about the channel
`QS_GetChannelValue`	- Returns the value measured at the channel
`QS_SetChannelValue`	- Writes value to channel's memory
`QS_GetChannelType`	- Returns the type of the channel
`QS_GetMuxConfig`	- Returns the mux configuration for the channel
`QS_GetThrCapabilities`	- Returns threshold detection information
`QS_GetToActiveThr`	- Returns inactive to active threshold value for a channel
`QS_GetToInactiveThr`	- Returns active to inactive threshold value for a channel
`QS_SetToActiveThr`	- Set inactive to active threshold value for a channel
`QS_SetToInactiveThr`	- Set active to inactive threshold value for a channel
`QS_EraseThresholds`	- Erase the pages containing thresholds
`QS_SetThrState`	- Update state of the channel with regards to thresholds
`QS_GetThrState`	- Returns state of channel with regards to thresholds
`QS_GetThrEnable`	- Return if threshold detection is enabled
`QS_SetThrEnable`	- Enable/Disable threshold detection for a channel
`QS_SetAllThrEnable`	- Enable/Disable threshold detection for all channels
`QS_GetUpdateEnable`	- Return if the channel is to be updated
`QS_SetUpdateEnable`	- Enable/Disable a channel to be updated
`QS_SetAllUpdateEnable`	- Enable/Disable all channels to be updated
`QS_SetGroupValue`	- Writes values to a group's memory
`QS_GetGroupType`	- Returns the type of the grouped channels
`QS_SetRuntimeInactiveBaseline`	- Updates a channel’s runtime inactive baseline
`QS_SetRuntimeActiveBaseline`	- Updates a channel’s runtime active baseline
`QS_GetRuntimeInactiveBaseline`	- Returns a channel’s runtime inactive baseline
`QS_GetRuntimeActiveBaseline`	- Returns a channel’s runtime active baseline
`QS_GetReferenceInactiveBaseline`	- Returns a channel’s reference inactive baseline
`QS_GetReferenceActiveBaseline`	- Returns a channel’s reference active baseline
`QS_SetReferenceInactiveBaseline`	- Updates a channel’s reference inactive baseline
`QS_SetReferenceActiveBaseline`	- Updates a channel’s reference active baseline
`QS_SetBaselineUpdateRate`	- Updates the runtime baseline update rate
`QS_GetBaselineStatus`	- Returns whether a channel’s baseline has been updated since last-HISI transfer
`QS_SetBaselineStatus`	- Sets or clears bit indicating a channel’s baseline update

5.2.2. Capacitive Sensing Routines

These routines initialize and control the QuickSense Firmware API's functionality. Routines on this list update channel values, threshold states, and group values. These routines should be called by the application layer when the API is operational.

`QS_Init`	- Initialize all enabled capacitive sensing peripherals
`QS_UpdateChannels`	- Update all capacitive sensing channels
`QS_CalcGroups`	- Updates group positions
`QS_CalcSlider`	- Updates a group position for a group defined as a slider
`QS_CalcControlWheel`	- Updates a group position for a group defined as a control wheel
`QS_ThresholdUpdate`	- Update the threshold state of a channel
`QS_ThresholdUpdateCheck`	- Check to see if a channel's threshold state can be updated

5.2.3. Device-Specific Routines

These routines call into low-level functionality that interacts with the MCU's capacitive sensing hardware. These routines are used internally within the API but should not be called by the application layer.

`QS_CS0_Init`	- Initialize the CS0 peripheral
`QS_OSC_Init`	- Initialize the comparator peripheral
`QS_CS0_UpdateChannels`	- Update all CS0 measured channels
`QS_OSC_UpdateChannels`	- Update all Oscillator measured channels
`FLASH_ByteWrite`	- Write a byte into flash
`FLASH_PageErase`	- Erase a page of flash

5.2.4. Baselining Routines

These routines maintain runtime inactive and active baseline levels for each channel. These values define average values for channels in inactive and active states. For more information on baselining, please see AN.

`QS_RuntimeBaselineInit`	- Initialize runtime baselines
`QS_RuntimeInactiveBaselineRefresh`	- Update all runtime inactive baselines
`QS_RuntimeActiveBaselineRefresh-`	- Update all runtime active baselines
`QS_RuntimeInactiveBaselineCheck-`	- Check a channel’s value, possibly force a baseline update
`QS_RuntimeActiveBaselineCheck-`	- Check a channel’s value, possibly force a baseline update
`QS_BaselineCalCheck`	- Return whether channel has stored calibration values

5.2.5. Support Routines

The routines below were created to minimize code size by encapsulating common functionality used in many higher-level functions. These routines should not be called by the application layer.

`GetMask`	- Return the mask for an encoded variable
`GetLeftShift`	- Get the number of places to shift the mask
`GetVal`	- Get channel info from an encoded variable
`SetVal`	- Set channel info of an encoded variable

5.2.6. User-Implemented Routines

These routines are call-back functions that are called when the QuickSense API detects an event such as a channel threshold crossing or a slider or control wheel press. When enabled by the API's configuration files, customers should add application-specific code to the bodies of these functions in order to respond to these events. For more information, see section “3.5. Thresholds.” and section “3.6. Groups.”

`QS_RisingEdgeEvent`	- Executes on rising edge threshold events, passing into the routine the channel that caused the event, and that channel's value
`QS_FallingEdgeEvent`	- Executes on falling edge threshold events, passing into the routine the number of the channel that caused the event and that channel's value
`QS_GroupEvent`	- Executes when a slider or control wheel is being pressed, passing into the routine the number of the group that caused the event and that group's position
`QS_RisingGroupEvent`	- Executes when a slider or control wheel is first pressed, passing into the routine the number of the group that caused the event and that group's position
`QS_FallingGroupEvent`	- Executes when a slider or control wheel is de-pressed, passing into the routine the number of the group that caused the event and that group's position

5.3. Data Definitions

The QuickSense Firmware API uses three different data types:

- Non-volatile data that is not modifiable

Examples:

  - Information generated at compile time
  - Includes system parameters that never change, such as the number of capacitive sensing input channels
- Non-volatile data that is modifiable

Examples:

  - User-configurable information that must persist across power cycling
  - Includes capacitive sensing threshold values
  - This data type should be stored on a page of code space that does not include any other data, as this page will be erased and rewritten frequently
- Volatile data

Examples:

  - User-configurable or device-generated data
  - Includes capacitive sensing input channel measured values, threshold states, and threshold detect enabled/disabled settings

Figure 7 shows a diagram of all defined variables and definitions, where each is located in memory, and how functions interact with each variable. The subsections that follow cover different memory types and describe each variable and definition individually.

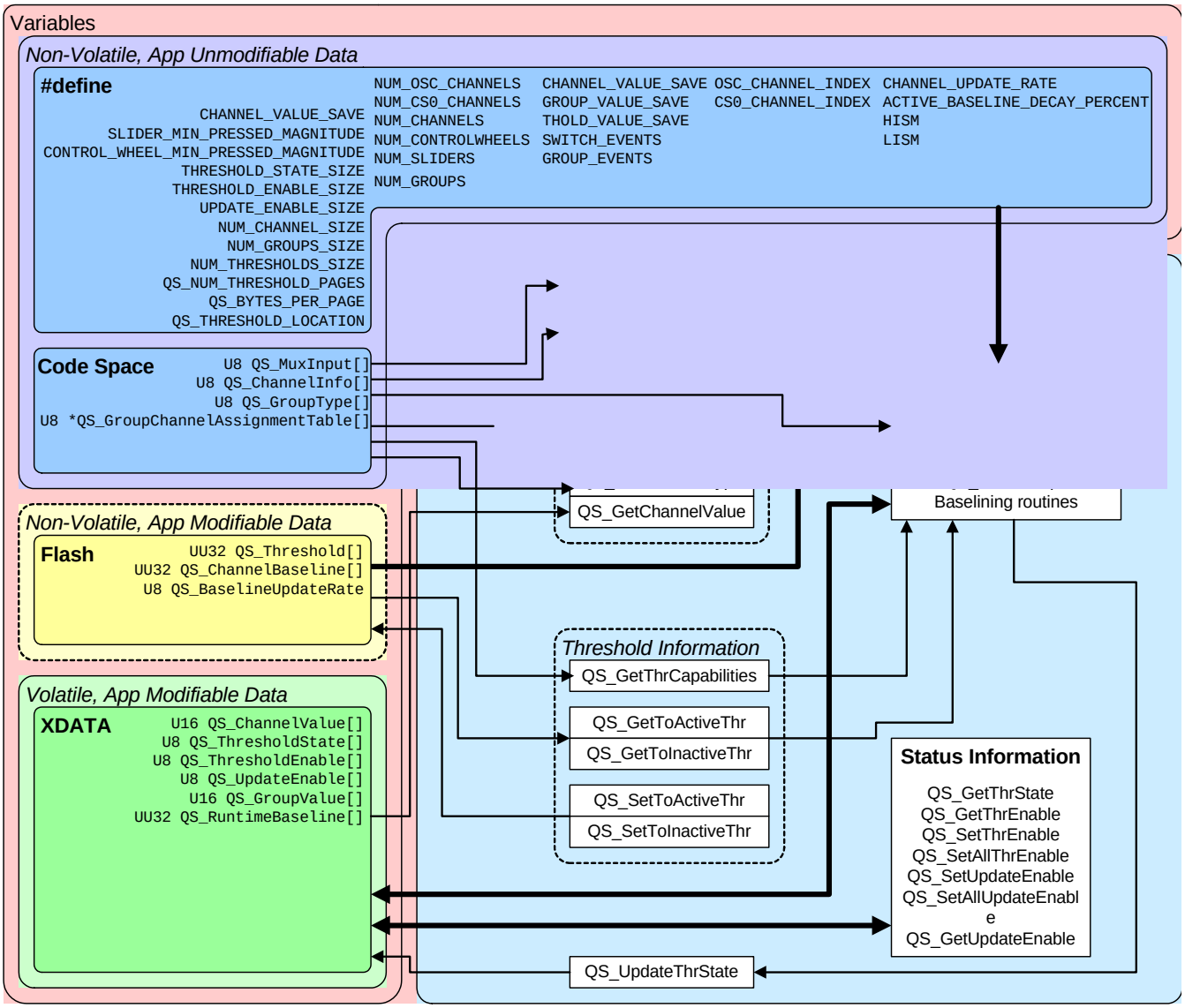

Figure 7. How Functions Interact with Memory

5.3.1. Non-volatile Data

Non-volatile data will be stored in a region of Flash memory. Non-volatile modifiable data is stored in an area referred to by the QuickSense Firmware API as the Non-Volatile Calibration and Configuration Area (NVCCA). This area includes modifiable data like thresholds and baseline information.

Other non-volatile data stored in Flash is defined at compile time and is not modifiable, such as channel information.

The following is a list of all variables stored in non-volatile space.

```
U8   QS_MuxInput[]                - Mux configuration for each channel
U8   QS_GroupType[]               - Defines the types of groups being used
U8*  QS_GroupChannelAssignmentTable - Maps channels to the groups
U16  QS_Threshold[]               - Threshold values for each channel
U8   QS_ChannelInfo[]             - Threshold Capabilities and other information
U32  QS_ChannelBaseline[]         - Stores reference baselines for all channels
U8  QS_BaselineUpdateRate         - Stores rate at which runtime baselines are updated
```

Figure 8 shows the array structures for variables stored in Flash.

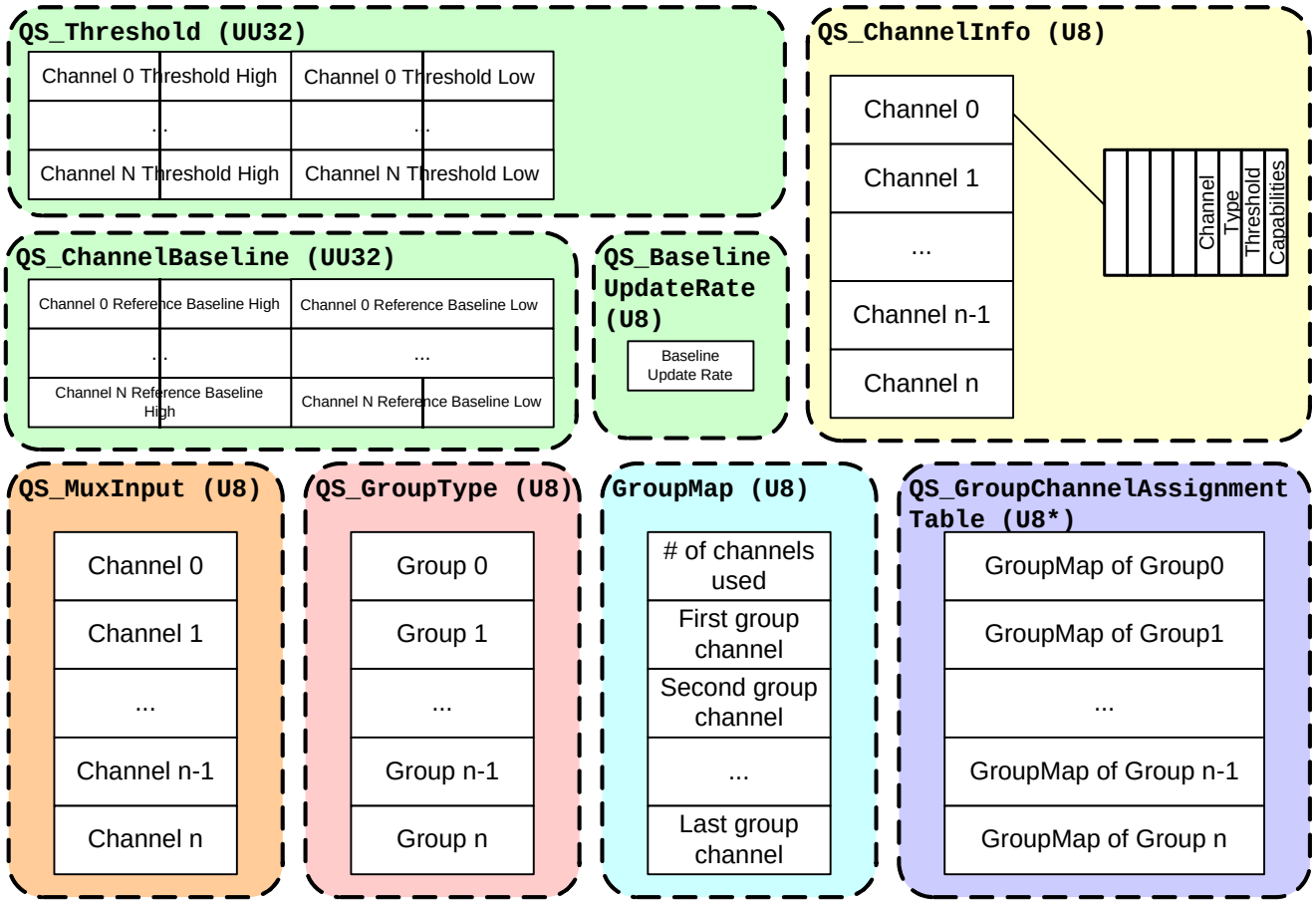

Figure 8. Flash Organization

5.3.2. Volatile Data

Volatile data is stored in MCU RAM. The type of data space used is set to XDATA by default. The following is a list of all volatile data stored in RAM:

```
U16 QS_ChannelValue[]          - Last measured value of the channel
U8  QS_ThresholdState[]        - Current threshold state of the channel
U8  QS_ThresholdEnable[]       - If threshold calculations are done on the channel
U8  QS_UpdateEnable[]          - If a measurement should be taken on channel
U32 QS_RuntimeBaseline[]       - saves runtime inactive and active baselines
```

Figure 9 shows the array structures for variables stored in RAM.

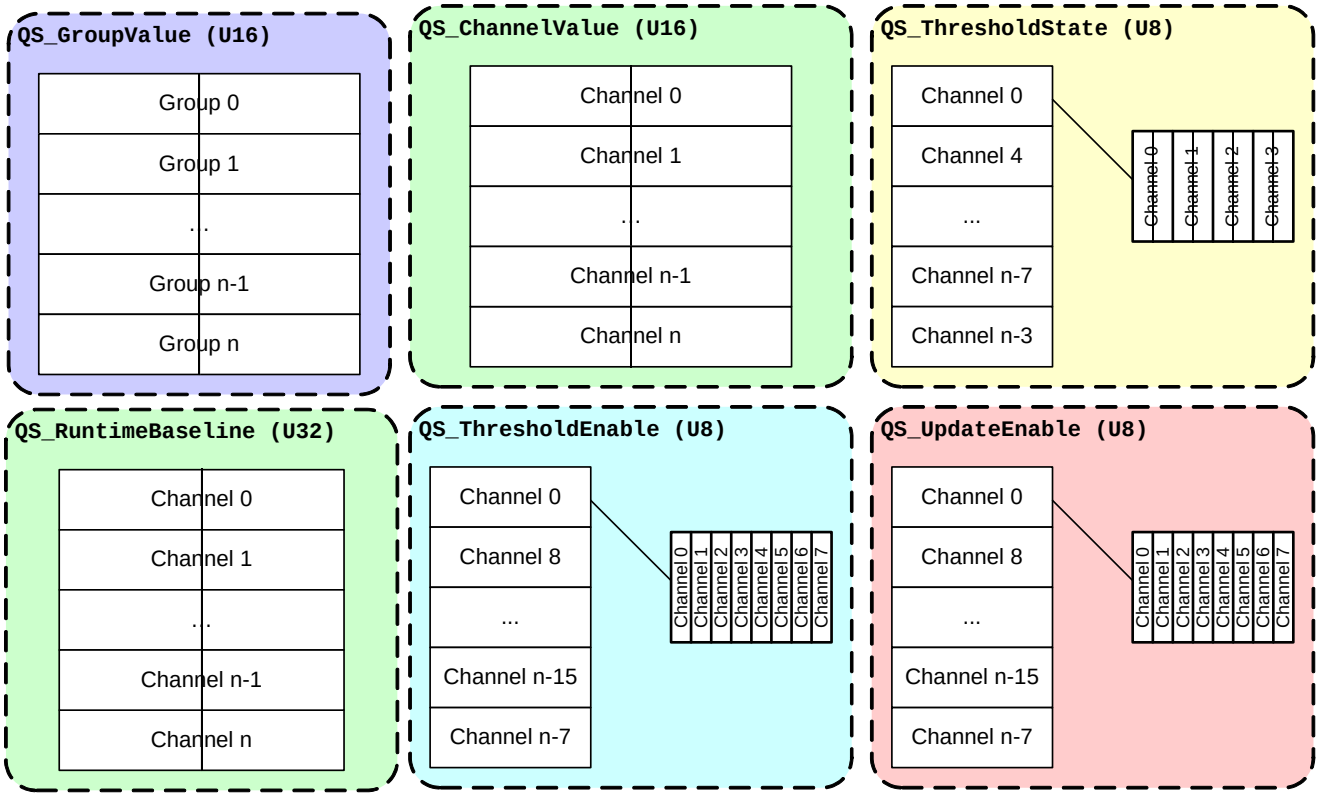

Figure 9. Data Organization

5.3.3. Array Formatting and Channel Numbering

All arrays containing information about channels are ordered identically, so that the same element index corresponds to the same channel for each defined array. Threshold arrays are formatted identically to channel arrays, so that a threshold array's element index equals the channel index.

To retrieve and update values in capacitive sensing arrays, we recommend that customers use the array access routines provided by the API, which are listed in section “5.2.1. Memory Access Routines.”

5.4. Definitions

The definitions listed below control capacitive sensing functionality and determine the size of arrays created in RAM and code space. Some of these values need to be modified as described in section “4. Designing with the API.”

5.4.1. Definitions that should be modified

NUM_OSC_CHANNELS	- Number of channels measured by Relaxation Oscillator
NUM_CS0_CHANNELS	- Number of channels measured by C2D
NUM_CHANNELS	- Total number of channels in the system
CHANNEL_VALUE_SAVE	- If measured values should be stored to memory
MCU	- Define which MCU is being used
SWITCH_EVENTS	- Enable/Disable rising/falling edge switch events
GROUP_EVENTS	- Enable/Disable group events
GROUP_VALUE_SAVE	- Enable/Disable the saving of intermediate group values
THOLD_VALUE_SAVE	- Enables/Disables thresholds comparison and allocation
NUM_CONTROLWHEEL	- Defines the number of control wheels being used
NUM_SLIDER	- Defines the number of sliders being used
NUM_GROUPS	- Defines the total number of grouped channels
QS_NVCCA_LOCATION	- Defines the starting location non-volatile storage space
ACTIVE_BASELINE_DECAY_PERCENT	- Defines minimum range active baseline can drop to
HISM	- High Inactive Sensitivity Margin for baselining
LISM	- Low Inactive Sensitivity Margin for baselining

5.4.2. Definitions that should not be modified

OSC_CHANNEL_INDEX	- Defines the starting index for OSC channels
CS0_CHANNEL_INDEX	- Defines the starting index for CS0 channels
QS_BYTES_PER_PAGE	- Defines the number of bytes per flash page
QS_NUM_THRESHOLD_PAGES	- Defines the number of pages thresholds are occupying
THRESHOLD_STATE_SIZE	- Defines the number of bytes reserved for threshold states
THRESHOLD_ENABLE_SIZE	- Defines the number of bytes reserved for threshold enable
UPDATE_ENABLE_SIZE	- Defines the number of bytes reserved for update enable
NUM_CHANNEL_SIZE	- Defines the size of the channel buffer in bytes
NUM_GROUPS_SIZE	- Defines the size of the group buffer in bytes
NUM_THRESHOLDS_SIZE	- Defines the size of the threshold buffer in bytes
SERIAL_INTERFACE	- Enable/Disable the serial interface

5.5. Files

The API includes many files that do not need to be modified along with a few files that will need modification in order to describe characteristics of a customer's application.

The following files do not need to be modified for most applications:

`QS_Global.h`	- Gives visibility to the project
`QS_Definitions.h`	- General and configured project definitions
`QS_Memory.h`	- Get/Set prototypes, memory visibility
`QS_Memory.c`	- Get/Set routines, memory allocation
`QS_Baselining.h`	- Baselining prototypes
`QS_Baselining.c`	- Baselining routines
`QS_CapacitiveSensing.h`	- Measurement/Interpretation prototypes
`QS_CapacitiveSensing.c`	- Measurement/Interpretation routines
`QS_Groups.h`	- Slider and control wheel prototypes
`QS_Groups.c`	- Slider and control wheel routines
`QS_MCURoutines.h`	- Overloaded functions
`QS_F700Routines.c`	- MCU specific implementations of overloaded functions
`SI_Core.h`	- Communication prototypes
`SI_Core.c`	- Communication routines
`SI_MCURoutines.c`	- Device specific communication protocols

The following files need to be modified:

`QS_Config.h`	- Contains control panel
`QS_Config.c`	- Code variables need to be filled

For details on how the two configuration files need to be modified, see section “3. QuickSense Firmware API Overview.”

5.5.1. File Structure

Figure 10 shows the file hierarchy of the API. To access API functionality, the user will only need to add the QS_Global.h header file to a project. This will automatically include the appropriate header files to make functions accessible to the project.

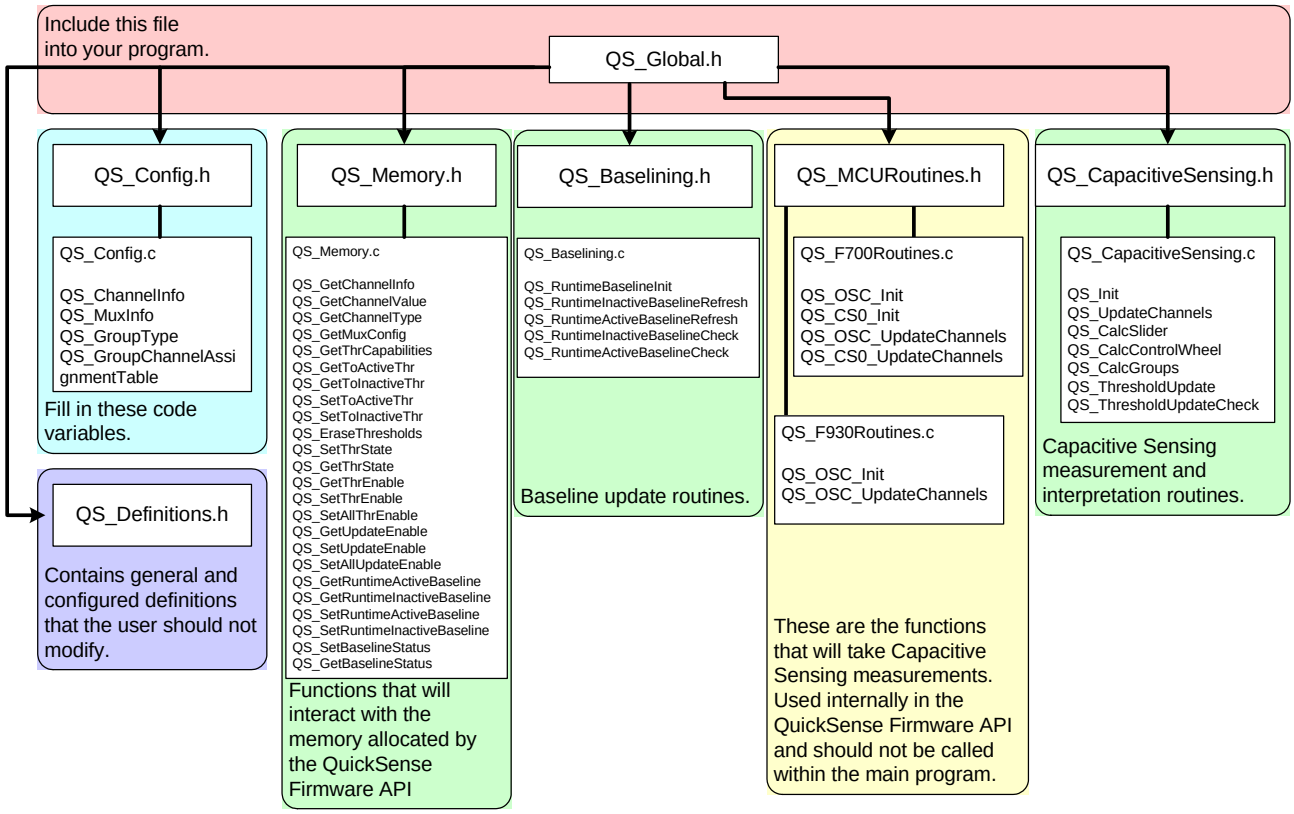

Figure 10. File and Functioning Mapping

5.6. MCU Resources Allocated to API

The amount of code space and RAM required by the API depends on the settings found in the configuration files during compile time. In addition to memory usage, the API uses the following hardware peripherals for operation:

- Timer 0—Provides timer to determine when to send capacitive sensing information out through the API
- Timer 1—Generates UART baud rate
- UART0—Enables serial communication with Human Interface Studio
- CS0—Converts capacitance found on input channels

6. Human Interface Serial Interface (HISI) version 1.1

Device targets running QuickSense Firmware API code and masters such as the Human Interface Studio communicate to each other through a serial interface. The interface allows the following:

- An enumeration-like process where the device describes its own capabilities
- Packetized data stream with structure that links channel identification to channel characteristics
- Command interface where the application can control, request, and update information through a set of commands
- All packets contain an 8-bit checksum

6.1. Packet Structure

All communication across this interface follows this structure:

- Command Identifier Byte plus upper 3 bits of 11-bit length field
- Lower 8 bits of length field
- 1-byte checksum

The bottom three bits of the command identifier byte hold additional length information and the remaining bits hold the actual command. This method allows for 32 separate commands and a data payload length of 2048, which provides the most flexibility in terms of defining new commands and scaling the amount of information to the number of switches.

Note that the length counts the number of bytes in the data payload and checksum, excluding the length and command bytes. For instance, a command that consists of a one byte data payload would have a length of 2:

Length = 1 byte data payload + checksum

Table 1 and Table 2 list all defined serial interface commands and responses.

Table 1. Defined Commands

Command ID	Command Name
0x00	Request Board ID
0x08	Start Enumeration
0x10	Set HISI Update Rate
0x18	Start Transfer
0x20	Stop Transfer
0x28	Enable Threshold Detect
0x30	Enable Threshold Detect All
0x38	Disable Threshold Detect
0x40	Disable Threshold Detect All
0x48	Set Inactive to Active Threshold
0x50	Set Active to Inactive Threshold
0x58	Get Threshold
0x60	Reset
0x68	Erase Non-Volatile Calibration and Configuration Area
0x70	Get Reference Baseline
0x78	Set Inactive Reference Baseline
0x80	Set Active Reference Baseline
0x88	Set Runtime Baseline Update Rate
0x90	Get Runtime Baseline Update Rate
0x98	Set Flash Key Codes
0xA0	Channel Calibration Check
0xA8	Get Firmware Revision and Capabilities

Table 2. Defined Responses

Command ID	Command Name
0x00	Board ID
0x08	Active Channel Information
0x10	Group Definition
0x18	Group Information
0x20	Threshold Information
0x28	HISI Update Frequencies
0x30	Enumeration Finished
0x38	Capacitive sensing Values
0x40	Capacitive sensing Group Values
0x48	Capacitive sensing Threshold States
0x50	Command Response
0x58	Get Threshold Response
0x68	Reference Baseline Information
0x70	Reference Baseline Response
0x78	Capacitive sensing Baseline Levels
0x90	Runtime Baseline Update Rate Response
0xA0	Channel Calibration Check Response
0xA8	Firmware Revision and Capabilities Response

6.2. Data Transmission Order

The serial interface transmits the following information, in the order that is shown below:

- Group position
- Threshold states
- Channel values

If any requested data is not available, a warning will be returned in the response packet and transmission will start with just the requested data that was available.

6.3. Calculating the Checksum

The checksum on a packet is generated by adding without carry all bytes of the packet, including the length and command bytes, excluding the checksum value. For a command with 1 byte of data payload, the checksum would equal:

Checksum = command ID + length byte + 1 byte data payload

6.4. Time-outs on Packets and Response Packets

The serial interface maintains a 500 ms time-out for each received command. Timeouts reset communications state machines after a set period of time if no data has been received. This gives both target and master a way to

recover from loss of synchronization. A target device that loses sync reverts back to the serial interface reset state.

All commands sent from a master to a target device elicit a target response command. Some master to target commands require that data be transferred in specific response commands. All other master to target device commands will cause a generic target device to master response command to be sent from target device to master. This generic response command has a byte of data payload containing the error condition associated with the master's request if an error occurred, and 0x00 otherwise. Errors will be discussed further in section “6.7.8. Command Response.”

6.5. Errors and Warnings

Table 3 contains the list of defined errors and warnings. Whenever an error occurs with the serial interface the associated byte should be sent with the command response packet. Warnings should not cause data to stop streaming. Errors should stop data from streaming.

Table 3. Error and Warning Codes

Warning/Error Code	Hex Value	Associated Message
0	0x00	No error or warning
1	0x01	ERROR: Bad checksum
2	0x02	ERROR: Unknown command
–3	0xFD	WARNING: Packet timeout
–4	0xFC	WARNING: Bad request
–5	0xFB	WARNING: Bad threshold

6.6. Detection and Enumeration Commands

When the master connects to a target device, a bi-directional sequence of commands executes that allows the master to determine whether the connected target device is part of a defined reference design board and which capacitive sensing inputs are enabled.

6.6.1. Request Board ID

Command ID: 0x00

Command Length: 1

Type: Command

Description: This command tells the target device to transmit back a Board ID.

Command Structure: Byte1: 0x00—Command ID

Byte 2: 0x01—Length

Byte 3: 0x01—checksum

Response: Target sends a Board ID command

6.6.2. Start Enumeration

Command ID: 0x08

Command Length: 1

Type: Command

Description: This command tells the target device to transmit back information about the target device's capabilities. Enumeration immediately disables all threshold detection on every channel.

Command Structure: Byte 1: 0x08—Command ID

Byte 2: 0x01—Length

Byte 3: 0x09—checksum

Response: Target sends all enumeration commands defined in firmware.

6.6.3. Board ID

Command ID: 0x00

Command Length: 2

Type: Response

Description: This command gives the master a Board ID number.

Defined Board IDs are:

0x02—F700 Music keyboard evaluation kit

0xFF—customer board

Command Structure: Byte 1: 0x00—Command ID

Byte 2: 0x0—Length

Byte 3: variable—Board ID

Byte 4: variable—checksum

6.6.4. Firmware Revision and Capabilities

Command ID: 0xA8

Command Length: 4

Type: Response

Description: This command gives the master the firmware revision number and describes other algorithmic capabilities of the firmware.

Information included in the data payload includes the following:

The firmware revision is a two-byte, BCD-encoded value. For firmware revision 1.1, the two byte revision number is 0x0110:

A byte describing the baselining method used in this version of firmware. For firmware revision 1.1, the baselining method used is defined as '1'.

Command Structure:

Byte 1: 0xA8—Command ID

Byte 2: 0x04—Length

Byte 3: 0x01—Firmware revision high byte

Byte 4: 0x10—Firmware revision low byte

Byte 5: 1—Baselining method used

Byte 6: 0x96 checksum

6.6.5. Active Channel Information

Command ID: 0x08

Command Length: Variable

Type: Response

Description: This command sends the QS_ChannelInfo array, which describes the characteristics of each defined channel, including the capacitive sensing method and the threshold detection capabilities.

Command Structure:

Byte 1: 0x08—(command ID)

Byte 2: Variable—(length)

Byte 3: Variable—(channel info)

. . .

Byte n: Variable—checksum

6.6.6. Group Definition

Command ID: 0x10

Command Length: Variable

Type: Response

Description: This command transmits the QS_GroupType array, which describes the type of each defined group in the firmware.

Defined groups are:

0x01 = Slider

0x02 = Control Wheel

Command Structure: Byte 1: 0x10—(command ID)

Byte 2: Variable—(length)

Byte 3: Variable—(Group type)

Byte 4: Variable—(Group type)

. . .

Byte n: Variable—checksum

6.6.7. Group Information

Command ID: 0x18

Command Length: Variable

Type: Response

Description: This command sends the contents of the arrays collected in the QS_GroupChannelAssignmentTable array. One Group information command is sent per defined group.

Command Structure: Byte 1: 0x18—(command ID)

Byte 2: Variable—(length)

Byte 3: Variable—(First channel number)

. . .

Byte n: Variable—(Last channel number)

Byte n+1: Variable—checksum

6.6.8. Threshold Information

Command ID: 0x20

Command Length: Variable

Type: Response

Description: This command transmits the threshold levels for all channels one channel at a time, transmitting the two-byte inactive-to-active threshold before the two-byte active-to-inactive threshold.

Command Structure:

Byte 1: 0x20—(command ID)

Byte 2: Variable—(length)

Byte 3: Variable—(Channel 1 to inactive threshold high byte)

Byte 4: Variable—(Channel 1 to inactive threshold low byte)

Byte 5: Variable—(Channel 1 to active threshold high byte)

Byte 6: Variable—(Channel 1 to active threshold low byte)

. . .

Byte n: Variable—checksum

6.6.9. Reference Baseline Information

Command ID: 0x68

Command Length: Variable

Type: Response

Description: This response transmits the two-byte active and two-byte inactive reference baseline levels for all channels.

Command Structure:

Byte 1: 0x68—(command ID)

Byte 2: Variable—(length)

Byte 3: Variable—(Channel 0 inactive baseline high byte)

Byte 4: Variable—(Channel 0 inactive baseline low byte)

Byte 5: Variable—(Channel 0 active baseline high byte)

Byte 6: Variable—(Channel 0 active baseline low byte)

. . .

Byte n: Variable—checksum

6.6.10. Runtime Baseline Update Rate

Command ID: 0x90

Command Length: 3

Type: Response

Description: This response transmits the 1-byte runtime baseline update rate for all channels.

Command Structure:
Byte 1: 0x90—(command ID)
Byte 2: 0x2 (length)
Byte 3: Variable—(baseline update rate)
Byte 4: Variable—checksum

6.6.11. HISI Update Frequencies

Command ID: 0x28

Command Length: 2

Type: Response

Description: This command transmits the UpdateFrequency array, which defines in Hz the minimum and maximum frequencies that the target firmware can send data to the master.

Command Structure:
Byte 1: 0x28—Command ID
Byte 2: 0x02—Length
Byte 3: variable—(channel update rate in Hz)
Byte 4 variable—checksum

6.6.12. Enumeration Finished

Command ID: 0x30

Command Length: 1

Type: Response

Description: This command tells the master that the target device is finished transmitting all target device capabilities.

Command Structure:
Byte 1: 0x30—(command ID)
Byte 2: 1—(length)
Byte 3: 0x31—checksum

6.7. Operational Commands

6.7.1. Start Transfer

Command ID: 0x18

Command Length: 2

Type: Command

Description: This command tells the target device to send some combination of channel values, threshold states, group positions, and runtime active and inactive baseline levels at a frequency defined by the Set HISI Update Frequency command.

The data payload defines what information should be transferred to the master. Bits 0–3 each define a different type of data that should be transferred.

Bit 0: Raw Data

Bit 1: Group Values

Bit 2: Threshold State

Bit 3: Runtime baseline levels

Command Structure: Byte 1: 0x18—(command ID)

Byte 2: 2—(length)

Byte 3: variable—(Bits to define transferred data)

Byte 4: variable—checksum

Response: Target sends a “no warnings/errors” response if it can support all requested data types. It sends a ”Bad Request” if it cannot support one or more of the requested data types.

6.7.2. Stop Transfer

Command ID: 0x20

Command Length: 1

Type: Command

Description: This command tells the target device to stop sending streaming data.

Command Structure: Byte 1: 0x20—(command ID)

Byte 2: 1—(length)

Byte 3: 0x21—checksum

Response: Device sends a ”no errors/warnings” acknowledgement once the packet is processed.

6.7.3. Set HISI Update Frequency

Command ID: 0x10

Command Length: 2

Type: Command

Description: This command is sent by the master in order to change the frequency at which the target device streams updated channel values, threshold states, group positions, and baselining levels to the master. The data should be in units of Hz.

Command Structure:
Byte 1: 0x10—(command ID)
Byte 2: 2—(length)
Byte 3: variable—(update rate in Hz)
Byte 4: variable—checksum

Response: Device sends a ”no errors/warnings” acknowledgement once the packet is processed.

6.7.4. Capacitive Sense Values

Command ID: 0x38

Command Length: Variable

Type: Response

Description: This command contains channel values found in the QS_ChannelValue array.

Command Structure:
Byte 1: 0x38—(command ID)
Byte 2: variable—(length)
Byte 3: variable—(channel value low byte)
Byte 4: variable—(channel value low byte)
…
Byte X: variable—checksum

6.7.5. Capacitive Sense Group Positions

Command ID: 0x48

Command Length: Variable

Type: Response

Description: Sends all calculated group positions stored in the QS_GroupValue array.

Command Structure:

Byte 1: 0x48—(command ID)

Byte 2: variable—(length)

Byte 3: variable—(grouping data high byte)

Byte 4: variable—(grouping data low byte)

Byte 5: variable—(grouping data high byte)

Byte 6: variable—(grouping data low byte)

. . .

Byte n: variable—checksum

6.7.6. Capacitive Sensing Threshold States

Command ID: 0x40

Command Length: Variable

Type: Response

Description: Sends the threshold states stored in the QS_ThresholdState array, with 2 bits per channel, 4 channels per byte.

Command Structure:

Byte 1: 0x40—(command ID)

Byte 2: variable—(length)

Byte 3: variable—(threshold states, 4 channels).

Byte n: variable—(threshold states, 4 channels)

…

Byte X: variable—checksum

6.7.7. Runtime Baseline Levels

Command ID: 0x78

Command Length: Variable

Type: Response

Description: This response sends runtime inactive and active baseline levels for all channels.

Note: this command is sent by HISI only when a baseline level has been updated since the last data transfer.

Command Structure:
Byte 1: 0x78—(command ID)
Byte 2: variable—(length)
Byte 3: variable—(channel 0 active baseline high byte)
Byte 4: variable—(channel 0 active baseline low byte)
Byte 5: variable—(channel 0 inactive baseline high byte)
Byte 6: variable—(channel 0 inactive baseline low byte)
…
Byte X: variable—checksum

6.7.8. Command Response

Command ID: 0x50

Command Length: 2

Type: Response

Description: Transmitted as described in section “6.5. Errors and Warnings.”

Command Structure:
Byte 1: 0x50—(command ID)
Byte 2: 2—(length)
Byte 3: variable—(error code)
Byte 4: variable—checksum

6.7.9. Enable Threshold Detect

Command ID: 0x28

Command Length: 2

Type: Command

Description: This command enables threshold detection for the channel listed in the data payload.

Command Structure:
Byte 1: 0x28—(command ID)
Byte 2: 2—(length)
Byte 3 variable—(channel number)
Byte 4: variable—checksum

Response: Device sends a ”no errors/warnings” acknowledgement if channel is defined in firmware. Sends ”Bad request” if channel is not defined.

6.7.10. Enable Threshold Detect All

Command ID: 0x30

Command Length: 1

Type: Command

Description: This command enables threshold detection for all active channels.

Command Structure: Byte 1: 0x30—(command ID)

Byte 2: 1—(length)

Byte 3: 0x32—checksum

6.7.11. Disable Threshold Detect

Command ID: 0x38

Command Length: 2

Type: Command

Description: This command disables threshold detection for the channel listed in the data payload.

Command Structure: Byte 1: 0x38—(command ID)

Byte 2: 2—(length)

Byte 3: variable—(channel 0x00–0xFF)

Byte 4: variable—checksum

Response: Device sends a ”no errors/warnings” acknowledgement if channel is defined in firmware. Sends ”Bad request” if channel is not defined.

6.7.12. Disable Threshold Detect All

Command ID: 0x40

Command Length: 1

Type: Command

Description: This command disables threshold detection for all channels listed in the data payload.

Command Structure: Byte 1: 0x40—(command ID)

Byte 2: 1—(length)

Byte 3: 0x41—checksum

6.7.13. Set Inactive-to-Active Threshold

Command ID: 0x48

Command Length: 4

Type: Command

Description: This command sets the two-byte inactive-to-active threshold level for a channel listed in the data payload.

Command Structure:

Byte 1: 0x48—(command ID)

Byte 2: 4—(length)

Byte 3: variable—(channel 0x00—–0xFF)

Byte 4: To Active threshold high byte

Byte 5: To Active threshold low byte

Byte 6: variable—checksum

Response: Device sends a ”no errors/warnings” acknowledgement if channel is defined in firmware. Sends ”Bad request” if channel is not defined.

6.7.14. Set Active-to-Inactive Threshold

Command ID: 0x50

Command Length: 4

Type: Command

Description: This command sets the two-byte active-to-inactive threshold level for a channel listed in the data payload.

Command Structure:

Byte 1: 0x50—(command ID)

Byte 2: 4—(length)

Byte 3: variable—(channel 0x00–0xFF)

Byte 4: To inactive threshold high byte

Byte 5: To inactive threshold low byte

Byte 6: variable—checksum

Response: Device sends a ”no errors/warnings” acknowledgement if channel is defined in firmware. Sends ”Bad request” if channel is not defined.

6.7.15. Get Threshold

Command ID: 0x58

Command Length: 2

Type: Command

Description: This command tells the target device to send upper and lower threshold information for a specified channel.

Command Structure:
Byte 1: 0x58—(command ID)
Byte 2: 2—(length)
Byte 3: variable—(channel 0x00–0xFF)
Byte 4: variable—checksum

Response: Device sends a “no errors/warnings” acknowledgement if channel is defined in firmware. Sends “Bad request” if channel is not defined. If the target sends a “no errors/warnings” response, the target will then send a Get Threshold Response command.

Description: This command tells the target device to send inactive-to-active and inactive-to-active threshold levels for a specified channel.

6.7.16. Get Threshold Response

Command ID: 0x58

Command Length: 6

Type: Response

Description: This command tells the target device to send the inactive-to-active threshold and the active-to-inactive threshold for a specified channel.

Command Structure:
Byte 1: 0x58—(command ID)
Byte 2: 6—(length)
Byte 3: variable—(active to inactive threshold high byte 0x00–0xFF)
Byte 4: variable—(active to inactive threshold low byte 0x00–0xFF)
Byte 5: variable—(inactive to active threshold high byte 0x00–0xFF)
Byte 6: variable—(inactive to active threshold low byte 0x00–0xFF)
Byte 7: variable—checksum

6.7.17. Reset

Command ID: 0x60

Command Length: 1

Type: Command

Description: This command tells the target device to issue a software reset, sending the system back to its initial state.

Command Structure: Byte 1: 0x60—(command ID)

Byte 2: 0x01—(length)

Byte 3: 0x61—checksum

6.7.18. Erase Non-Volatile Calibration And Configuration Area

Command ID: 0x68

Command Length: 1

Type: Command

Description: This command tells the target device to erase the page(s) of flash memory that contains the threshold levels, reference baseline levels, and runtime baseline update settings.

Command Structure: Byte 1: 0x68—(command ID)

Byte 2: 0x01—(length)

Byte 3: 0x69—checksum

Response: Device sends a “no errors/warnings” acknowledgement if THOLD_SAVE is ENABLED. Sends “Bad request” if THOLD_SAVE is DISABLED.

6.7.19. Set Runtime Baseline Update Rate

Command ID: 0x88

Command Length: 2

Type: Command

Description: This command saves a one-byte value configuring the baseline update per channel (in units of seconds) to the non-volatile storage area.

Command Structure: Byte 1: 0x68—(command ID)

Byte 2: 0x02—(length)

Byte 3: Variable—(Runtime Baseline update rate)

Byte 4: Variable—checksum

Response: Device sends a “no errors/warnings” acknowledgement once command is processed.

6.7.20. Get Runtime Baseline Update Rate

Command ID: 0x90

Command Length: 2

Type: Command

Description: This command requests a one-byte runtime baseline update rate in the non-volatile storage area. The system responds by sending the Runtime Baseline Update Rate response that was also sent during enumeration.

Command Structure:
Byte 1: 0x90—(command ID)
Byte 2: 0x02—(length)
Byte 3: Variable—(Runtime Baseline update rate)
Byte 4: Variable—checksum

Response: Device sends a Runtime Baseline Update Rate response.

6.7.21. Set Inactive Reference Baseline

Command ID: 0x78

Command Length: 4

Type: Command

Description: This command transmits the inactive reference baseline to be saved to the target device's non-volatile storage area.

Command Structure:
Byte 1: 0x78—(command ID)
Byte 2: 4—(length)
Byte 3: variable—(channel 0x00–0xFF)
Byte 4: variable—(channel inactive reference baseline high byte)
Byte 5: variable—(channel inactive reference baseline low byte)
Byte 6: variable—checksum

Response: Device sends a "no errors/warnings" acknowledgement if channel is defined in firmware. Sends "Bad request" if channel is not defined.

6.7.22. Set Active Reference Baseline

Command ID: 0x80

Command Length: 4

Type: Command

Description: This command transmits the active reference baseline to be saved to the target device's non-volatile storage area.

Command Structure:
Byte 1: 0x80—(command ID)
Byte 2: 4—(length)
Byte 3: variable—(channel 0x00–0xFF)
Byte 4: variable—(channel active reference baseline high byte)
Byte 5: variable—(channel active reference baseline low byte)
Byte 6: variable—checksum

Response: Device sends a "no errors/warnings" acknowledgement if channel is defined in firmware. Sends "Bad request" if channel is not defined.

6.7.23. Get Reference Baseline

Command ID: 0x70

Command Length: 2

Type: Command

Description: This command requests that a channel's reference baseline values be transmitted back to the master.

Command Structure:
Byte 1: 0x70—(command ID)
Byte 2: 2—(length)
Byte 3: variable—(channel 0x00–0xFF)
Byte 4: variable—checksum

Response: Device sends a set of reference baseline values for the channel requested if channel is defined. Sends "Bad request" if channel is not defined.

6.7.24. Get Reference Baseline Response

Command ID: 0x70

Command Length: 4

Type: Response

Description: This command transmits the active and inactive reference baseline requested by the Get Reference Baseline command.

Command Structure:

Byte 1: 0x70—(command ID)

Byte 2: 4—(length)

Byte 3: variable—(channel inactive reference baseline high byte)

Byte 4: variable—(channel inactive reference baseline low byte)

Byte 5: variable—(channel active reference baseline high byte)

Byte 6: variable—(channel active reference baseline low byte)

Byte 7: variable—checksum

6.7.25. Set Flash Key Codes

Command ID: 0xA0

Command Length: 3

Type: Command

Description: This command writes to the Flash key codes on the target device. It allows the master to enable or disable Flash writes/erases.

Command Structure:

Byte 1: 0xA0—(command ID)

Byte 2: 3—(length)

Byte 3: variable—(first Flash key code)

Byte 4: variable—(second Flash key code)

Byte 5: variable—checksum

6.7.26. Channel Calibration Check

Command ID: 0xA0

Command Length: 2

Type: Command

Description: Requests that the firmware perform a check on a channel's calibration values and respond indicating whether calibration values are valid or not.

Command Structure:

Byte 1: 0x98—(command ID)

Byte 2: 2—(length)

Byte 3: variable—(channel value)

Byte 4: variable—checksum

6.7.27. Channel Calibration Check Response

Command ID: 0xA0

Command Length: 2

Type: Response

Description: Transmits whether a channel's calibration values have been saved. During enumeration, this command is sent once per channel, each instance of the response corresponding to each channel in ascending order. After a master transmits the Channel Calibration Check command, the response payload's information refers to the channel requested by the command.

Valid calibration check response values:

0x00—channel not calibrated

0x01—valid channel calibration information saved to Flash

Command Structure: Byte 1: 0xA0—(command ID)

Byte 2: 2—(length)

Byte 3: variable—(channel calibration information)

Byte 4: variable—checksum

6.7.28. Firmware Revision And Capabilities

Command ID: 0xA8

Command Length: 1

Type: Command

Description: Requests that the target device transmit the Firmware Revision And Capabilities response.

Command Structure: Byte 1: 0xA8—(command ID)

Byte 2: 1—(length)

Byte 3: 0xA9—checksum

6.8. Changes from Version 1.0 to Version 1.1

- Updated description of packet structure to describe the interface's 11-bit length field
- Added Baseline Information response, which transmits saved idle and active reference baseline information during enumeration
- Added bit to Start Transfer command's parameter to enable/disable streaming runtime baseline levels during operation
- Added Firmware Revision and Capabilities response to describe firmware revision and algorithmic capabilities of the device at enumeration
- Added Runtime Baseline levels command, which transmits runtime baseline levels for all channels, at a frequency defined by the HISI update frequency
- Changed name "Erase Thresholds" to "Erase Non-Volatile Calibration and Configuration Area"
- Added Set Baseline Update Rate command, which configures the speed at which baseline levels for a channel are updated
- Added Set Active Reference Baseline command and Set Inactive Reference Baseline command, which write an active or inactive 2-byte reference baseline to Flash memory for a specified channel.
- Added Get Reference Baseline, which allows the master to retrieve inactive and active baseline levels for a requested channel.
- Get Reference Baseline Response is the target's response to Get Reference Baseline, transmitting inactive and active reference baseline levels (4 bytes total)
- Changed "Update Frequency" to "Update Human Interface Serial Interface Frequency"
- Removed "Command Name" fields from command descriptions
- Added Get Runtime Baseline Update Rate command, which forces the target device to send a Runtime Baseline update rate response back to the master
- Added Set Flash Key Codes command that allows the master to enable or disable Flash writes/erases
- Added Channel Calibration Check command and response to allow master to determine whether a given channel has valid calibration data

Contact Information

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page:
https://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.